Yo Cmás

Comunidad del taller avanzado

18 de diciembre de 2020

Índice general

Esta es una documentación realiza por Mateo León, Vicente Villaroel, alumnos del Taller Yo C+ y el tutor Gabriel Carmona, tutor del taller de nivel avanzado.

La principal idea de este documento es realizar un espacio en el cual se presente la materia vista a través del taller y explicaciones a los contest que se han hecho a través del tiempo.

Cualquier aporte es bienvenido.

CAPÍTULO 1

Estructuras de Datos

Las estructuras de datos son una forma que el programador tiene para organizar los datos de manera eficiente. Existen diferentes tipos de estructuras de datos, algunas que se pueden aplicar a muchos temas mientras que otras son especializaciadas en tareas específicas.

En este curso vamos a revisar dos tipos: estructuras de datos lineales y las no lineales.

1.1 Estructuras de Datos Lineales

1.1.1 Acerca de

Estas estructuras se basan en que los elementos estan contiguos, esto quiere decir que los elementos estan ordenados de manera que un elemento va a después del otro.

1.1.2 Arrays

Los arrays son conjuntos de datos que se almacenan en memoria de manera contigua usando el mismo nombre. Se usan índices para diferenciar los distintos valores del arreglo.

Para crear una arreglo es necesario saber la cantidad de elementos que van a componer nuestro arreglo, pues la cantidad de elementos no se puede cambiar.

Una cosa muy importante que al principio uno suele olvidar es que los índices de los arreglos parten en 0. Es decir, un arreglo de tamaño n puede tomar índices desde el 0 hasta el n-1.

Inicialización

Hay dos formas de inicializar un array, especificando su tamaño o llenandolo con valores por defecto.

Inicializar un array por su tamaño

Para inicializar un array por su tamaño, lo único que hay que especificar es el nombre de la variable y colocar entre «[]» el tamaño de nuestro arreglo. Recordar que como los arrays parten en 0, el valor máximo que podremos acceder será el tamaño del array menos 1.

```
    // Inicializar un arreglo de ints de tamaño 38.
    int arreglo[38];
```

Inicializar un array con valores por defecto

Para inicializar un array con valores por defecto, no es necesario especificar el tamaño de nuestro array, pues este está implícito en la cantidad de valores que introducimos.

```
    // Esto genera el arreglo siguiente
    //[3, 6, 2, 1]
    int arreglo2[] = {3, 6, 2, 1};
```

Leer el valor de un arreglo

Para leer un valor de nuestro arreglo, solo hay que especificar el nombre del arreglo y su índice. Notar que sólo se puede acceder a un valor de un arreglo a la vez.

```
    // Retorna 1
    arreglo2[3];
```

Asignar valores a un arreglo

Para esto, hay que especificar la variable, el índice al cuál acceder y especificar su nuevo valor. Recordar que el valor del índice puede ser una variable, pero que no puede estar fuera del rango del tamaño de nuestro arreglo.

```
    // El indice 3 de nuestro arreglo, ahora tiene el valor 3
    arreglo2[3] = 5;
```

1.1.3 Arrays multidimensionales

Un array multidimensional es aquel que requiuere de más de un índice para ser llamado, como su nombre lo indica, es útil para cuando necesitamos acceder a datos que requieren más de una dimensión, como por ejemplo valores dentro de una malla. Otra forma de enternder los arrays multidimensionales es como un arreglo de arreglos (de arreglos de arreglos.. n veces, siendo n la cantidad de dimensiones). Los arreglos de 2 dimensiones también son conocidos como matrices.

Inicializar un arreglo multidimensional

Para esto, hay que especificar el tamaño de cada dimensión del array. Notar que la regla de que empiezan en 0 y terminan en n-1 se sigue cumpliendo. Es posible asignarles valores predeterminados, pero creo que eso se va poniendo exponencialmente más psicópata a medida de que vas incrementando las dimensiones del array a crear, por lo que no lo voy a demostrar.

```
    int matriz[3][4];
    int tridimensional[100][100][100];
```

Asignar valores en un arreglo multidimensional

Es muy similar a cómo se hace en un arreglo unidimensional, solo que se especifica cada índice.

```
    int n = 3, m = 5;
    matriz[2][1] = 10;
    tridimensional[n][m] = 4;
```

Vectors (vectores) {#vectors–vectores}

Los vectores son como arreglos, excepto de que el tamaño es dinámico, es decir, se puede cambiar.

Inicializar un vector

Incluimos la librería:

```
    #include <vector>
```

Inicializamos nuestro vector «vec»:

```
    // Inicializa un vector de tamaño 3 con todos sus valores = 0
    // Tanto el tamaño como valor son opcionales.
    int n = 3;
    vector < int > vec(n, 0);
```

Asignar un valor

```
    // Asigna el valor "1" al índice 2 (es decir, al 3er valor del vector)
    vec[2] = 1;
```

push back (empujar atrás)

Si no sabemos el tamaño de nuestro vector, podemos simplemente usar push_back(valor); para enviar es valor al final del vector.

```
    // Inserta un 1 al final del vector
    vec.push_back(1);
```

Por ejemplo, se podría usar en un for, sin necesidad de inicializar el vector con una cantidad de valores.

```
    vector <int> vec2;
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i){
        int valor;
        cin >> valor;
        vec2.push_back();
    }
```

pop back (quitar atrás)

Elimina el último valor del vector.

```
    // En este caso, elimina el 1
    vec.pop_back();
```

insert (insertar)

Podemos insertar un valor entre dos indices de un vector. El problema de esto es que mueve todos los valores que estén más adelante, lo que es lento.

```
    // Inserta el valor 4 al índice 2
    vec.insert(vec.begin() + 2, 4);
```

erase (borrar)

Borra un dato del vector. Al igual que el insert, tiene que mover todos los datos siguientes (esta vez a la derecha).

```
    // Elimina el valor con índice 2
    // En nuestro caso, el 4 que insertamos antes.
    vec.erase(vec.begin() + 2);
```

Iteradores de un arreglo

Hay ciertos iteradores que podemos usar en un arreglo que nos ayudarán en algunos casos, como por ejemplo si quieremos recorrer un arreglo. Estos son:

- begin() – Iterador que accede al primer valor del arreglo.
- end() – Accede al final del arreglo.
- rbegin() – Accede al ultimo elemento del arreglo
- rend – Accede al inicio del arreglo

1.1.4 Stacks (pilas)

La pila es una estructura de datos lineal al que sólo puedes acceder al último elemento que fue insertado. Imagina una pila de platos, por ejemplo.

```
    stack < int > pilita;
```

push (empujar)

Empuja un dato a la cima de la pila.

```
    // Empuja un 8 a la cima de la pila.
    pilita.push(8);
```

top (cima)

Lee lo que hay en la cima de la pila.

```
    // Siguiendo el ejemplo anterior
    // Esto retorna 8
    pilita.top();
```

pop (quitar)

Remueve el dato de la cima de la pila.

```
    // Siguiendo el ejemplo anterior
    // Remueve el 8
    pilita.pop();
```

empty (vacío)

Retorna 1 si la pila está vacía, de lo contrario retorna 0.

```
    // Retorna 1 ya que nuestra pila está vacía.
    pilita.empty();
```

size (tamaño)

Retorna el tamaño de nuestra pila.

```
    // Retorna 0 ya que nuestra pila no tiene datos.
    pilita.size();
```

1.1.5 Queues (colas)

La cola es una estructura de datos lineal al que sólo puedes acceder al primer elemento que fue insertado. Imagina una fila de una caja de un supermercado, por ejemplo.

```
    queue < int > colita;
```

push (empujar)

Añade un dato al final de la cola.

```
    colita.push(5);
    colita.push(4);
    colita.push(3);
    colita.push(2);
    colita.push(1);
```

front (frente)

Lee el dato que está al frente de la cola.

```
    // Siguiendo el ejemplo construido antes
    // Retorna 5, ya que fue lo primero que empujamos a la cola
    colita.front();
```

pop (quitar)

Remueve el dato que está al frente de la cola

```
    // Remueve el 5
    colita.pop();
    // Retorna 4, ya que fue lo segundo que empujamos a la cola (y que ahora está␣
↪primero).
    colita.front();
```

empty (vacío)

Retorna 1 si la cola está vacía, de lo contrario retorna 0.

```
    // Retorna 1 ya que nuestra cola está vacía.
    colita.empty();
```

size (tamaño)

Retorna el tamaño de nuestra cola.

```
    // Retorna 0 ya que nuestra cola no tiene datos.
    colita.size();
```

1.2 Estructuras de Datos No Lineales

1.2.1 Acerca de

A diferencia de las EDD lineales, las no lineales se basan en el hecho de que ahora los elementos no estan contiguos, sino que la forma de buscarlos sigue alguna otra lógica que no sea mirar el siguiente.

Algunos ejemplos son:

- Árboles binarios
- Tablas de hashing

1.2.2 Set (conjunto)

Es una EDD que consiste en organizar elementos no repetidos. Esto quiere decir que cada elemento va a encontrar una y una sola vez en el conjunto.

Inicialización

Incluimos la librería:

```
    #include <set>
```

Inicializamos nuestro conjunto:

```
    set < int > conjunto; // int puede ser reemplazado con cualquier otro tipo de dato
```

insert (Insertar)

Inserta un dato. Retorna un par de elementos, el primero siendo el iterador del valor insertado y el segundo siendo un bool que marca si es que ya existía o no. En el ejemplo de abajo, usamos .second para comprobar si se insertó correctamente o no.

```
    // Retorna TRUE ya que no estaba anteriormente
    if (conjunto.insert(10).second)
            // Retorna FALSE ya que ya había un 10.
        cout << "ganai\n"; if (conjunto.insert(10).second);
    cout << "no ganai\n";
    conjunto.insert(20);
    conjunto.insert(40);
    conjunto.insert(30);
    conjunto.insert(11);
```

find (Encontrar)

Busca un elemento en el set y si lo encuentra retorna un iterador al valor. De lo contrario, retorna conjunto.end();

```
    if (conjunto.find(10) != conjunto.end())
        cout << "ganai\n";
```

erase (borrar)

Puedes borrar un valor si le entregas el iterador al valor.

```
    set < int >.. code:: cpp
:number-lines: iterator it = conjunto.find(11);
    if (it != conjunto.end())
        conjunto.erase(it);
```

Iterar a través de un conjunto

Puedes iterar a través de un conjunto con los valores ya ordenados con un iterador:

```
    // Imprime 10 11 20 30 40
    for (it = conjunto.begin(); it != conjunto.end(); ++it)
        cout << *it << " ";
    cout << '\n';
```

1.2.3 Map (mapa, tabla de hashing)

Toma dos datos, una llave y un valor. Puedes buscar una llave en tiempo logarítmico con la implementación de la STL. Pero con otras implementaciones se puede hacer en tiempo constante. Las llaves no se pueden repetir.

Ejemplo cotidiano

- Libros:

Título (Llave)	Autor (Valor)
The C Programming Language	Brian Keringhan
The AWK Programming Language	Brian Keringhan
1984	George Orwell

- Curso:

Apellido	Cantidad de alumnos con el apellido
Gonzalez	3
Perez	2

Inicializar {#inicializar}

Incluimos la librería de map:

```
    #include <map>
```

Inicializamos el mapa curso:

```
    map<string, int> curso;
```

Insert (insertar)

- Forma 1:

```
    curso["perez"] = 1;
```

- Forma 2:

```
    curso.insert(pair<string, int>("gonzalez, 3"));
```

Operar con los valores

Se puede operar con el valor tomando la llave. Ejemplo 1:

```
    // Incrementar el valor de la llave perez, por ejemplo.
    ++curso.["perez"];
```

Ejemplo 2:

```
    cout << curso.["perez"] << endl; // El output será 2.
```

Cuidado con operar con valores no existentes, pues los inicializará de una forma inesperada.

Find (encontrar)

Retorna un iterador, si no lo encuentra, apunta a map.end(). Asignamos el iterador `it` a gonzalez, y luego lo usamos:

```
    map<string, int>.. code:: cpp
:number-lines: iterator it;
    it = curso.find("gonzalez");

    if (it != curso.end()){
            cout << "Hay " << it->second << " " << it->first << " en el curso:\n";
            cout << "Llave: " << it->first << " Valor: " << it->second << '\n';
    }
```

Podemos incluso operar usando los iteradores:

```
    it->++second;
```

Erase (borrar)

- Forma 1:

```
it = curso.find("perez");
curso.erase(it);
```

- Forma 2:

```
curso.erase("gonzalez");
```

Recorrer los valores de un mapa

Es exactamente igual que en un conjunto:

```
for (it = curso.begin(); it != curso.end(); ++it){
    cout << "Llave: " << it->first << " Valor: " << it->second << '\n';
}
```

Dudas que no dejan dormir

1. ¿Qué pasa si modifico una llave?

No se puede, tu código no compilará pues es ilegal hacerlo

2. ¿Puedo buscar con el second?

No, en ese caso recomendamos otra estructura, o tener dos maps

3. ¿Puedo tener un map dentro de un map?

Si, pero es de psicópata buscar dentro de ese map.

CAPÍTULO 2

Algoritmos de Ordenamiento

Uno de los problemas más clásicos de la informática es el problema de ordenamiento, este corresponde a tomo un grupo de datos y los ordenan a base de algún criterio. Hay muchos algoritmos actualmente, vamos a revisar en detalle tres algoritmos. Dos que corresponden a algoritmos que siguen la técnica de fuerza bruta y uno que consiste a decrecer y conquistar. Estos si bien no son los algoritmos más eficientes, son muy útiles para entende la base del ordenamiento.

2.1 Algoritmos de Fuerza Bruta

2.1.1 Acerca de

Los algoritmos de fuerza bruta son algoritmo en general simples, que realizan una idea de “”Todos con todos””. Esto puede llegar a ser muy costoso en complejidad.

2.1.2 Bubble sort

- Se van comparando los elementos, haciendo que los más grandes suban en el arreglo, como una burbuja.
- Da lo mismo cómo estén ordenadas las cosas, porque puedes modificar la función de comparación.
- Funciona invirtiendo el orden de cada par de elementos, si es que el primero es mayor que el segundo.
- Por ejemplo, se podría usar para contar el número de inversiones que hay que hacer.

Ejemplo

Step	Pos 0	Pos 1	Pos 2	Pos 3
0	5	3	4	1
1	*5*	*3*	4	1
2	3	*5*	*4*	1
3	3	4	*5*	*1*
4	3	4	1	**5**
5	*3*	*4*	1	**5**
6	3	*1*	*4*	**5**
7	3	1	**4**	**5**
8	*3*	*1*	**4**	**5**
9	1	**3**	**4**	**5**
10	**1**	**3**	**4**	**5**

Código

```
void swap(vector<int> &arr, int i, int j){
    int aux = arr[i];
    arr[i] = arr[j];
    arr[j] = aux;
}

void bubblesort (vector<int> &vec) {
    int size = vec.size();
    for (int i = size - 1; i > 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (vec[j] > vec[j+1]) {
                swap(arr, j, j + 1);
            }
        }
    }
    return;
}
```

2.1.3 Insertion sort

- Se va de izquierda a derecha, se compara el segundo con el primero, se intercambian si el segundo es menor, si este es el caso, se vuelve a preguntar si el de la izquierda es menor al de mas a la izquierda y así hasta que se encuentre un caso en el que no o se llegue al principio del arreglo.

Ejemplo

Step	Pos 0	Pos 1	Pos 2	Pos 3
0	5	3	4	1
1	*5*	*3*	4	1
2	**3**	5	4	1
3	3	*5*	*4*	1
4	*3*	*4*	5	1
5	**3**	**4**	5	1
6	3	4	*5*	*1*
7	3	*4*	*1*	5
8	*3*	*1*	**4**	**5**
9	**1**	**3**	**4**	**5**

Código

```
void swap(vector<int> &arr, int i, int j){
    int aux = arr[i];
    arr[i] = arr[j];
    arr[j] = aux;
}

void insertion_sort(vector<int> &arr){
    for(int i = 1; i < arr.size(); ++i){
        for(int j = i; j >= 0; --j){
            //caso swap
            if(arr[j] < arr[j-1]){
              swap(arr, j, j-1);
            }
            //caso no swap = todo ok todo correcto :]
            else if(arr[j] >= arr[j-1] || j == 0){
                break;
            }
        }
    }
    return;
}
```

Nota: este código fue robado de Julieta Coloma

2.2 Algoritmos de Decrecer y Conquistar

2.2.1 Acerca de

Rellenar

2.2.2 Selection sort

- Tiene dos sub-arreglos, uno de elementos ya ordenados y uno de los elementos resantes.
- El arreglo ya ordenado parte vacío.
- Busca el valor mínimo entre los elementos no ordenados y lo añade al final de los ordenados.

Ejemplo

Step	Pos 0	Pos 1	Pos 2	Pos 3
0	5	3	4	1
1	5	3	4	*1*
2	**1**	5	*3*	4
3	**1**	**3**	5	*4*
4	**1**	**3**	**4**	**5**

Código de ejemplo

```
1   void swap(vector<int> &vec, int i, int j){
2           int aux = vec[i];
3           vec[i]=vec[j];
4           vec[j]= aux;
5   }
6
7   void selectionSort(vector <int> &vec){
8           int size = vec.size(); //tamaño :D
9           for(int i = 0; i < size - 1; i++){ //
10                  int min_in = i;
11                  for(int j= i + 1; j < size; j++){
12                          if(vec[j] < vec[min_in]){
13                                  min_in=j;
14                          }
15                  }
16                  swap(vec, min_in, i);
17          }
18  }
```

Nota: este código fue robado de Natalia Carrión

CAPÍTULO 3

Algoritmos de búsqueda

Otro problema muy recurrente dentro de la programación es el buscar uno o más elementos dentro de un grupo de datos. Para ello se han desarrollado varios algoritmos de tal forma de optimizar esta busqueda.

Los dos principales y más nombrados son la búsqueda lineal y binaria. Estas dos búsquedas se detallarán para comprender su uso y benificio.

3.1 Búsqueda Lineal

3.1.1 Acerca de

La búsqueda lineal es un algoritmo se trata de pasar elemento por elemento hasta encontrar el correcto. Esto lo hace de forma secuencial, de ahí el nombre lineal.

Entonces analizando, si uno tiene **N** elementos toma un elemento y ve si es el correcto. Si corresponde listo ganamos, si no pasa al siguiente. Esto lo repite hasta el último elemento.

Este algoritmo tiene complejidad **O(N)**. La principal gracia de este algoritmo es cuando el grupo de elementos no están ordenados por lo cual uno no sabría como moverse. En la siguiente sección se revisará un algoritmo para cuando el grupo este ordenado.

3.1.2 Ejemplo

Step	Pos 0	Pos 1	Pos 2	Pos 3	Ele
0	4	2	5	1	5
1	**4**	2	5	1	5
2	4	**2**	5	1	5
3	4	2	**5**	1	5

Aquí se ve como en la iteración número 3 encuentra el elemento por lo deja de revisar :D.

3.1.3 Código

```cpp
int linealSearch(vector< int > &vec, int ele){
    for(int i = 0; i < vec.size(); i++){
            if(vec[i] == ele){
                    // Retorna la posicion en que se encuentra
                    return i;
            }
    }
    // Retorna -1 en caso de que el elemento no este
    return -1;
}
```

3.2 Búsqueda Binaria

3.2.1 Acerca de

La búsqueda binaria es un algoritmo de *divide and conquer* (dividir y conquistar), que nos permite encontrar un elemento dentro de una estructura **ordenada** rápidamente. Al ejecutarse, toma el centro de un arreglo y comprueba si el valor que se busca es igual al del centro. De no serlo, verifica si el valor es menor o mayor al del centro.

Si el valor es mayor al del centro, se ignoran todos los valores anteriores al centro, dividiendo la cantidad de números a la mitad.

Si el valor es menor al del centro, se ignoran todos los valores de después del centro, dividiendo la cantidad de números a la mitad.

La complejidad de este algoritmo es **O(log(N))**, comparada con un algoritmo lineal, que en el peor de los casos tiene complejidad **O(N)**.

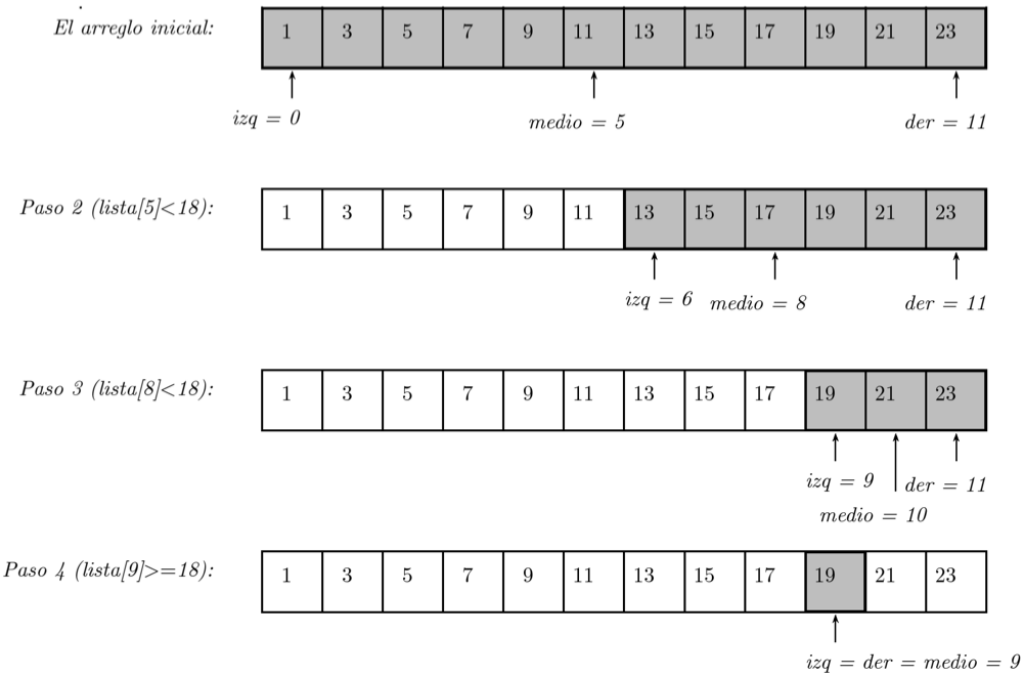
Pero hay que notar que para aplicar este algoritmo uno debe tener el grupo ordenado, debido a que si no esta ordenado no hay forma de poder saber hacia donde moverse según la respuesta dada a la comparación.

1. Ejemplos
    1. en la vida real

        Ir al medio de un diccionario, y buscar alfabéticamente, tomando una palabra central (más o menos), viendo si la palabra que queremos está antes o después y repetir el proceso de buscar una palabra central.

    2. Ejemplo computacional

        En esta imágen, se usa la búsqueda binaria para encontrar el 19.



3.2.2 Uso mediante la librería STL

- Importar

  Podemos directamente importar toda la stl o podemos importar la librería <algorithm> de la siguiente forma:

```
#include <algorithm>
```

- binary_search()

  La librería STL ya incluye binary search, si queremos saber si el valor 3 está en un vector, podemos ejecutar:

```
vector<int> v{1,2,5,7};
if (binary_search (v.begin(), v.end(), 3)) {
  cout << "Se encuentra el valor 3 en nuestro vector\n";
}
else {
  cout << "No hay ningún 3 en nuestro vector\n";
}
```

Retorna un *bool*.

- lower_bound() (límite inferior)

  La función lower_bound() de la librería STL retorna un puntero a un valor **superior** o, si es posible, **igual** al entregado dentro de una estructura ordenada.

  Si todos los elementos en el arreglo son inferiores al valor pedido, se entrega el último elemento del arreglo. Si los elementos del arreglo son superiores al valor pedido, se entrega el primer elemento del arreglo.

  Por ejemplo:

```
vector<int> v{ 10, 20, 30, 30, 30, 40, 50 };
*lower_bound(v.begin(), v.end(), 30);
```

Valor de retorno: un iterador que apunta hacia **30** (el primero en el arreglo, en la posición [2])

- upper_bound() (límite superior)

  La función $\text{upper}_{\text{bound}}()$ de la librería STL nos entrega un puntero a un valor **superior** al pedido en un arreglo ordenado.

  En el caso de que no haya un valor superior al pedido, nos entrega el último valor del arreglo.

  Por ejemplo:

```
*upper_bound(v.begin(), v.end(), 30);
```

Valor de retorno: iterador al **40** (posición [4])

3.2.3 Implementaciones propias

Implementación con while

Esta implementación nos retorna el índice del número a buscar dentro de un arreglo.

```
int binarySearch(int arr[], int l, int r, int x)
{

    while (l <= r) {
        // Esto es lo mismo que hacer (r + l) / 2
        int m = l + (r - l) / 2;

        // Revisa si x esta al medio
        if (arr[m] == x){
            return m;
        }

        // Si x es mayor, ignorar la izquierda
        if (arr[m] < x){
            l = m + 1;
        }
        // Si x es menor, ignorar la derecha
        else{
            r = m - 1;
        }
    }
    return -1;

}
```

Implementación recursiva

Esta implementación nos retorna el índice del número a buscar dentro de un arreglo.

```
int binarySearch(int arr[], int l, int r, int x){
    if (r >= l){
        // Esto es lo mismo que hacer (r + l) / 2
        int mid = l + (r - l) / 2;

        // Revisa si x esta al medio

        if (arr[mid] == x){
            return mid;
        }

        // Si x es mayor, ignorar la izquierda
        if (arr[mid] > x){
            return binarySearch(arr, l, mid - 1, x);
        }
        // Si x es mayor, ignorar la derecha
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```