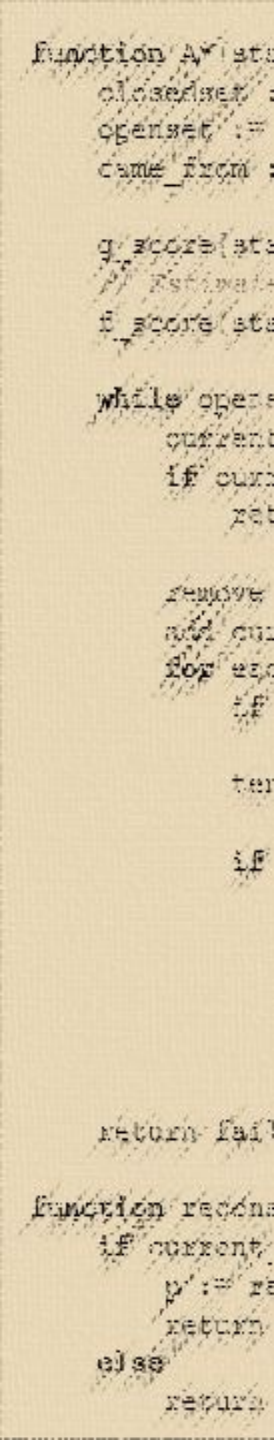


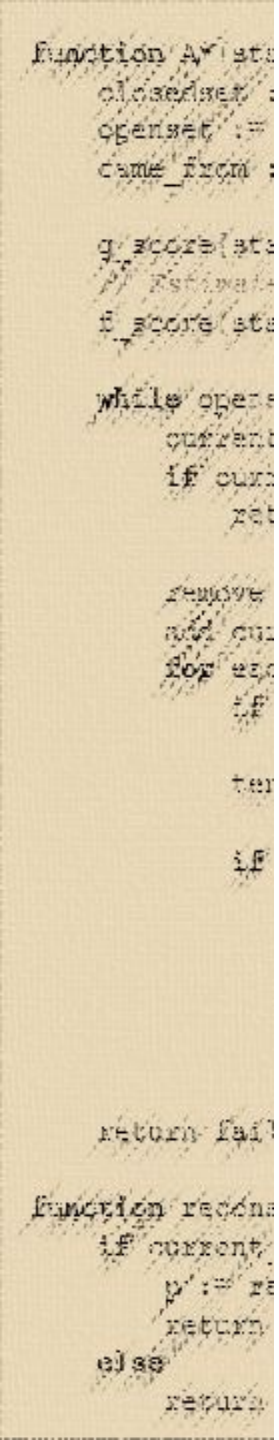
Tecnológico de Monterrey

Algoritmos de búsqueda

Estructura de Datos

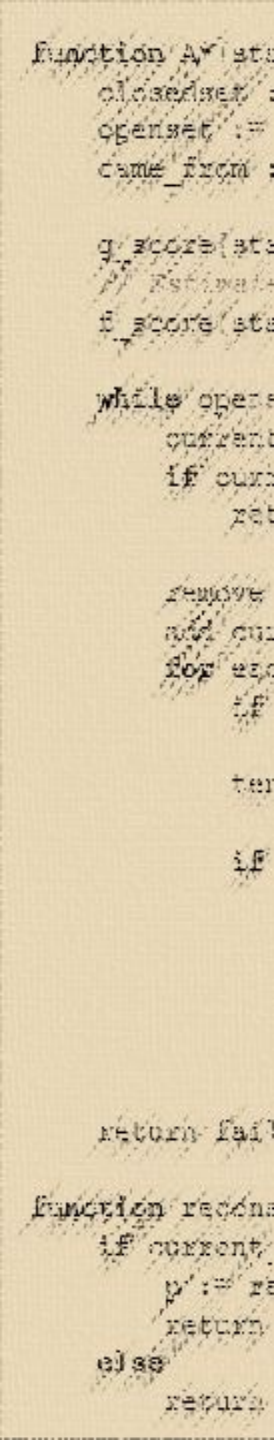
Víctor de la Cueva

vcueva@itesm.mx



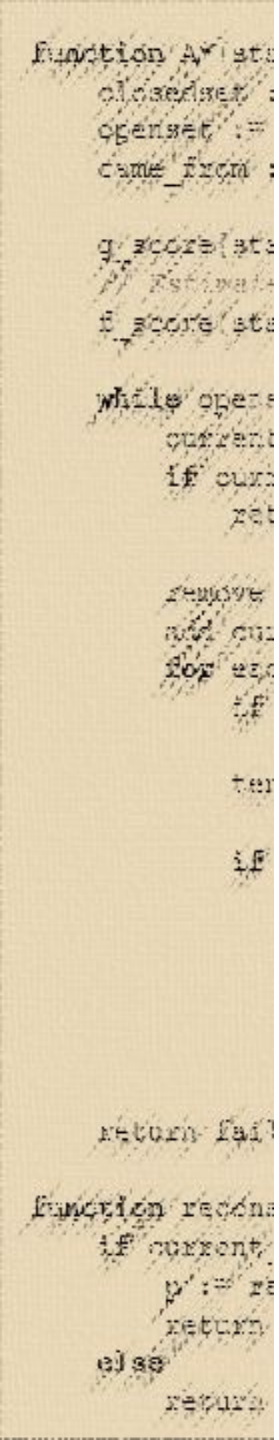
Búsqueda

- En computación, “buscar” significa ver si un dato particular se encuentra o no en un conjunto de datos guardados que se encuentran en una estructura de datos particular.
- Un algoritmo de búsqueda recibe un dato, el cual busca, generalmente, en un arreglo, y regresa como respuesta “verdadero”, si el dato se encuentra en el arreglo, o “falso”, si no se encuentra.
- Es muy común que, en caso de que el dato sí se encuentre, regrese la posición del arreglo en la que está. En caso de que no se encuentre puede regresar un número negativo.



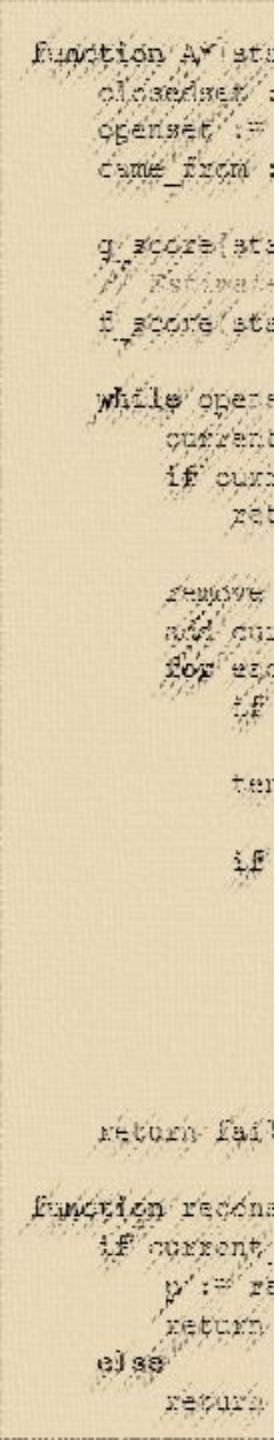
Importancia

- Aunque la búsqueda no es una estructura de datos, es una de las operaciones más importantes que complementa el uso de cualquier estructura de datos.
- Almacenar información en una estructura de datos tiene sentido si después se puede tener acceso a ella, para lo cual se requiere el uso de algoritmos de búsqueda.
- Esta operación es la que permite recuperar la información previamente almacenada.



Búsqueda en arreglos

- Los métodos o algoritmos de búsqueda se aplican a cualquier estructura de datos, sin embargo, como todavía no vemos ninguna estructura específica, nos concentraremos en búsquedas en arreglos.
- Hay tres formas generalizadas de búsquedas en arreglos:
  - Secuencial: para cualquier tipo de arreglo.
  - Binaria: sólo para arreglos ordenados.
  - Transformación de claves: funciones Hash.
- Por el momento sólo trataremos las dos primeras pero más adelante veremos la tercera y cómo hacer búsquedas en otras estructuras de datos.



Búsqueda secuencial

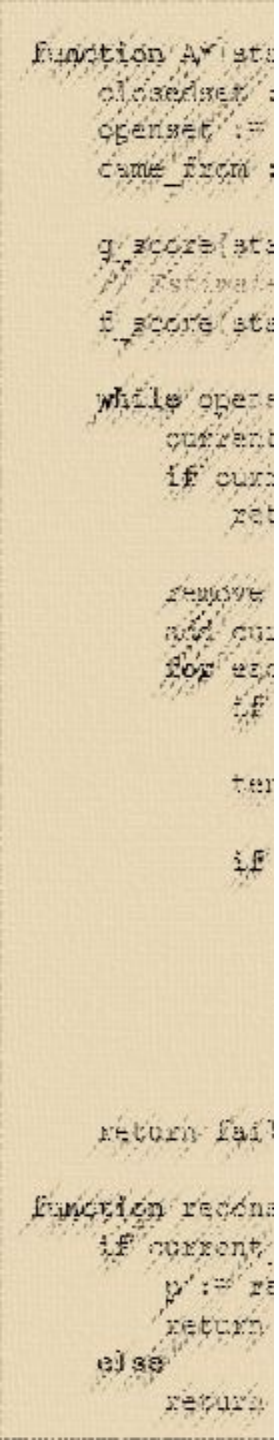
- La forma trivial (*naive*) de buscar en un arreglo es hacerlo elemento por elemento.
  - Si en algún momento encontramos el elemento buscado regresamos su posición en el arreglo.
  - Si llegamos al final y no lo hemos encontrado, regresamos algo que diga que no lo encontramos (normalmente es un -1 ya que no hay posiciones negativas en los arreglos).
- ¿Cuánto se va a tardar en responder?
  - Depende de si está o no el dato, y de en qué posición está.
  - ¿Se puede analizar el trabajo realizado?

Algoritmo: Búsqueda secuencial

Entradas: Un arreglo E con n elementos (indexados de 0 a n-1) y el elemento a buscar K (por sencillez suponemos que son enteros).

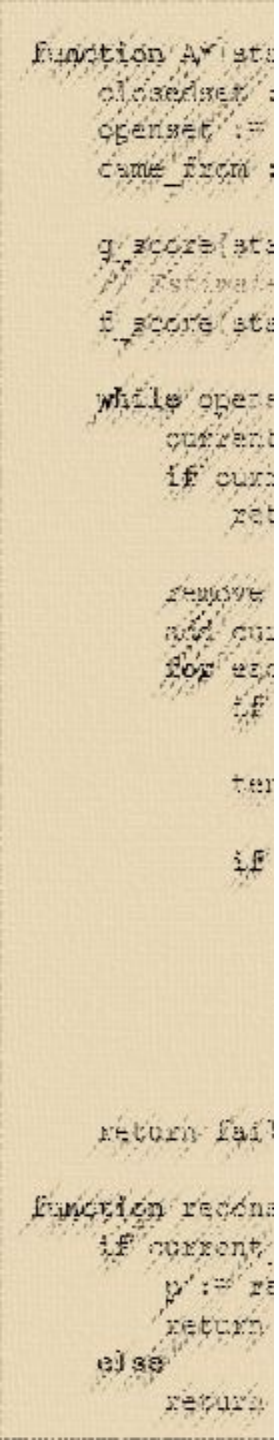
Salidas: Regresa la ubicación de K en E o -1 si no está.

```
int busquedaSecuencial(int[] E, int n, int K)
   int respuesta, índice;
   respuesta = -1; // suponer fracaso
   for (índice = 0; índice < n; índice ++)
      if (K == E[indice])
         respuesta = indice; // Éxito
         break;
   return respuesta;
```



¿Se puede hacer mejor?

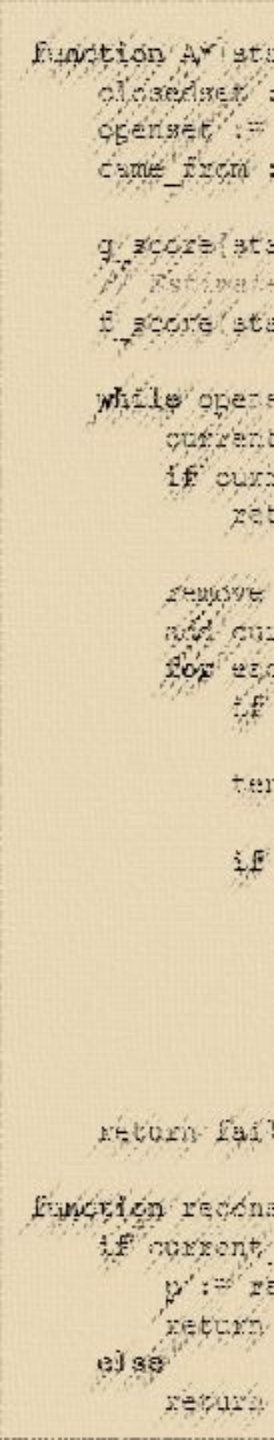
- Todos los diseñadores de algoritmos se deben preguntar en un momento dado: ¿Se pude hacer mejor? (a menos que ya se haya demostrado que tal o cual algoritmo es óptimo en el número de operaciones).
- Mejorar un algoritmo tiene muchas facetas:
  - Memoria requerida
  - Tiempo de ejecución
  - Código (elegante)
  - …



Búsqueda en un arreglo ordenado

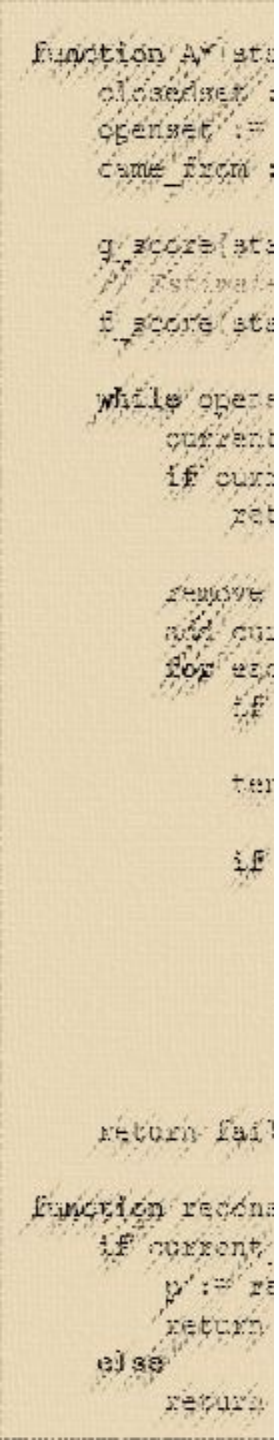
- Dado un el arreglo E que contiene n elementos en orden no decreciente (puede ser no creciente), y un valor K a ser buscado, encontrar un índice para el cual E[indice] = K, o bien, si K no está en el arreglo, devolver -1 como respuesta.
- La primera mejora proviene de que, dado que el arreglo está ordenado en forma no decreciente, en cuanto se llegue a un elemento mayor que K se sabrá que K no está en el arreglo.

NOTA: En la práctica, K suele ser la clave (*key*) de un elemento y los elementos pertenecen a alguna clase con muchos campos además de la clave, así que una mejor comparación sería: K = E[indice].clave



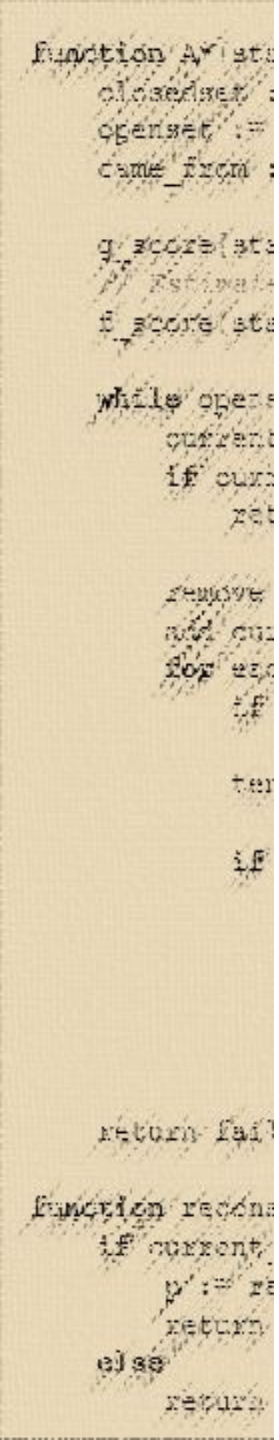
¿Podemos mejorarlo?

- Supongamos ahora que comparamos K con cada 4 elementos del arreglo.
  - Si hay coincidencia, habremos terminado.
  - Si K es mayor que el elemento con el que se comparó, digamos E[i], entonces no será necesario examinar los 3 elementos que preceden a E[i].
  - Si K < E[i], entonces K estará entre los dos últimos elementos que se comparó y unas cuantas comparaciones más (¿cuántas?) bastarán para determinar la posición de K, si está en el arreglo, o determinar que no está.
- ¿Cuántas comparaciones se hacen en el peor caso?



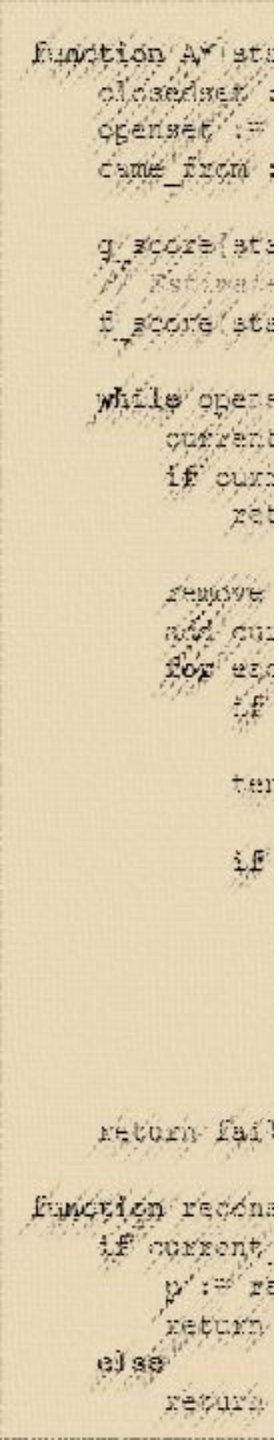
¿Podemos mejorarlo?

- Podríamos generalizar el procedimiento y comparar cada j elementos.
  - Esto requeriría alrededor de j/n comparaciones para seleccionar una sección del arreglo y alrededor de j comparaciones más para terminar.
- ¿Cuál será el mejor valor para j?



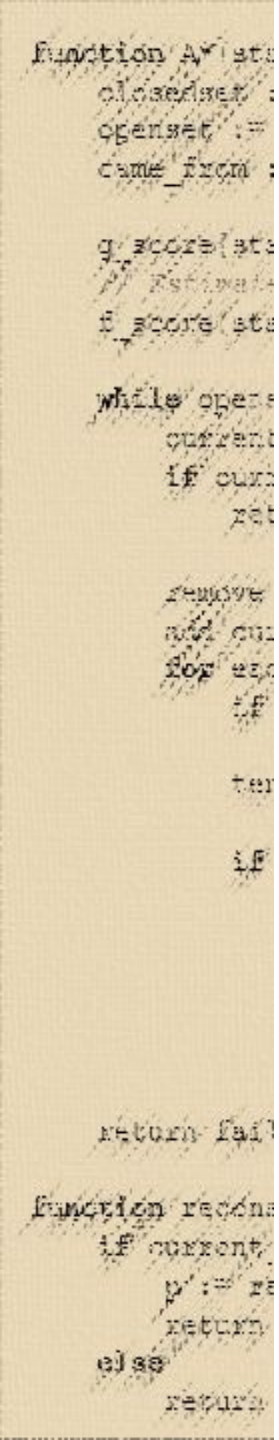
¿Podemos mejorarlo?

- Observemos que nuestra estrategia cambia una vez que encontramos la sección pequeña.
- Esta sección tiene aproximadamente j elementos y tenemos que hacer una búsqueda secuencial entre estos, pagando j comparaciones, lo cual es un costo lineal (depende de n).
- Pero ahora sabemos que un costo lineal es excesivo (en el caso anterior hicimos $2\sqrt{n}$ comparaciones).
- Esto sugiere que debemos aplicar nuestra "estrategia maestra" recursivamente para buscar ahora en la sección pequeña.



Búsqueda Binaria

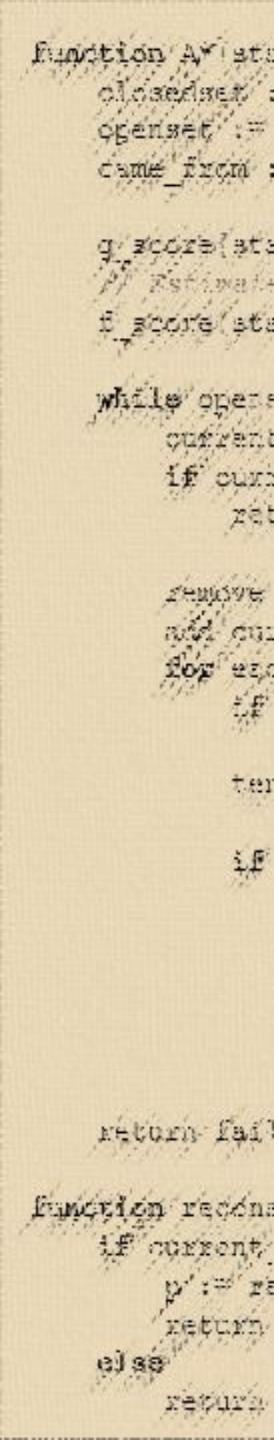
- La idea del tan conocido algoritmo de Búsqueda Binaria lleva aquello de “cada j-ésimo elemento” a su extremo lógico, saltándose la mitad de los elementos en un solo paso.
- En lugar de seleccionar un entero j y comparar K con cada j-ésimo elemento, comparamos K primero con el elemento que está a la mitad del arreglo.
- Esto elimina la mitad de las claves del arreglo.
- Una vez que hemos determinado cuál mitad podría contener a K, aplicamos la misma estrategia recursivamente.



Terminación

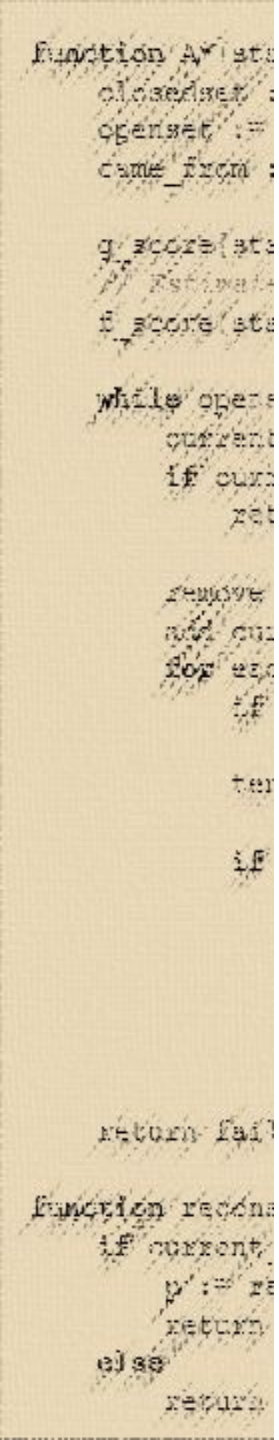
- Hasta que la sección que podría contener a K se haya encogido a tamaño cero, o se haya encontrado a K en el arreglo, seguiremos comparando K con el elemento que está a la mitad de la sección del arreglo que estamos considerando.

NOTA: Este procedimiento es un ejemplo sobresaliente del paradigma de diseño de algoritmos “Divide y Vencerás”.



¿Cuántas comparaciones realiza?

- Analizando el peor caso, como se acostumbra en los algoritmos, vemos lo siguiente:
  - ¿Cuántas veces podemos dividir n entre 2 sin obtener un resultado menor que 1?
  - En otras palabras, ¿qué valor máximo puede tener d sin que deje de cumplirse $n/2^d \geq 1$?
  - Por lo tanto, podemos efectuar floor(lg(n)) comparaciones después de invocaciones recursivas, y una comparación antes de cualquier invocación recursiva, es decir, un total de floor(ln(n)) + 1 = ceil(lg(n+1)).



Algoritmo: Búsqueda Binaria

Entradas: Un arreglo ordenado E en el intervalo *primero…ultimo*, y la clave que se busca K.

Salidas: la posición (indice) para la cual K = E[indice], si K está entre la posición *primero…ultimo* del arreglo o indice = -1 si no está.

Algoritmo …

```
int busquedaBinaria(int[] E, int primero, int ultimo, int K)
    if (ultimo < primero)
        indice = -1;
    else
        medio = (primero + ultimo) / 2; // división entera o floor
        if (K == E[medio])
            indice = medio;
        else if (K < E[medio])
            indice = busquedaBinaria(E, primero, medio – 1, K);
        else
            indice = busquedaBinaria(E, medio + 1, ultimo, K);
    return indice
```

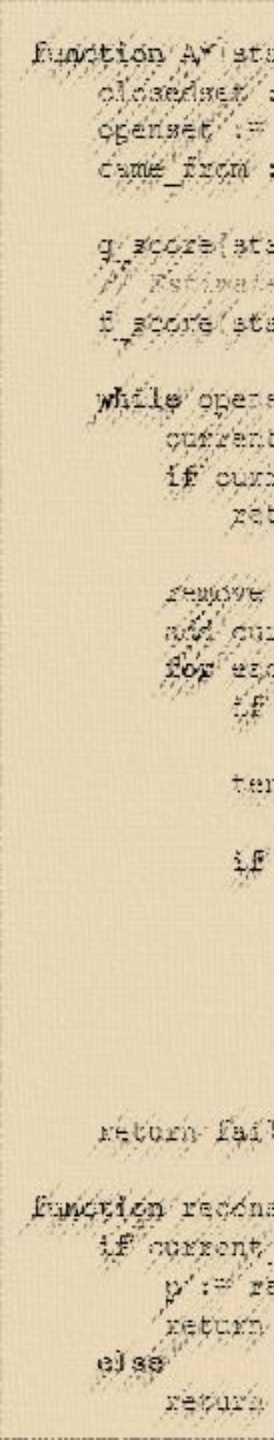
Ejemplo de escritorio

- Buscar K = 5 en el arreglo

  E = [-3, -1, 1, 3, 5, 7, 9, 11, 13]

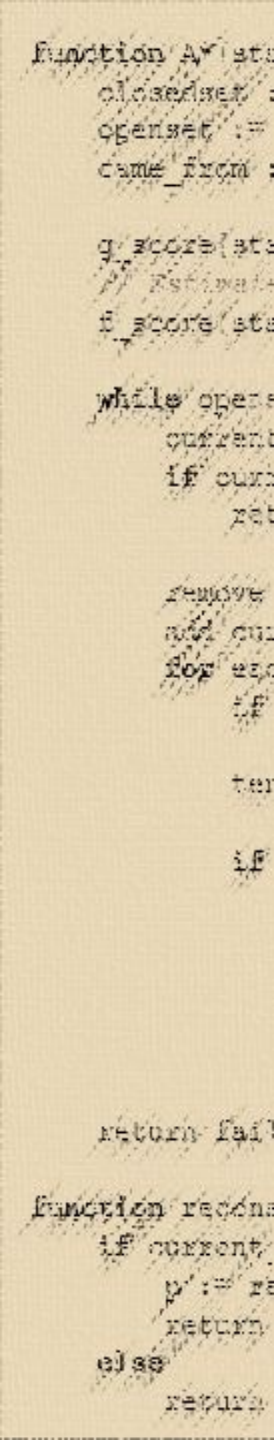
- Buscar K = 0, en el mismo arreglo
- Buscar K = 13 en el arreglo

  E = [-1, 1, 3, 5, 7, 9, 11, 13]



Proyecto 3

- Implementar búsqueda secuencial y búsqueda binaria para buscar en arreglos de enteros.
- Se debe implementar una clase Arreglo, la cual tiene:
  - Una variables de instancia que es el arreglo
  - Algunos de sus métodos son los algoritmos de búsqueda.



Referencias

[1] Elliot B. Koffman. Objects, abstraction, data structures, and design: using C++. John Wiley & Sons Inc. (2005).

[2] Silvia Guardati. Estructura de datos orientada a objetos: Algoritmos con C++. Pearson (2007).