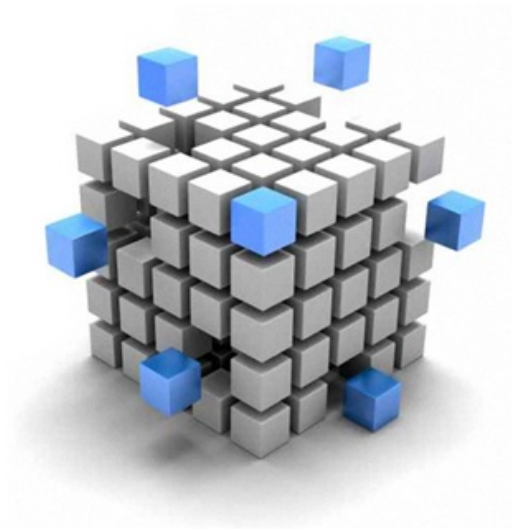


Búsqueda Binaria Recursiva

Grupo Alias



Análisis de Algoritmos
INF-648

<https://github.com/Lanceconan/AnalisisDeAlgoritmos>

Manuel Irrarrázaval
Juan Cid
Rafael Vivar
Daniel Gutiérrez

14 Mayo 2014

Índice

1	Algoritmo de búsqueda binaria recursiva	2
1.1	Repositorio GitHub	2
1.2	Definición y explicación del algoritmo	2
1.3	Explicación	4
1.4	Comparación con Búsqueda Lineal	5
1.5	Procedimientos	5
2	Complejidad teórica de la Búsqueda Binaria Recursiva	6
3	Sobre nuestro código programado en Ruby	8
3.1	Lenguaje de programación Ruby	8
3.2	Sobre el entorno de desarrollo	8
3.3	Código en Ruby	11
3.4	Pruebas al programa	13
3.4.1	Peor caso	13
3.4.2	Mejor caso	14
3.4.3	Ejecución del programa	14
4	Aplicaciones y Ventajas/Desventajas del uso de Algoritmo de Búsqueda Binaria Recursiva	15
4.1	Ventajas	15
4.2	Desventajas	15
4.3	Aplicaciones	15
5	Conclusiones	16
6	Anexos	17
6.1	Búsqueda binaria iterativa	17
6.1.1	Definición	17
6.1.2	Características (Ventajas/Desventajas)	17
6.1.3	En qué consiste el método	17
6.1.4	Algoritmo	18
6.1.5	Código	18
6.2	Árboles AVL	18

1 Algoritmo de búsqueda binaria recursiva

1.1 Repositorio GitHub

Se adjunta el siguiente repositorio de GitHub donde se puede encontrar en el directorio ‘presentación’ una PPT hecho en Latex con Beamer su respectivo Código en Latex editado en TexWorks.

En el directorio **Informe** se encontrará el código \LaTeX del informe y el mismo en su correspondiente formato PDF.

También se puede encontrar en el directorio ‘BusquedaBinaria’ el código en Ruby de la búsqueda Binaria Recursiva:

<https://github.com/Lanceconan/AnalisisDeAlgoritmos>

1.2 Definición y explicación del algoritmo

La búsqueda binaria recursiva en un vector ordenado de datos se realiza comprobando el elemento que está en el centro del vector y mirando si el elemento buscado es mayor o menor.

La primera iteración de este algoritmo evalúa el elemento medio del arreglo. Si éste coincide con la clave de búsqueda, el algoritmo termina. Suponiendo que el arreglo se ordene en forma ascendente, entonces si la clave de búsqueda es menor que el elemento de en medio, no puede coincidir con ningún elemento en la segunda mitad del arreglo, y el algoritmo continúa sólo con la primera mitad (es decir, el primer elemento hasta, pero sin incluir, el elemento de en medio). Si la clave de búsqueda es mayor que el elemento de en medio, no puede coincidir con ninguno de los elementos de la primera mitad del arreglo, y el algoritmo continúa sólo con la segunda mitad del arreglo (es decir, desde el elemento después del elemento de en medio, hasta el último elemento). Cada iteración evalúa el valor medio de la porción restante del arreglo. Si la clave de búsqueda no coincide con el elemento, el algoritmo elimina la mitad de los elementos restantes. Para terminar, el algoritmo encuentra un elemento que coincide con la clave de búsqueda o reduce el sub-arreglo hasta un tamaño de cero. Como ejemplo, se consideró el siguiente arreglo ordenado de 15 elementos:

2 3 5 10 27 30 34 51 65 77 81 82 93 99

y una clave de búsqueda de 65. Un programa que implemente el algoritmo de búsqueda binaria primero comprobaría si el 51 es la clave de búsqueda (ya que 51 es el elemento de en medio del arreglo). La clave de búsqueda (65) es mayor que 51, por lo que este número se descarta junto con la primera mitad del arreglo (todos los elementos menores que 51). A continuación, el algoritmo comprueba si 81 (el elemento de en medio del resto del arreglo) coincide con la clave de búsqueda. La clave de búsqueda (65) es menor que 81, por lo que se descarta este número junto con los elementos mayores de 81. Después de sólo dos pruebas, el algoritmo ha reducido el número de valores a comprobar a tres (56, 65 y 77). Después el algoritmo comprueba el 65 (que coincide indudablemente con la clave de búsqueda), y devuelve el índice del elemento del arreglo que contiene el 65. Este algoritmo sólo requirió tres comparaciones para determinar si la clave de búsqueda coincidió con un elemento del arreglo. Un algoritmo de búsqueda lineal hubiera requerido 10 comparaciones.

Un buen ejemplo de búsqueda binaria sería el siguiente:

```
// Fig. 16.4: ArregloBinario.java
// Clase que contiene un arreglo de enteros aleatorios y un método
// que utiliza la búsqueda binaria para encontrar un entero.
import java.util.Random;
import java.util.Arrays;

public class ArregloBinario
{
    private int[] datos; // arreglo de valores
    private static Random generador = new Random();

    // crea un arreglo de un tamaño dado y lo llena con enteros aleatorios
    public ArregloBinario( int tamaño )
    {
        datos = new int[ tamaño ]; // crea espacio para el arreglo

        // llena el arreglo con enteros aleatorios en el rango de 10 a 99
        for ( int i = 0; i < tamaño; i++ )
            datos[ i ] = 10 + generador.nextInt( 90 );

        Arrays.sort( datos );
    } // fin del constructor de ArregloBinario

    // realiza una búsqueda binaria en los datos
    public int busquedaBinaria( int elementoBusqueda )
    {
        int inferior = 0; // extremo inferior del área de búsqueda
        int superior = datos.length - 1; // extremo superior del área de búsqueda
        int medio = ( inferior + superior + 1 ) / 2; // elemento medio
        int ubicacion = -1; // devuelve el valor; -1 si no lo encontró

        do // ciclo para buscar un elemento
        {
            // imprime el resto de los elementos del arreglo
            System.out.print( elementosRestantes( inferior, superior ) );

            // imprime espacios para alineación
            for ( int i = 0; i < medio; i++ )
                System.out.print( " " );
            System.out.println( " * " ); // indica el elemento medio actual

            // si el elemento se encuentra en medio
            if ( elementoBusqueda == datos[ medio ] )
                ubicacion = medio; // la ubicación es el elemento medio actual

            // el elemento medio es demasiado alto
            else if ( elementoBusqueda < datos[ medio ] )
                superior = medio - 1; // elimina la mitad superior
        } while ( ubicacion == -1 );
    }
}
```

```

        else // el elemento medio es demasiado bajo
            inferior = medio + 1; // elimina la mitad inferior
            medio = ( inferior + superior + 1 ) / 2; // recalcula el elemento medio
    } while ( ( inferior <= superior ) && ( ubicacion == -1 ) );

    return ubicacion; // devuelve la ubicación de la clave de búsqueda
} // fin del método busquedaBinaria

// método para imprimir ciertos valores en el arreglo
public String elementosRestantes( int inferior, int superior )
{
    StringBuilder temporal = new StringBuilder();

    // imprime espacios para alineación
    for ( int i = 0; i < inferior; i++ )
        temporal.append( " " );

    // imprime los elementos que quedan en el arreglo
    for ( int i = inferior; i <= superior; i++ )
        temporal.append( datos[ i ] + " " );

    temporal.append( "\n" );
    return temporal.toString();
} // fin del método elementosRestantes

// método para imprimir los valores en el arreglo
public String toString()
{
    return elementosRestantes( 0, datos.length - 1 );
} // fin del método toString
// fin de la clase ArregloBinario

```

1.3 Explicación

Para este caso se declara la clase `ArregloBinario`. Esta clase es similar a `ArregloLineal`: tiene dos variables de instancia `private`, un constructor, un método de búsqueda (`busquedaBinaria`), un método `elementosRestantes` y un método `toString`. En las líneas 13 a 22 se declara el constructor. Una vez que se inicializa el arreglo con valores `int` aleatorios de 10 a 99 (líneas 18 y 19), en la línea 21 se hace una llamada al método `Arrays.sort` en el arreglo `datos`. El método `sort` es un método `static` de la clase `Arrays`, que ordena los elementos en un arreglo en orden ascendente de manera predeterminada; una versión sobrecargada de este método nos permite cambiar la forma de ordenar los datos. Recordemos que el algoritmo de búsqueda binaria sólo funciona en un arreglo ordenado.

En las líneas 25 a 56 se declara el método `busquedaBinaria`. La clave de búsqueda se pasa al parámetro `elementoBusqueda` (línea 25). En las líneas 27 a 29 se calcula el índice del extremo inferior, el índice del extremo superior y el índice medio de la porción del arreglo en la que el programa está buscando actualmente. Al principio del método, el extremo inferior es 0, el extremo superior es la longitud del arreglo menos 1, y `medio` es el promedio de estos dos valores. En la línea 30 se inicializa la `ubicacion` del elemento en -1; el valor que se devolverá si no se encuentra el elemento. En las líneas 32 a 53 se itera hasta

que inferior sea mayor que superior (esto ocurre cuando no se encuentra el elemento), o cuando ubicacion no sea igual a -1 (lo cual indica que se encontró la clave de búsqueda). En la línea 43 se evalúa si el valor en el elemento medio es igual a elementoBusqueda. Si esto es true, en la línea 44 se asigna medio a ubicacion. Después el ciclo termina y ubicacion se devuelve al método que hizo la llamada. Cada iteración del ciclo evalúa un solo valor (línea 43) y elimina la mitad del resto de los valores en el arreglo (línea 48 o 50). En las líneas 26 a 44 se itera hasta que el usuario escriba -1. Para cada uno de los otros números que escriba el usuario, el programa realiza una búsqueda binaria en los datos para determinar si coinciden con un elemento en el arreglo. La primera línea de salida de este programa es el arreglo de valores int, en orden ascendente. Cuando el usuario indica al programa que busque el número 23, el programa primero evalúa el elemento medio, que es 42 (según lo indicado por el símbolo *). La clave de búsqueda es menor que 42, por lo que el programa elimina la segunda mitad del arreglo y evalúa el elemento medio de la primera mitad. La clave de búsqueda es menor que 34, por lo que el programa elimina la segunda mitad del arreglo, dejando sólo tres elementos. Por último, el programa comprueba el 23 (que coincide con la clave de búsqueda) y devuelve el índice 1.

1.4 Comparación con Búsqueda Lineal

Método de Búsqueda	Lineal	Binario	Binario Recursivo
Complejidad	$O(n^2)$	$O(\log n)$	$O(\log n)$
Variables Creadas	1 Índice	3 Variables	1 Auxiliar

1.5 Procedimientos

Tamaño del Arreglo	Búsqueda Lineal	Búsqueda Binaria
1	1	1
10	10	4
5.000	5.000	11
100.000	100.000	18
1.000.000	1.000.000	21

2 Complejidad teórica de la Búsqueda Binaria Recursiva

A diferencia de la búsqueda binaria iterativa, su versión recursiva es más lenta con cada incremento de número de elementos, ya que existirán más llamadas a la función por resolver, con el consiguiente gasto de tiempo de guardar y restaurar parámetros.

Mejor caso: la búsqueda binaria coincide con el elemento buscado en el primer punto medio: sólo se necesitaría una comparación de elementos. Esto significa que sus tiempos de ejecución óptimos no dependen de la cantidad de datos: son constantes y por tanto proporcionales a 1, es decir, son de $O(1)$.

Peor caso: En el peor caso la búsqueda binaria recursiva el valor a encontrar no se encuentra en el arreglo tras haber realizado las llamadas correspondientes del algoritmo (dividir y comparar), requiriendo un nivel de orden $O(\log n)$.

Caso promedio: Se debe tomar en cuenta el comportamiento del algoritmo y su condición de término, obteniéndose la siguiente ecuación de recurrencia:

$$\begin{aligned}T(n) &= T(n/2) + C \\ T(1) &= 1\end{aligned}$$

Donde " $n/2$ " responde al proceso de comparar el elemento medio del arreglo con el valor a buscar y el $T(1) = 1$ resume el mejor caso: el dato es encontrado en centro de la estructura mediante una llamada.

Para hallar el nivel de complejidad promedio se hacen los siguientes pasos:

I. Reemplazar las n en la ecuación original con valores sucesivos:

$$\begin{aligned}T(n/2) &= T(n/4) + C \\ T(n/4) &= T(n/8) + C \\ T(n/8) &= T(n/16) + C \\ T(n/16) &= T(n/32) + C\end{aligned}$$

II. Generalizar en términos de n y k para eliminar la recurrencia:

$$\begin{aligned}T(n) &= T(n/2) + C \\ T(n) &= T(n/4) + C + C \\ T(n) &= T(n/8) + C + C + C \\ T(n) &= T(n/16) + C + C + C + C \\ T(n) &= T(n/32) + C + C + C + C + C\end{aligned}$$

Nótese que al realizar las sucesiones se llega dar con el siguiente patrón: que los denominadores de las fracciones obedecen a potencias de 2 ($2^1, 2^2, 2^3, 2^4$, etc...) y que las constantes " C " son directamente proporcionales a los valores arrojados en los exponentes. Por lo que al condensar todo en una fórmula se obtiene que:

$$T(n) = T(n/2^k) + kC$$

III. Igualar con la condición inicial los términos encerrados en $T(n)$

PREVIO: se debe eliminar la variable " k " con la condición de tope:

$$n/2^k = 1 \rightarrow n = 2^k$$

Se aplica propiedades de logaritmos para dejar la ecuación en términos de "n" :

$$\mathbf{log_2 n = k}$$

Resultado: $T(n) = T((n/2)^{log_2 n}) + log_2 n * C$

Usando la condición de tope $T(1) = 1$

$$T(n) = 1 + log_2 n * C$$

Con lo obtenido recientemente se llega a la conclusión de que para el caso promedio la Búsqueda Binaria Recursiva tiene complejidad de orden $\mathbf{O(log_2 n)}$.

3 Sobre nuestro código programado en Ruby

3.1 Lenguaje de programación Ruby

Ruby es un lenguaje con un balance cuidado. Su creador, Yukihiro "Matz" Matsumoto, mezcló partes de sus lenguajes favoritos (Perl, Smalltalk, Eiffel, Ada, y Lisp) para formar un nuevo lenguaje que incorporara tanto la programación funcional como la programación imperativa.

A menudo ha manifestado que está *"tratando de hacer que Ruby sea natural, no simple", de una forma que se asemeje a la vida real.*

Continuando sobre esto, agrega: Ruby es simple en apariencia, pero complejo por dentro, como el cuerpo humano.

Ruby tiene un conjunto de otras funcionalidades entre las que se encuentran las siguientes:

- Manejo de excepciones, como Java y Python, para facilitar el manejo de errores.
- Un verdadero mark-and-sweep garbage collector para todos los objetos de Ruby. No es necesario mantener contadores de referencias en bibliotecas externas. Como dice Matz, "Esto es mejor para tu salud".
- Escribir extensiones en C para Ruby es más fácil que hacer lo mismo para Perl o Python, con una API muy elegante para utilizar Ruby desde C. Esto incluye llamadas para embeber Ruby en otros programas, y así usarlo como lenguaje de scripting. También está disponible una interfaz SWIG.
- Puede cargar bibliotecas de extensión dinámicamente si lo permite el sistema operativo.
- Tiene manejo de hilos (threading) independiente del sistema operativo. De esta forma, tienes soporte multi-hilo en todas las plataformas en las que corre Ruby, sin importar si el sistema operativo lo soporta o no, ¡incluso en MS-DOS!
- Ruby es fácilmente portable: se desarrolla mayoritariamente en GNU/Linux, pero corre en varios tipos de UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP, DOS, BeOS, OS/2, etc.
- Al ser un lenguaje moderno, maneja cantidades numéricas estratosféricas.

3.2 Sobre el entorno de desarrollo

Sobre el IDE utilizado:

- Nombre del IDE: Aptana Studio 3
- Link de descarga del IDE: <http://aptana.com/products/studio3>
- Descripción del IDE: Aptana Studio 3 es una herramienta de desarrollo profesional de código abierto para la web abierta. Su principal función es desarrollar y probar la aplicación web completa utilizando un único entorno. Con soporte para las últimas especificaciones de tecnología del navegador, como HTML5, CSS3, JavaScript, Ruby, Rails, PHP y Python.



Sobre el lenguaje de programación utilizado:

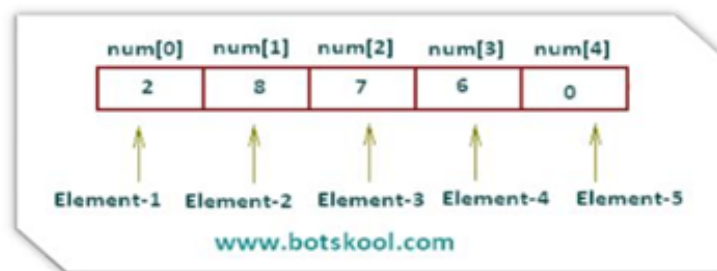
- Nombre del lenguaje: Ruby, versión 1.9.3
- Link de descarga: <https://www.ruby-lang.org/en/downloads/>

Sobre el código implementado para representar la búsqueda binaria recursiva, podemos resumir las siguientes características:

- Una única clase con sus inherentes métodos
- Métodos que permiten fácil manipulación y entendimiento del código
- Variables Nemotécnicas

Sobre el código en sí:

- Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N



- Permite distintos poblamientos de los datos:
 - Ordenado automático (de entrada sin usar ordenamiento burbuja)
 - Aleatorio
 - Manual

```
binaria.rb [Ruby Application] c:\Ruby193\bin\rubyw.EXE
=====
Busqueda Binaria Recursiva
Con Ordenamiento Burbuja
=====
Ingrese la opcion correspondiente
1.- Poblal vector inicial con Numeros Correlativos Ordenados y no repetitivos
2.- Poblal vector inicial Con Numeros aleatorios
3.- Poblal vector inicial Manualmente
X.- Cualquier otra opcion para salir

Ingrese Opcion:
```

- Fácil Manipulación del Código

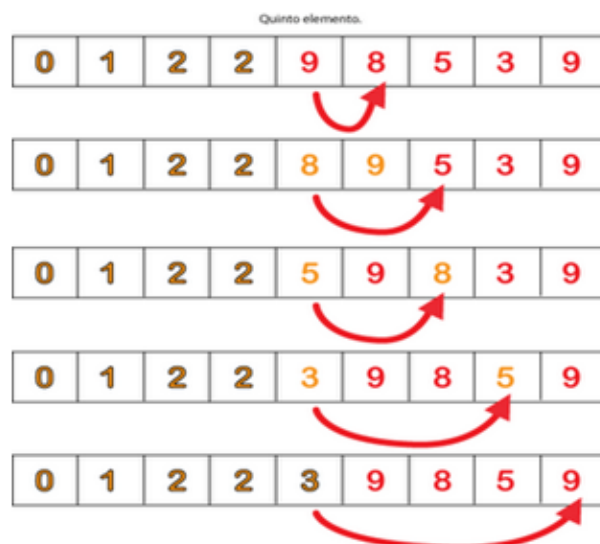
Al ser un programa escrito a modo de ejemplo se intentó hacer lo más simple de manipular posible, con tal de alguna vez aplicarlo en algún problema real. Se documentó muy en extenso el código y se indentó de tal forma que para cualquier programador, por inexperto que sea podrá manipular el código.

Se abusó del uso de métodos para que la comprensión y manipulación del código fuera la más fidedigna posible, resultando seguro, fiable, robusto, usable y portable.

- Método de Ordenamiento Burbuja (alternativo)

El algoritmo de ordenación por el método de la burbuja, también conocido como intercambio directo, es uno de los más simples que se conocen. Se basa en una serie de intercambios entre elementos adyacentes. Esos intercambios dan la impresión de que cada elemento va ascendiendo a través del array acercándose cada vez más a su posición final, recordando a cómo ascienden las burbujas de gas en un líquido.

A efectos prácticos, el algoritmo de la burbuja no es adecuado prácticamente para ninguna situación, ya que realiza muchas comparaciones y muchos intercambios. Pero para efectos prácticos solo se usa este algoritmo para demostrar la dependencia de datos ordenados en la búsqueda binaria recursiva. Hay algoritmos similares que se comportan bastante mejor pero su interés es más bien teórico, ya que sirve para establecer comparativas con otros métodos y extraer conclusiones teóricas.



3.3 Código en Ruby

```
#programa:  Busqueda Binaria Recursiva

require 'date'
require 'Time'

class Bbinario

  #Variables de la clase
  @@operaciones = 0
  @@tiemporFinalBusqueda = 0.00000000
  @@tiemporFinalOrdenamiento = 0.00000000
  #Llena el vector inicial manualmente
  def self.llenarManualmente(vector, num)
    i = 0
    loop do
      puts "Ingrese Numero #{i}"
      vector[i] = gets.chomp.to_i
      return vector if i == num
      i=i+1
    end
  end

  #Llenar aleatoriamente
  def self.llenarAleatorio(vector, num)

    i = 0
    loop do
      vector[i] = rand(999) + 1
      return vector if i == num
      i=i+1
    end
  end

  #Llenar en Orden no repetidos
  def self.llenarNoRepetidos(vector, num)

    i = 0
    loop do
      vector[i] = i
      i=i+1
      return vector if i == num
    end
  end

  # metodo de ordenamiento burbuja
  def self.ordenar(vector)
    inicio = Time.now
    for i in 0..vector.length-2
      for j in i+1...vector.length
        if vector[i] > vector[j]
          aux = vector[j]
          vector[j] = vector[i]
          vector[i] = aux
        end
      end
    end
    termino = Time.now
    @@tiemporFinalOrdenamiento = termino - inicio
    return vector
  end

  #Mostrar en pantalla el vector generado y ordenado
  def self.imprimir(vector)
    vector.each do |i|
      print "|#{i}"
    end
    print "|"
  end
end
```

```

# Se define la cantidad de entrada en el arreglo
def self.getCantidad
  print "Ingrese Cantidad de Datos en el arreglo inicial: "
  STDOUT.flush
  cant = gets.chomp.to_i
  return cant
end

# Se obtiene el valor a encontrar
def self.getBuscado
  print " extbackslashn extbackslashnIngrese el numero a buscar: "
  STDOUT.flush
  cant = gets.chomp.to_i
  return cant
end

# Algoritmo de busqueda binaria recursiva
def self.busquedaBinariaRecursiva(vector, val, left, right)

  if (left + right) % 2 == 0
    mid = (left + right) / 2
  else
    mid = (left + right) / 2 + 1
  end
  return -999 if left > right

  @@operaciones = @@operaciones + 1;

  if val == vector[mid]
    return mid
  elsif val > vector[mid]
    busquedaBinariaRecursiva(vector, val, mid + 1, right)
  else
    busquedaBinariaRecursiva(vector, val, left, mid - 1)
  end
end

#Funcion donde se defie como y donde aplicar el algoritmo, con las condiciones entregadas por parametro
def self.aplicarAlgoritmo(arreglo, cant)
  inicio = Time.now.to_f
  #ordenar(arreglo)
  termino = Time.now.to_f
  print " extbackslashn extbackslashnEl tiempo total empleado en el ordenamiento fue: "
  orden = termino - inicio
  print "%.7f Segundos" %(termino - inicio)

  #Para numeros grandes deshabilitar mostrar el vector ordenado
  puts " extbackslashnEl vector ingresado y ordenado fue"
  #imprimir(arreglo)

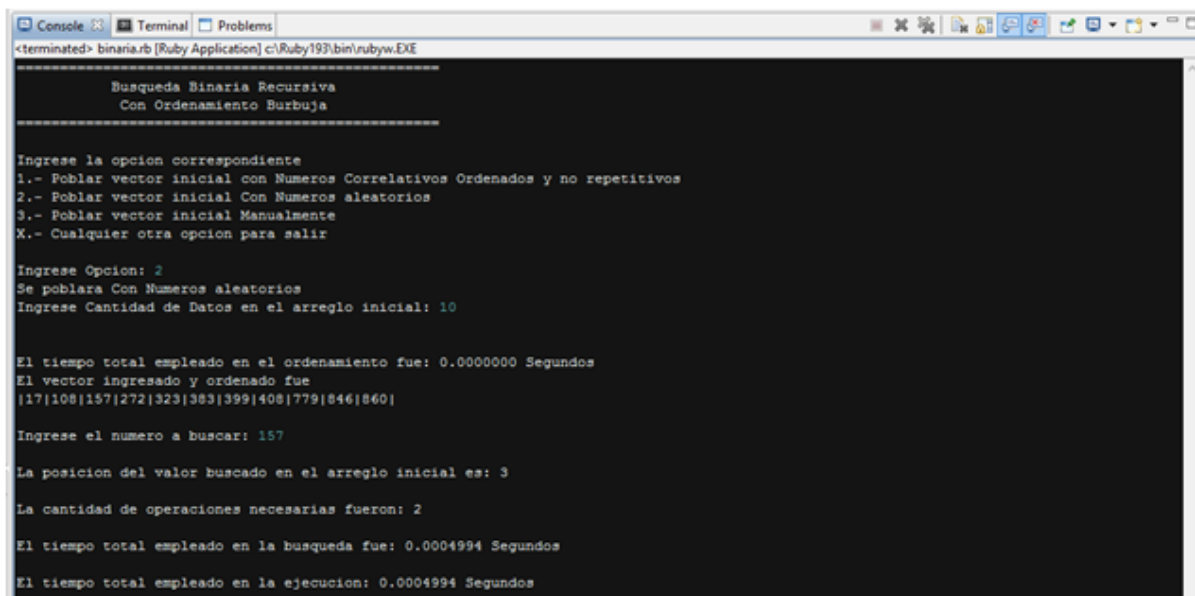
  buscado = getBuscado
  inicioU = Time.now.to_f
  sleep(0.1)
  nuevo = busquedaBinariaRecursiva(arreglo,buscado,0,cant-1)
  terminoU = Time.now.to_f
  busqueda = termino - inicio
  if nuevo == - 999
    puts "No se encontro el valor buscado, por lo tanto es el peor caso a evaluar"
    print " extbackslashn extbackslashnLa cantidad de operaciones necesarias fueron: "
    print @@operaciones
    print " extbackslashn extbackslashnEl tiempo de ejecucion total en la busqueda fue: "
    print "%.7f Segundos" %(terminoU - inicioU - 0.1)
    print " extbackslashn extbackslashnEl tiempo total empleado en la ejecucion: "
    print "%.7f Segundos" %((termino - inicio) + (terminoU - inicioU - 0.1))
  else
    print " extbackslashnLa posicion del valor buscado en el arreglo inicial es: "
    print nuevo + 1
    print " extbackslashn extbackslashnLa cantidad de operaciones necesarias fueron: "
    print @@operaciones
    print " extbackslashn extbackslashnEl tiempo total empleado en la busqueda fue: "

```


3.4.2 Mejor caso

Número de datos	Cantidad de operaciones
1.000	1
10.000	1
100.000	1
1.000.000	1
10.000.000	1
100.000.000	1

3.4.3 Ejecución del programa



```
<terminated> binaria.rb [Ruby Application] c:\Ruby193\bin\rubyw.EXE
=====
Busqueda Binaria Recursiva
Con Ordenamiento Burbuja
=====

Ingrese la opcion correspondiente
1.- Poblal vector inicial con Numeros Correlativos Ordenados y no repetitivos
2.- Poblal vector inicial Con Numeros aleatorios
3.- Poblal vector inicial Manualmente
X.- Cualquier otra opcion para salir

Ingrese Opcion: 2
Se poblara Con Numeros aleatorios
Ingrese Cantidad de Datos en el arreglo inicial: 10

El tiempo total empleado en el ordenamiento fue: 0.0000000 Segundos
El vector ingresado y ordenado fue
[17|108|157|272|323|383|399|408|779|846|860]

Ingrese el numero a buscar: 157

La posicion del valor buscado en el arreglo inicial es: 3

La cantidad de operaciones necesarias fueron: 2

El tiempo total empleado en la busqueda fue: 0.0004994 Segundos
El tiempo total empleado en la ejecucion: 0.0004994 Segundos
```

4 Aplicaciones y Ventajas/Desventajas del uso de Algoritmo de Búsqueda Binaria Recursiva

4.1 Ventajas

- Es un método eficiente siempre que el vector se encuentre ordenado.
- Proporciona un medio para reducir el tiempo requerido para buscar en una lista.
- Es más rápido debido a su recursividad, su mayor ventaja es con los archivos extensos.
- El código del procedimiento de esta búsqueda es corto en comparación con las demás técnicas de búsquedas.

4.2 Desventajas

- El archivo debe estar ordenado y el almacenamiento de un archivo suele plantear problemas en la inserción y eliminación de elementos.
- No revisa todos los elementos del archivo.
- Debe conocerse el número de elementos.
- Uso de memoria por recursividad.

4.3 Aplicaciones

- Partición Binaria del Espacio, usada en muchos videojuegos 3D para determinar que objeto necesita ser renderizado.
- Binary Tries, usada en casi todos los router de alta banda ancha para guardar las tablas de enrutamiento.
- Codificación Huffman, algoritmo usado para la compresión de datos, se utiliza en los formatos .jpeg y .mp3.

5 Conclusiones

La premisa que tenemos es que nosotros tenemos que buscar un elemento x en un vector v de tamaño N que está previamente ordenado. Teniendo eso en cuenta podemos afirmar que cualquier subvector w de v de tamaño $[0..N]$ también está ordenado. Teniendo eso en cuenta vamos reduciendo el tamaño del problema (N) a la mitad en cada llamada recursiva. ¿Por qué? Porque si x no es el elemento medio del vector v de tamaño N , entonces verificamos si es menor o mayor que él. Si es menor, buscamos en el subvector de tamaño $N/2$ izquierdo, sino en el derecho. Como se ve, en nuestro algoritmo se pasa de forma cíclica a forma recursiva casi sin pensar. ¿Por qué? Estamos ante una función recursiva y podemos pensar en definitiva que estamos ejecutando un ciclo simplemente. Decir simplemente que hay tener en cuenta que el vector en el que vayamos a buscar un elemento, debe estar previamente ordenado. Decir también que hay estructuras de datos eficientes para la búsqueda como lo son por ejemplo las Tablas Hash o los Árboles Binarios de Búsqueda, ABB (variante AVL), por ejemplo.

6 Anexos

6.1 Búsqueda binaria iterativa

6.1.1 Definición

La búsqueda binaria es el método más eficiente para encontrar elementos en un arreglo ordenado. El proceso comienza comparando el elemento central del arreglo con el valor buscado. Si ambos coinciden finaliza la búsqueda. Si no ocurre así, el elemento buscado será mayor o menor en sentido estricto que el central del arreglo. Si el elemento buscado es mayor se procede a hacer búsqueda binaria en la parte superior, si el elemento buscado es menor que el contenido de la casilla central, se debe cambiar el segmento a considerar al segmento que está a la izquierda de tal sitio central.

6.1.2 Características (Ventajas/Desventajas)

- Este método es muy eficiente siempre que el vector esté ordenado. En la práctica, esto suele suceder, pero no siempre. Por esta razón la búsqueda binaria iterativa exige una ordenación previa del archivo.
- La búsqueda binaria proporciona un medio para reducir el tiempo requerido para buscar en una lista.
- Es más rápido, su mayor ventaja es con los archivos extensos.
- El código del procedimiento de esta búsqueda es corto en comparación con las demás técnicas de búsqueda.
- En esencia, con una sola comparación eliminamos la mitad de la tabla; este es el método más eficiente de buscar en una lista ordenada sin emplear tablas o índices adicionales.
- El archivo debe estar ordenado y el almacenamiento de un archivo ordenado suele plantear problemas en las inserciones y eliminaciones de elementos.
- No revisa todos los elementos del archivo, requiere que todos los elementos estén ordenados.
- Mantener ese archivo ordenado es muy costoso.

6.1.3 En qué consiste el método

La búsqueda binaria iterativa utiliza un método de “divide y vencerás” para localizar el valor deseado. Con este método se examina primero el elemento central de la lista; si éste es el elemento buscado, entonces la búsqueda ha terminado. En caso contrario, se determina si el elemento buscado estará en la primera o la segunda mitad de la lista y a continuación se repite este proceso, utilizando el elemento central de esa sub-lista. Se puede aplicar tanto a datos en listas lineales como en árboles binarios de búsqueda. Los pre-requisitos principales para la búsqueda binaria son:

- La lista debe estar ordenada en un orden específico (no decreciente).
- Debe conocerse el número de registros.

6.1.4 Algoritmo

- El algoritmo compara el medio del espacio de búsqueda con el objetivo.
- Si el elemento analizado corresponde a la búsqueda; fin de búsqueda, si no vuelve a repetir el proceso.
- Si el elemento buscado es menor que la analizada repetir proceso en mitad superior, sino en la mitad inferior.
- El proceso partirá por la mitad el arreglo hasta encontrar el registro y dará la posición; en caso contrario nos retornará -1, lo cual implica que el valor del elemento buscado no está en la lista.

6.1.5 Código

```
int busqueda(int A[7], int tam, int n){

    int medio, inicio = 0, fin = tam - 1, encontro = -1;
    while((inicio <= fin) && (encontro == -1)){
        medio = (inicio + fin) / 2;
        if (A[medio] == n){
            encontro = medio;
        }
        else{
            if (A[medio] >= n){
                fin = medio - 1;
            }
            else{
                inicio = medio + 1;
            }
        }
    }
}
```

6.2 Árboles AVL

Un árbol AVL es un árbol binario de búsqueda que cumple con la condición de que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como mucho uno.

La denominación de árbol AVL viene dada por los creadores de tal estructura (Adelson-Velskii y Landis).

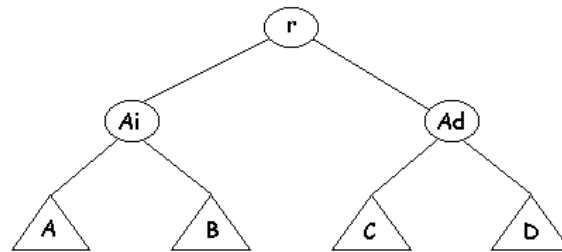
Recordamos que un árbol binario de búsqueda es un árbol binario en el cual cada nodo cumple con que todos los nodos de su subárbol izquierdo son menores que la raíz y todos los nodos del subárbol derecho son mayores que la raíz.

Recordamos también que el tiempo de las operaciones sobre un árbol binario de búsqueda son $O(\log n)$ promedio, pero el peor caso es $O(n)$, donde n es el número de elementos.

La propiedad de equilibrio que debe cumplir un árbol para ser AVL asegura que la profundidad del árbol sea $O(\log(n))$, por lo que las operaciones sobre estas estructuras

no deberán recorrer mucho para hallar el elemento deseado. Como se verá, el tiempo de ejecución de las operaciones sobre estos árboles es, a lo sumo $O(\log(n))$ en el peor caso, donde n es la cantidad de elementos del árbol.

Sin embargo, y como era de esperarse, esta misma propiedad de equilibrio de los árboles AVL implica una dificultad a la hora de insertar o eliminar elementos: estas operaciones pueden no conservar dicha propiedad.



Esquema general de árbol AVL.