Algoritmo de Busqueda Binaria Recursiva

Grupo Alias

Universidad Tecnológica Metropolitana

`https://github.com/Lanceconan/AnalisisDeAlgoritmos`

5 de mayo de 2014

Manuel Irarrázabal
Rafael Vivar
Daniel Gutiérrez
Juan Cid

Introducción

Definicion

La búsqueda binaria 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.
Consiste en buscar el elemento del medio (por ejemplo, si hubiera 13 elementos en la colección se comenzaría por el elemento en la 7ma posición; si hubiera 10 elementos se comienza por el que está en la 5ta posición) y comparar ese elemento por mayor o menor con respecto al elemento buscado. Si el buscado es mayor que el del medio, se descarta la mitad menor y se continúa buscando en la mitad mayor. Si es menor, se hace a la inversa. Si es igual, se ha encontrado el elemento buscado.

Propósito búsqueda binaria

Se utiliza cuando el vector en el que queremos determinar la existencia de un elemento está previamente ordenado.
Este algoritmo reduce el tiempo de búsqueda considerablemente, ya que disminuye exponencialmente el número de iteraciones necesarias.
Está altamente recomendado para buscar en arrays de gran tamaño. Por ejemplo, en uno conteniendo 50.000.000 elementos, realiza como máximo 26 comparaciones (en el peor de los casos).

Funcion Binaria Recursiva en C/C++

```
#include <iostream>
#include <vector>

bool busqueda_binaria(const vector<int> &v, int principio, int fin, int &x){
    bool res;
    if(principio <= fin){
        int m = (principio + fin)/2;
        if(x < v[m]) res = busqueda_binaria(v, principio, m-1, x);
        else if(x > v[m]) res = busqueda_binaria(v, m+1, fin, x);
        else res = true;
    }else res = false;
    return res;
}
/*{Post: Si se encuentra devuelve true, sino false}*/
```

Figura: Codigo

Calculo de complejidad

Dado que este algoritmo compara el valor a buscar con el elemento central del vector presenta la siguiente ecuación de recurrencia:

$$T(n) = T(n/2) + C \tag{1}$$

$$T(1) = 1 \tag{2}$$

Reemplazando las n en la ecuación original con valores sucesivos queda así:

$$T(n/2) = T(n/4) + C \tag{3}$$

$$T(n/2) = T(n/8) + C \tag{4}$$

$$T(n/2) = T(n/16) + C \tag{5}$$

$$T(n/2) = T(n/32) + C \tag{6}$$

Calculo de complejidad

Luego, se generaliza en términos de n y k para eliminar la recurrencia

$$T(n) = T(n/2) + C \tag{7}$$

$$T(n) = T(n/4) + C + C \tag{8}$$

$$T(n) = T(n/8) + C + C + C \tag{9}$$

$$T(n) = T(n/16) + C + C + C + C \tag{10}$$

$$T(n) = T(n/32) + C + C + C + C + C \tag{11}$$

Resultado:

$$T(n) = T(n/2^k) + kC \tag{12}$$

Lo siguente es igualar con la condición inicial los términos encerrados en T(n). Pero primero, se debe despejar la variable “k”

$$n/2^k = 1 ===> n = 2^k \tag{13}$$

Calculo de complejidad

Aplicando propiedades de logaritmos para despejar la variable “k”

$$log_2(n) = k \tag{14}$$

Finalmente, se procede a reemplazar los términos k con la condición inicial:

$$T(n) = T(n/2^{log_2(n)}) + log_2(n) * C \tag{15}$$

Como la componente

$$T(n/2^{log_2(n)}) \tag{16}$$

Calculo de Complejidad

quedó igualada con la condición

$$T(1) = 1 \tag{17}$$

se llega al siguiente resultado

$$T(n) = 1 + log_2(n) * C ===> T(n) = log_2(n) \tag{18}$$

Finalmente, se concluye que la búsqueda binaria recursiva tiene complejidad de orden

$$O(n) = log_2(n) \tag{19}$$

Mejor Caso

Como en todo algoritmo (dígase de búsqueda, ordenamiento, etc.) posee su nivel de complejidad y éste no es la excepción.
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 divide el arreglo,requiriendo sólo un tiempo O(log n).

Mejor Caso

Como en todo algoritmo (dígase de búsqueda, ordenamiento, etc.) posee su nivel de complejidad y éste no es la excepción.
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 divide el arreglo,requiriendo sólo un tiempo O(log n).

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Sobre el Código

- IDE utilizado: Aptana Studio 3
- Programación en Ruby, version 1.9.3
- Una única clase con sus inherentes métodos
  - Métodos que permiten facil manipulación y entendimiento del código
  - Variables Nemotécnicas
  - Búsqueda binaria recursiva
- Estructura utilizada: Un arreglo de largo N-1
- Metodo de Ordenamiento Burbuja (alternativo)

Caracteristicas

- Permite distintos poblamientos de los datos
  - Ordenado automatico (de entrada)
  - Aleatorio
  - Manual
- Facil Manipulacion del Codigo
- Utiliza el ordenamiento Burbuja Complejidad

$$O(n) = n^2 \tag{20}$$

Caracteristicas

- Permite distintos poblamientos de los datos
  - Ordenado automatico (de entrada)
  - Aleatorio
  - Manual
- Facil Manipulacion del Codigo
- Utiliza el ordenamiento Burbuja Complejidad

$$O(n) = n^2 \tag{20}$$

Caracteristicas

- Permite distintos poblamientos de los datos
  - Ordenado automatico (de entrada)
  - Aleatorio
  - Manual
- Facil Manipulacion del Codigo
- Utiliza el ordenamiento Burbuja Complejidad

$$O(n) = n^2 \tag{20}$$

Caracteristicas

- Permite distintos poblamientos de los datos
  - Ordenado automatico (de entrada)
  - Aleatorio
  - Manual
- Facil Manipulacion del Codigo
- Utiliza el ordenamiento Burbuja Complejidad

$$O(n) = n^2 \tag{20}$$

Caracteristicas

- Permite distintos poblamientos de los datos
  - Ordenado automatico (de entrada)
  - Aleatorio
  - Manual
- Facil Manipulacion del Codigo
- Utiliza el ordenamiento Burbuja Complejidad

$$O(n) = n^2 \tag{20}$$

Caracteristicas

- Permite distintos poblamientos de los datos
  - Ordenado automatico (de entrada)
  - Aleatorio
  - Manual
- Facil Manipulacion del Codigo
- Utiliza el ordenamiento Burbuja Complejidad

$$O(n) = n^2 \tag{20}$$

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

- Peor Caso: Que el valor no esté en el vector unidimensional
  - 1.000 : 10 llamados
  - 10.000 : 14 llamados
  - 100.000 : 17 llamados
  - 1.000.000 : 20 llamados
  - 10.000.000 : 24 llamados
  - 100.000.000 : 27 llamados
- Mejor Caso: Que el valor buscado está en el vector unidimensional justo en el centro estimado
  - 1.000 : 1 llamado
  - 10.000 : 1 llamado
  - 100.000 : 1 llamado
  - 1.000.000 : 1 llamado
  - 10.000.000 : 1 llamado
  - 100.000.000 : 1 llamado

Pruebas

Probemos el programa

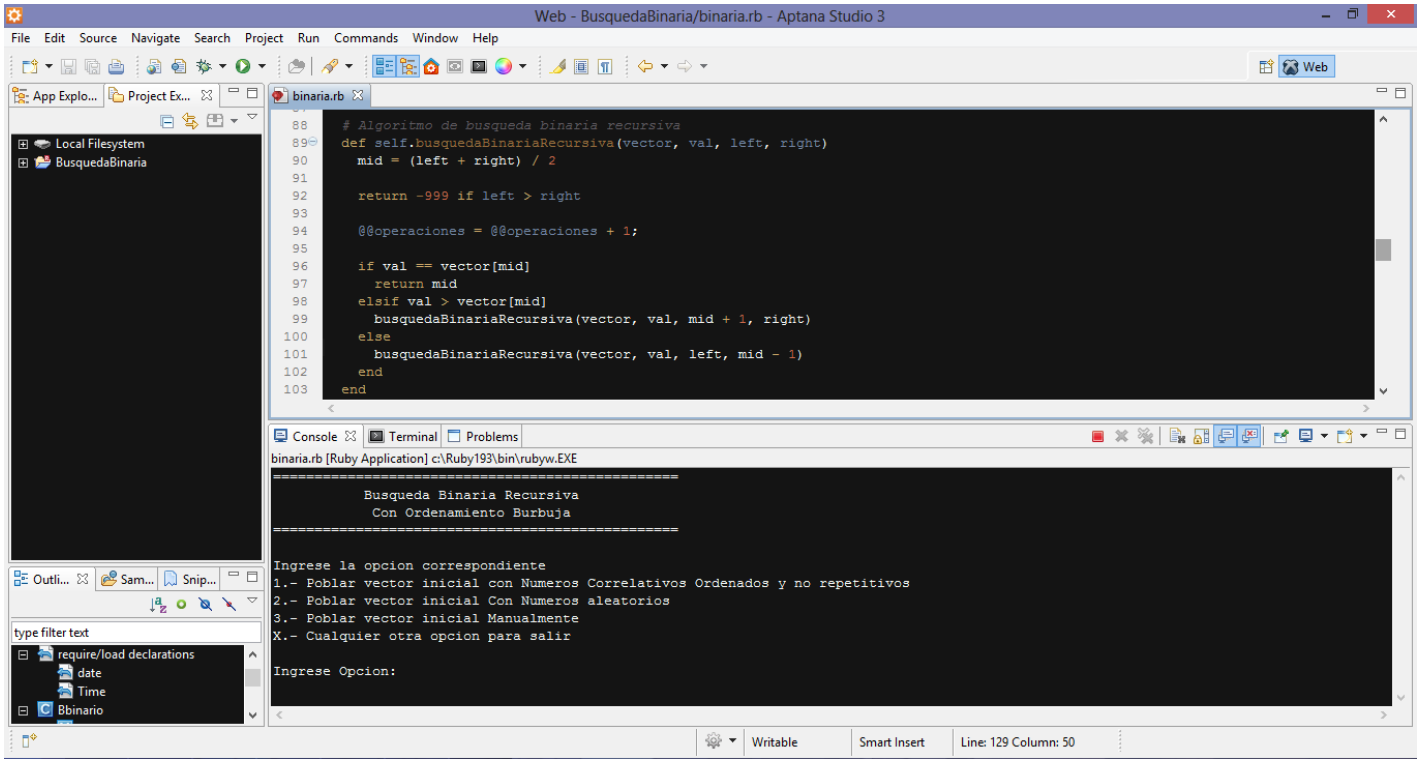

Figura: Pantallazo Aptana

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 está búsqueda es corto en comparación con las demás técnicas de búsquedas.

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.

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.

Conclusiones

- Se puede mejorar usando arboles autobalanceables, aunque estar asegurando minuciosamente puede añadir un costo considerable.
- No se nota la tendencia logaritmica de las inserciones debido a la complejidad lineal del recorrido.
- Otras Conclusiones

Conclusiones

- Se puede mejorar usando arboles autobalanceables, aunque estar asegurando minuciosamente puede añadir un costo considerable.
- No se nota la tendencia logaritmica de las inserciones debido a la complejidad lineal del recorrido.
- Otras Conclusiones

Conclusiones

- Se puede mejorar usando arboles autobalanceables, aunque estar asegurando minuciosamente puede añadir un costo considerable.
- No se nota la tendencia logaritmica de las inserciones debido a la complejidad lineal del recorrido.
- Otras Conclusiones

Conclusiones

Fin

¿Alguna Pregunta?