



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO



ANÁLISIS DE ALGORITMOS

PROFESOR: CRISTHIAN ALEJANDRO ÁVILA
SÁNCHEZ

PRÁCTICA:
ESTRATEGIA DIVIDE-AND-CONQUER

ALUMNO: GARAYOA FLORES ROBERTO
ALESSANDRO

GRUPO: 3CM11

1. Introducción

Randomized Binary Search Algorithm

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos. Funciona al dividir repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una. Usamos la búsqueda binaria en el juego de adivinar en la lección introductoria.

Una de las maneras más comunes de usar la búsqueda binaria es para encontrar un elemento en un arreglo. Por ejemplo, el catálogo estelar Tycho-2 contiene información acerca de las 2,539,913 estrellas más brillantes en nuestra galaxia. Supón que quieres buscar en el catálogo una estrella en particular, con base en el nombre de la estrella. Si el programa examinara cada estrella en el catálogo estelar en orden empezando con la primera, un algoritmo llamado búsqueda lineal, la computadora podría, en el peor de los casos, tener que examinar todas las 2,539,913 de estrellas para encontrar la estrella que estás buscando. Si el catálogo estuviera ordenado alfabéticamente por nombres de estrellas, la búsqueda binaria no tendría que examinar más de 22 estrellas, incluso en el peor de los casos.

Al describir un algoritmo para un ser humano, a menudo es suficiente una descripción incompleta. Algunos detalles pueden quedar fuera de una receta de un pastel; la receta supone que sabes cómo abrir el refrigerador para sacar los huevos y que sabes cómo romper los huevos. La gente puede saber intuitivamente cómo completar los detalles faltantes, pero los programas de computadora no. Es por eso que necesitamos describir completamente los algoritmos computacionales.

Para poder implementar un algoritmo en un lenguaje de programación, necesitarás entender un algoritmo hasta sus últimos detalles. ¿Cuáles son las entradas del problema? ¿Las salidas? ¿Qué variables deben crearse, y qué valores iniciales deben tener? ¿Qué pasos intermedios deben tomarse para calcular otros valores y calcular en última instancia la salida? ¿Estos pasos repiten instrucciones que se pueden escribir en forma simplificada al usar un bucle?

Veamos cómo describir cuidadosamente la búsqueda binaria. La idea principal de la búsqueda binaria es llevar un registro del rango actual de intentos razonables. Digamos que estoy pensando en un número entre uno y 100, justo como en el juego de adivinar. Si ya intentaste decir 25 y te dije que mi número es más grande, y ya intentaste decir 81 y te dije que mi número es más chico, entonces los números en el rango de 26 a 80 son los únicos intentos razonables. Aquí, la sección roja de la recta numérica contiene los intentos razonables, y la sección negra muestra los intentos que hemos descartado:



En cada turno, haces un intento que divide el conjunto de intentos razonables en dos rangos de aproximadamente el mismo tamaño. Si tu intento no es correcto, entonces te digo si es muy alto o muy bajo, y puedes eliminar aproximadamente la mitad de los intentos razonables. Por ejemplo, si el rango actual de los intentos razonables es de 26 a 80, intentarías adivinar a la mitad del camino, $(26+80)/2$, o 53. Si después te digo que 53 es demasiado alto, puedes eliminar todos los números de 53 a 80, dejando 26 a 52 como el nuevo rango de intentos razonables, reduciendo a la mitad el tamaño del rango.



2. Planteamiento del problema

Nos dan una matriz ordenada de $A[]$ de n elementos. Necesitamos encontrar si x está presente en A o no. En la búsqueda binaria siempre se usa el elemento medio, en este problema se deberá de elegir al azar un elemento en el rango dado.

3. Implementación

```
File Edit Selection View Go Run Terminal Help
binarySearch.cpp - Análisis de Algoritmos - Visual Studio Code

Prácticas > Divide-and-conquer > C++ binarySearch.cpp > main(void)
22 #include <iostream>
23 #include <ctime>
24 using namespace std;
25
26 //Función que generará un número aleatorio entre inicio y final y lo retornará
27 int getRandom(int inicio, int final){ //O(1)
28     srand(time(NULL)); //O(1)
29     return (inicio + rand() % (final - inicio + 1)); //O(1)
30 }
31
32 /*
33 Una función de un Randomize Binary Search
34 regresa la ubicación de "inicio" en un arreglo dado arr[lizq..der] si está presente el número, sino retorna un "-1"
35 */
36 int randomBinSearch(int arr[], int lizq, int der, int inicio){
37
38     if(der >= lizq){ //O(1)
39         //Aquí se define el "medio" como un índice random entre lizq y der
40         int medio = getRandom(lizq, der); //O(1)
41
42         //Si el elemento se encuentra presente en el medio entonces..
43         if(arr[medio] == inicio) //O(1)
44             return medio;
45
46         //Si el elemento es menor que el medio, entonces sólo puede estar presente en el subarreglo de la izquierda
47         if(arr[medio] > inicio) //O(1)
48             return randomBinSearch(arr, lizq, medio-1, inicio); //O(1)
49
50         //Si no sucede lo anterior, entonces el elemento se encuentra en el subarreglo de la derecha
51         return randomBinSearch(arr, medio+1, der, inicio); //O(1)
52     }
53
54     //En caso de que no se encuentra el elemento en medio del arreglo
55     return -1;
56 }
57
58 //Main
59 int main(void){
60     int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}; //O(1)
61     int number = sizeof(arr)/sizeof(arr[0]);
62     int inicio = 45;
63     int result = randomBinSearch(arr, 0, number-1, inicio);
64     (result == -1) ? printf("El elemento no está presente en el arreglo") : printf("El elemento está presente en el index %d", result);
65     return 0;
66 }
67
```

```

#include <iostream>
#include <ctime>
using namespace std;

//Función que generará un número aleatorio entre inicio y final y lo retornará
int getRandom(int inicio, int final){
    srand(time(NULL));
    return (inicio + rand() % (final - inicio + 1));
}

/*
Una función de un Radomize Binary Search
regresa la ubicación de "inicio" en un arreglo dado arr[izq..der] si está presente
el número, sino retorna un "-1"
*/
int randomBinSearch(int arr[], int izq, int der, int inicio){

    if(der >= izq){
        //Aquí se define el "medio" como un índice random entre izq y der
        int medio = getRandom(izq, der);

        //Si el elemento se encuentra presente en el medio entonces..
        if(arr[medio] == inicio)
            return medio;

        //Si el elemento es menor que el medio, entonces sólo puede estar presente
        en el subarreglo de la izquierda
        if(arr[medio] > inicio)
            return randomBinSearch(arr, izq, medio-1, inicio);

        //Si no sucede lo anterior, entonces el elemento se encuentra en el
        subarreglo de la derecha
        return randomBinSearch(arr, medio+1, der, inicio);
    }

    //En caso de que no se encuentra el elemento en medio del arreglo
    return -1;
}

//Main
int main(void){
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    int number = sizeof(arr)/sizeof(arr[0]);
    int inicio = 45;
    int result = randomBinSearch(arr, 0, number-1, inicio);
    (result == -1) ? printf("El elemento no esta presente en el arreglo") :
    printf("El elemento esta presente en el index %d", result);
}

```

```
    return 0;
}
```

4. Análisis de Complejidad

Comenzamos con el análisis de la complejidad con la primera línea del código:

```
//Función que generará un número aleatorio entre inicio y final y lo retornará
int getRandom(int inicio, int final){
    srand(time(NULL));
    return (inicio + rand() % (final - inicio + 1));
}
```

En la función *getRandom()* y *srand()* lo que esperamos es que retorne un número aleatorio que se haya generado entre el inicio y el final del arreglo. Su complejidad es:

$O(1)$ (en cada línea)

Pasando al primer condicional del código, este únicamente se encargará de crear y pasar la ubicación del arreglo que se creó de derecha a izquierda.

```
if(der >= izq){
```

Su complejidad:

$O(1)$

Iniciando con el caso base inicial tenemos:

```
//Si el elemento se encuentra presente en el medio entonces..
if(arr[medio] == inicio) //O(1)
    return medio;
```

Aquí el arreglo se colocará en medio del arreglo, y “arreglo” deberá regresar el “medio” del arreglo. Complejidad.

$O(1)$

Para este caso tenemos la siguiente lógica

```
//Si el elemento es menor que el medio, entonces sólo puede estar presente en el
subarreglo de la izquierda
    if(arr[medio] > inicio)                //O(1)
        return randomBinSearch(arr, izq, medio-1, inicio);
```

Ya que hayamos encontrado nuestra ubicación en el “medio” de nuestro arreglo, lo que tenemos que saber hacia dónde nos tenemos que mover, si el elemento es menor que el medio, nos tendremos que mover hacia la izquierda para encontrar el número. Para este caso tendremos una complejidad de:

$$T(n) = T(n - 1) + O(1)$$

Continuando con la 2da parte del condicional

```
//Si no sucede lo anterior, entonces el elemento se encuentra en el subarreglo de
la derecha
    return randomBinSearch(arr, medio+1, der, inicio);                //T(n-
1)+O(1)
```

Para esta línea de código podemos encontrar una complejidad de:

$$T(n) = T(n - 1) + O(1)$$

Ya para concluir con el análisis de complejidad.

```
int main(void){
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};                //O(1)
    int number = sizeof(arr)/sizeof(arr[0]);
    int inicio = 45;
    int result = randomBinSearch(arr, 0, number-1, inicio);
    (result == -1) ? printf("El elemento no esta presente en el arreglo") :
printf("El elemento esta presente en el index %d", result);
    return 0;
```

Podemos encontrar a casi todas las declaraciones de las variables como un $O(1)$.

Pero al final tenemos lo siguiente:

$$T(n) = p \cdot T(1) + p \cdot T(2) + \dots + p \cdot T(n) + 1$$

poniendo $p = 1/n$

$$T(n) = (T(1) + T(2) + \dots + T(n)) / n + 1$$

$$n \cdot T(n) = T(1) + T(2) + \dots + T(n) + n \dots \text{eq(1)}$$

Del mismo modo para $n-1$

$$(n-1)*T(n-1) = T(1) + T(2) + \dots + T(n-1) + n-1 \dots \text{ecuación(2)}$$

Reste eq(1) - eq(2)

$$n*T(n) - (n-1)*T(n-1) = T(n) + 1$$

$$(n-1)*T(n) - (n-1)*T(n-1) = 1$$

$$(n-1)*T(n) = (n-1)*T(n-1) + 1$$

$$T(n) = 1/(n-1) + T(n-1)$$

$$T(n) = 1/(n-1) + 1/(n-2) + T(n-2)$$

$$T(n) = 1/(n-1) + 1/(n-2) + 1/(n-3) + T(n-3)$$

Similarmente,

$$T(n) = 1 + 1/2 + 1/3 + \dots + 1/(n-1)$$

Por lo tanto, $T(n)$ es igual al $(n-1)$ número armónico,

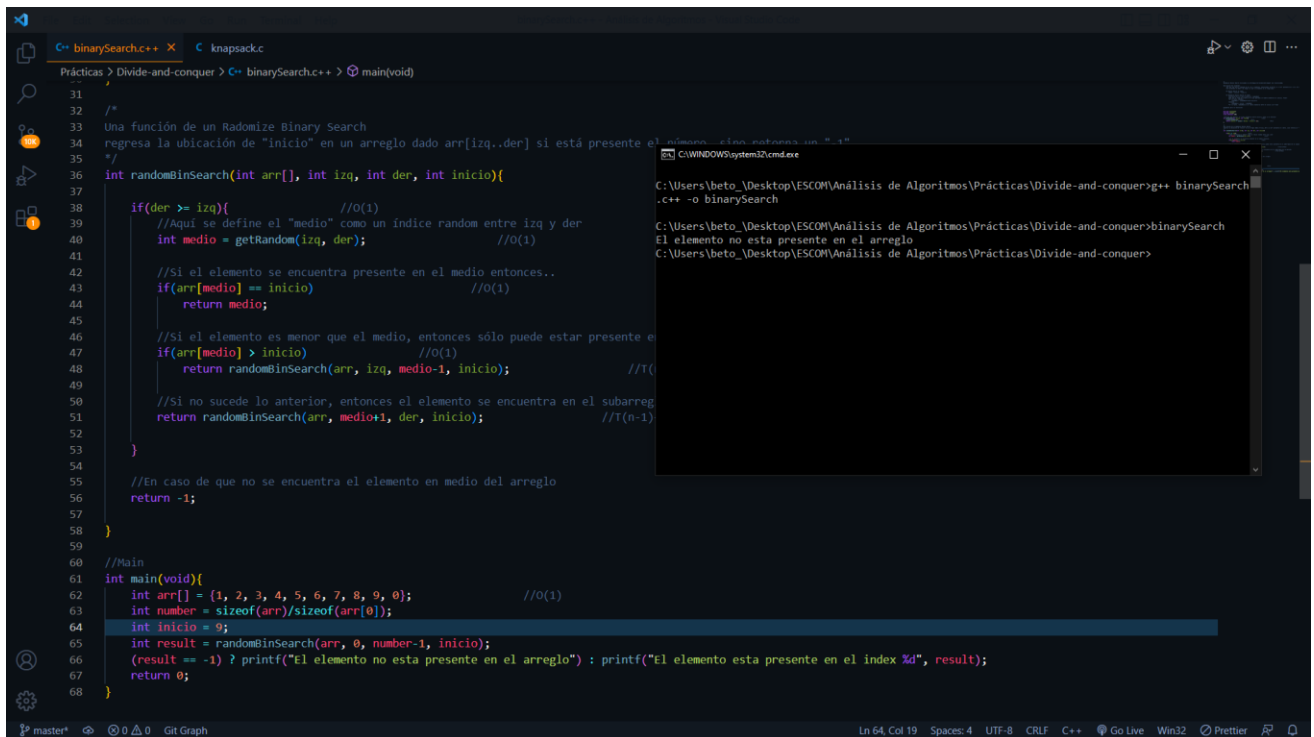
El n -ésimo número armónico es $O(\log n)$

Por lo tanto $T(n)$ es **$O(\log n)$**

5. Pruebas

1er prueba:

Como primera prueba tenemos que se le está solicitando que busque el número 9 al algoritmo, aunque el número se encuentra dentro del arreglo, este no está cerca del medio, por lo tanto nos retorna que *"El elemento no está presente en el arreglo"*.



```
31
32
33 /*
34  Una función de un Randomize Binary Search
35  regresa la ubicación de "inicio" en un arreglo dado arr[izq..der] si está presente el elemento "inicio" en el arreglo.
36  */
37 int randomBinSearch(int arr[], int izq, int der, int inicio){
38     if(der >= izq){ //O(1)
39         //Aquí se define el "medio" como un índice random entre izq y der
40         int medio = getRandom(izq, der); //O(1)
41
42         //Si el elemento se encuentra presente en el medio entonces..
43         if(arr[medio] == inicio) //O(1)
44             return medio;
45
46         //Si el elemento es menor que el medio, entonces sólo puede estar presente en el subarreglo [izq, medio-1]
47         if(arr[medio] > inicio) //O(1)
48             return randomBinSearch(arr, izq, medio-1, inicio); //T(n)
49
50         //Si no sucede lo anterior, entonces el elemento se encuentra en el subarreglo [medio+1, der]
51         return randomBinSearch(arr, medio+1, der, inicio); //T(n-1)
52     }
53 }
54
55 //En caso de que no se encuentra el elemento en medio del arreglo
56 return -1;
57
58 }
59
60 //Main
61 int main(void){
62     int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}; //O(1)
63     int number = sizeof(arr)/sizeof(arr[0]);
64     int inicio = 9;
65     int result = randomBinSearch(arr, 0, number-1, inicio);
66     (result == -1) ? printf("El elemento no esta presente en el arreglo") : printf("El elemento esta presente en el index %d", result);
67     return 0;
68 }
```

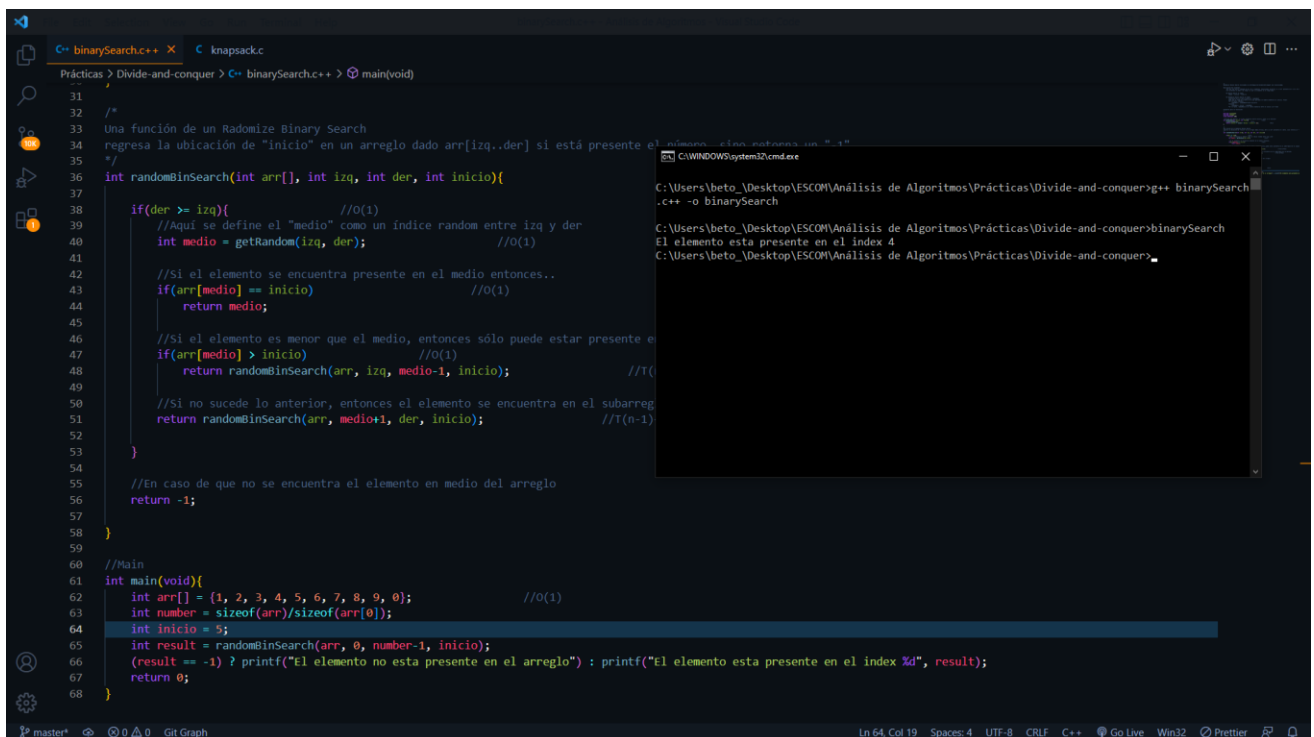
C:\Users\beto\Desktop\ESCOM\Análisis de Algoritmos\Prácticas\Divide-and-conquer>g++ binarySearch.cpp -o binarySearch

C:\Users\beto\Desktop\ESCOM\Análisis de Algoritmos\Prácticas\Divide-and-conquer>binarySearch

El elemento no esta presente en el arreglo

C:\Users\beto\Desktop\ESCOM\Análisis de Algoritmos\Prácticas\Divide-and-conquer>

En cambio, si pedimos que busque el número 5 en el arreglo, nos retornará lo siguiente.



```
31
32
33 /*
34  Una función de un Randomize Binary Search
35  regresa la ubicación de "inicio" en un arreglo dado arr[izq..der] si está presente el elemento "inicio" en el arreglo.
36  */
37 int randomBinSearch(int arr[], int izq, int der, int inicio){
38     if(der >= izq){ //O(1)
39         //Aquí se define el "medio" como un índice random entre izq y der
40         int medio = getRandom(izq, der); //O(1)
41
42         //Si el elemento se encuentra presente en el medio entonces..
43         if(arr[medio] == inicio) //O(1)
44             return medio;
45
46         //Si el elemento es menor que el medio, entonces sólo puede estar presente en el subarreglo [izq, medio-1]
47         if(arr[medio] > inicio) //O(1)
48             return randomBinSearch(arr, izq, medio-1, inicio); //T(n)
49
50         //Si no sucede lo anterior, entonces el elemento se encuentra en el subarreglo [medio+1, der]
51         return randomBinSearch(arr, medio+1, der, inicio); //T(n-1)
52     }
53 }
54
55 //En caso de que no se encuentra el elemento en medio del arreglo
56 return -1;
57
58 }
59
60 //Main
61 int main(void){
62     int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}; //O(1)
63     int number = sizeof(arr)/sizeof(arr[0]);
64     int inicio = 5;
65     int result = randomBinSearch(arr, 0, number-1, inicio);
66     (result == -1) ? printf("El elemento no esta presente en el arreglo") : printf("El elemento esta presente en el index %d", result);
67     return 0;
68 }
```

C:\Users\beto\Desktop\ESCOM\Análisis de Algoritmos\Prácticas\Divide-and-conquer>g++ binarySearch.cpp -o binarySearch

C:\Users\beto\Desktop\ESCOM\Análisis de Algoritmos\Prácticas\Divide-and-conquer>binarySearch

El elemento esta presente en el index 4

C:\Users\beto\Desktop\ESCOM\Análisis de Algoritmos\Prácticas\Divide-and-conquer>

El algoritmo nos indica que se encuentra en el índice 4, por lo que se entiende que sí se encuentra dentro del rango de "medio" como se había considerado en un inicio.

6. Conclusiones

El problema de búsqueda binaria aleatoria puede llegar a ser un poco confuso, pero cuando se aplica a otras situaciones, este puede ser muy útil. Se puede formar un árbol de búsqueda y el árbol puede generar N niveles de búsqueda hasta llegar al final.

7. Bibliografías

- ✓ (Dasgupta, July 18, 2006, p. 318)
- ✓ <https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search#:~:text=La%20b%C3%BAqueda%20binaria%20es%20un,ubicaciones%20posibles%20a%20solo%20una.>