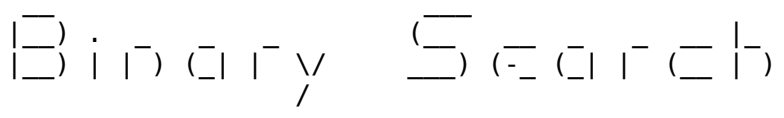
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬▬ 1. Introducción.

En el capítulo anterior se presentó el algoritmo de búsqueda lineal, o búsqueda secuencial conocido como “Linear Search”. En esa ocasión, nos dimos cuenta que aunque los números del vector se encuentran ordenados, no se utiliza esta cualidad de los datos para mejorar el proceso de búsqueda.

A continuación se presenta un nuevo proceso de búsqueda. En este caso si se utilizará el hecho que los elementos se encuentran ordenados para acelerar el proceso de búsqueda.

▬▬▬▬ 2. Búsqueda Binaria.

La búsqueda binaria, conocida como “binary search”, resuelve el siguiente problema. Dada una lista ordenada de “n” elementos, sin elementos repetidos, se

busca un elemento “x”. Si este elemento está presente, se devuelve la posición del subíndice donde se encuentra. De lo contrario se devuelve el valor de “false”.

Utiliza el siguiente algoritmo:

• Inicie con un intervalo que cubre todo el vector.

• Si el valor buscado está en la mitad del intervalo, devuelva el subíndice.

• Si el valor buscado es menor que la mitad del intervalo, reduzca el intervalo a la mitad inferior.

• En otro caso, reduzca el intervalo a la mitad superior.

• Repita el proceso hasta que se encuentre el valor, o el intervalo quede vacío.

▬▬▬ 3. Primer Ejemplo.

Deseamos responder:

```
(%i) bsearch(13,[])
```

Entonces:

```
index:  []
vector: []

ini: 1  ... Siempre iniciamos el vector en “1”.
fin: 0  ... Este es el largo del vector.

Como ini > fin

(%i) bsearch(13,[])
(%o) false
```

▬▬▬ 4. Segundo Ejemplo.

Deseamos responder:

```
(%i) bsearch(13,[12])


index:  [1]
vector: [12]



>>> Iteración 1.

ini: 1
fin: 1

m: (1+1)/2
m: 1
```

```
       i/m/f
index:  [1]
vector: [12]


Tenemos vector[1]#13,
        ini >= fin


(%i) bsearch(13,[12])
(%o) false
```

▬▬▬▬ 5. Tercer Ejemplo.

```
Deseamos responder:

(%i) bsearch(13,[13])


index:  [1]
vector: [13]



>>> Iteración 1.

ini: 1
fin: 1

m: (1+1)/2
m: 1


       i/m/f
index:  [1]
vector: [13]


Como vector[1]=13
Se retorna “1”


Entonces:
(%i) bsearch(13,[13])
(%o) 1
```

▬▬▬▬ 6. Ejemplo Más Complejo.

```
Tenemos un vector de 10 elementos ordenados de menor a mayor:

index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]
```

Deseamos saber:

(%i) bsearch(38,vector)

>>> Iteración 1.

ini:1

fin:10

m: (10 + 1)/2
m: 5.5 ~ 6

```
          i                    m                f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]
```

>>> Iteración 2.

Como 23 < 38

ini: 7

fin: 10

m: (7 + 10) / 2
m: 8.5 ~ 9

```
                                  i       m   f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]
```

>>> Iteración 3:

Recortamos el vector, pues 38 < 56.

ini: 7

fin: 8

m: (7 + 8) / 2
m: 7.5 ~ 8

```
                                  i  m/f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]
```

>>> Iteración 4:

Como 38 < 56

```
ini: 7

fin: 7

m: (7 + 7) / 2
m: 7


                                i/m/f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]



Por lo tanto:
(%i) bsearch(38,vector)
(%o) 7
```

▬▬▬ 7. Otro Ejemplo.

```
Tenemos:

index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]


Supongamos que deseamos saber:

(%i) bsearch(25,vector)



>>> Iteración 1:

ini:1

fin:10

m: (10 + 1)/2
m: 5.5 ~ 6


          i                    m               f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]




>>> Iteración 2:

Como 23 < 25

ini:7

fin:10

m: (7 + 10) / 2
m: 8.5 ~ 9
```

```
                                  i       m   f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]
```

>>> Iteración 3:

Como 25 < 72

ini: 7

fin: 8

m: (7 + 8) / 2
m: 7.5 ~ 8

```
                                  i  m/f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]
```

>>> Iteración 4.

Como 25 < 56

ini: 7

fin: 7

m: (7 + 7) / 2
m: 7

```
                                i/m/f
index:  [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
vector: [ 2,  5,  8, 12, 16, 23, 38, 56, 72, 91]
```

Como vector[7]#25
y ini >= fin
Se devuelve false

```
(%i) bsearch(25,vector)
(%o) false
```

▬▬▬▬ 8. Analizando la Complejidad.

La búsqueda binaria aprovecha que los números de encuentran ordenados. De esta manera en cada intento se reduce el conjunto a la mitad. Por eso de iniciamos con

10 elementos donde se debe buscar, de 10 se pasa a 5, de 5 a 3, de 3 a 2 y de 2 a 1. Por lo tanto, si tenemos la seguridad de terminar el problema en 4 intentos.

10 → 5 → 3 → 2 → 1

Si se tiene un vector con 100 elementos, tenemos la seguridad que encontraremos cualquier número que se escoja en a lo sumo 7 intentos. Puesto que en cada intento se quitan la mitad de los números.

100 → 50 → 25 → 13 → 7 → 4 → 2 → 1

Si se tiene un diccionario con 240,000 palabras. Y se busca la definición de una palabra. Cuánto es el máximo número de preguntas, o pasos, que se debe realizar para encontrar la palabra que buscamos?

240,000 → 120,000 → 60,000 → 30,000 → 15,000

→ 7,500 → 3,750 → 1,875 → 938 → 469

→ 235 → 118 → 59 → 30 → 15

→ 8 → 4 → 2 → 1

Se necesitan un máximo de 18 pasos.

●● Un repaso de logaritmos.

Los exponenciales y los logaritmos son funciones inversas. Por lo tanto:

$\log_{10}(100) = 2$, por lo tanto, $10^2 = 100$

$\log_{10}(1000) = 3$, por lo tanto, $10^3 = 1000$

$\log_2(8) = 3$, por lo tanto, $2^3 = 8$

$\log_2(16) = 4$, por lo tanto, $2^4 = 16$

$\log_2(32) = 5$, por lo tanto, $2^5 = 32$

●● Relación entre Búsqueda Binaria y Logaritmos.

En una búsqueda binaria de 8 elementos, se necesitan 3 pasos:

8 → 4 → 2 → 1

Que se puede calcular como:

$\log_2(8) = 3$

Veamos las iteraciones con más cuidado:

$8/2 = 4$

$4/2 = 2$

$2/2 = 1$

Es equivalente a:

$8/2^1 = 4$

$8/2^2 = 2$

$8/2^3 = 1$

En general se tiene que:

$n / 2^k = 1$

$n = 2^k$

$\log_2(n) = \log_2(2^k)$

$\log_2(n) = k*\log_2(2)$

$\log_2(n) = k*1$

$\log_2(n) = k$

$k = \log_2(n)$

●● Veamos si es correcto el valor de “k”.

Con un vector de 8 elementos:

$8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

Y tenemos que:

$\log_2(8) = 3$

Una búsqueda binaria de 10 elementos, se necesitan 4 pasos:

$10 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1$

Que se puede calcular como:

$\log_2(10) = 3.3219 \approx 4$

En una búsqueda binaria de 100 elementos, se necesitan 7 pasos:

100 → 50 → 25 → 13 → 7 → 4 → 2 → 1

Que se puede calcular como:

$\log_2(100) = 6.6438 \approx 7$

En general para una lista de tamaño “n” se necesitan los pasos dados por:

$\log_2(n) \approx$ número_de_pasos

●● Un comentario adicional.

La búsqueda binaria funciona porque la lista está ordenada. En estos ejemplos no se ha tomado en cuenta el tiempo que tarda, ordenar una lista, o bien mantenerla ordenada.

Una lista recibe constantemente números nuevos, o se borran algunos números. Estos procesos para mantener la lista ordenada todavía no los hemos estudiado.

▬▬▬ 9. “Running Time”.

Usualmente cuando hablamos de un algoritmo, nos interesa su tiempo de corrida, en inglés “Running Time”. Siempre se trata de escoger los algoritmo más eficientes, ya sea en tiempo o en espacio de memoria.

A continuación presentamos una comparación entre la búsqueda lineal y la búsqueda binaria:

Búsqueda Lineal	Búsqueda Binaria
100 elementos → 100 pasos	100 elementos → 7 pasos
4,000,000 elementos → 4,000,000 pasos	4,000,000 elementos → 32 pasos
“n” elementos → “n” pasos	“n” elementos → $\log_2(n)$ pasos
O(n)	O($\log_2(n)$)

▬▬▬ 10. Comparación de los Algoritmos.

Supongamos que se está desarrollando un algoritmo para calcular de forma automática el punto de aterrizaje de un cohete. El algoritmo debe funcionar correctamente y ser rápido. Se tienen 10 segundos para calcular el lugar del aterrizaje. Por un lado la búsqueda lineal es sencillo de programar, por lo hay menos posibilidad de que tenga errores, pero es más lento. La búsqueda binaria, es más difícil de programar, por lo que puede tener más errores, pero es más rápido. Comparemos los dos algoritmos.

Probemos ambos métodos con una lista de 100 elementos. Asumamos que toma 1 milisegundo (1 ms) para ver si un elemento de la lista es el que se está buscando. Como se muestra en la tabla anterior:

Con búsqueda lineal, para 100 elementos se necesitan 100ms.

Con búsqueda binaria, para 100 elementos se necesitan aproximadamente 7ms.

Por supuesto, que en la realidad, se necesitarán listas de mucho mayor tamaño. Veamos el tiempo que tarda cada algoritmo con diferentes listas:

Cantidad de Elementos	Búsqueda Lineal O(n)	Búsqueda Binaria $O(\log_2(n))$
100	100ms	7ms
10,000	10segs	14ms
10,000,000	2horas 46min	24ms
1,000,000,000	11 días	32ms

▬▬▬ 11. Resumen.

• La búsqueda binaria es mucho más rápida que la búsqueda secuencial.

• O(log n) es mucho más rápido que O(n), pero esto ocurre cuando el valor de “n” es lo suficientemente grande.

▬▬▬ 12. Ejercicios.

●● Ejercicio 1.

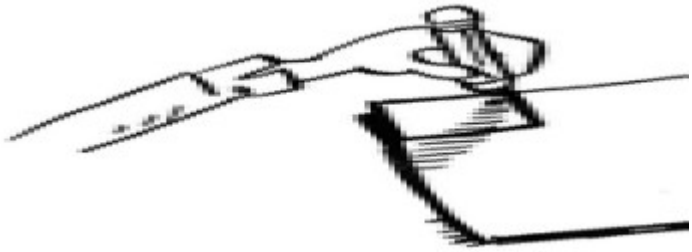
Si se tiene una lista de 128 nombres y se utiliza búsqueda binaria para buscar si aparece un nombre. Cuál es el máximo número de pasos que se necesitan para encontrarlo?

●● Ejercicio 2.

Supongamos que ahora se tiene un lista con el doble de elementos que en ejercicio anterior. Es decir, se tiene una lista con 256 elementos. Cuál es el máximo número de pasos que se necesitan para encontrarlo?

●● Ejercicio 3.

Se tiene una hoja de papel. En esta hoja se pueden dibujar 16 cuadrados del mismo tamaño. Se decide dibujar un cuadrado, y después otro y así sucesivamente. Si dibujar un cuadrado se considera una operación, en este caso se deben realizar 16 operaciones. Cuál es el tiempo de ejecución de este algoritmo?



●● Ejercicio 4.

Se desea usar este nuevo algoritmo para dibujar 16 cuadrados en una hoja de papel. Se dobla la hoja de papel en la mitad, se vuelve a doblar la hoja y así sucesivamente hasta tener 16 cuadrados. No es necesario de contabilizar el tiempo que se tarde desdoblando la hoja. Si es así, cuál es el tiempo de ejecución de este algoritmo?