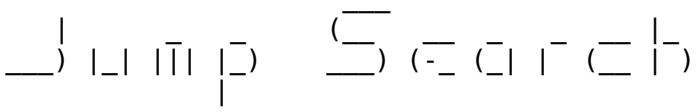
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬▬ 1. Introducción.

“Jump Search” o la búsqueda con brincos, desea buscar la posición en la que se encuentra un elemento en una lista. La lista no debe tener elementos repetidos y debe estar ordenada de menor a mayor.

La idea principal es chequear una menor cantidad de elementos que cuando se utiliza una búsqueda lineal. Para lograr esto se debe mover hacia adelante una cierta cantidad de pasos y así brincarse esos elementos.

▬▬▬▬ 2. Búsqueda por Brincos.

La búsqueda por brincos, 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. Si el elemento no existe se devuelve el valor de “false”.

Se utiliza el siguiente algoritmo:

Supongamos que tenemos un arreglo de tamaño “n” llamado vec[], y que vamos a utilizar un brinco de tamaño “m”. Entonces de deben revisar las posiciones del arreglo:

vec[m], vec[2*m], vec[3*m], vec[4*m] y así sucesivamente.

Una vez que se encuentra un subintervalo donde se localiza el valor que buscamos, cambiamos los índices de búsqueda y se realiza otra revisión.

Por ejemplo, si el elemento “x” que buscamos se encuentra entre:

vec[k*m] < x < vec[(k+1)*m]

Entonces se realiza una búsqueda secuencial en ese intervalo.

▬▬▬▬ 3. Primer ejemplo.

Suponga que tenemos el siguiente vector, con elementos ordenados de menor a mayor:

```
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,  13,  14,  15,  16]
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

>>> Iniciamos la búsqueda.

Supongamos que deseamos saber si x=89 está en la lista.

Iniciamos con un brinco de tamaño 4.

i: 4

```
                     ▼                   ▼                   ▼
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,  13,  14,  15,  16]
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Como vec[4]= 6 < 55 seguimos.

Como vec[8]=13 < 55 seguimos.

Como vec[12]=89 = 89 nos detenemos.

▬▬▬▬ 4. Segundo Ejemplo.

Suponga que tenemos el siguiente vector, con elementos ordenados de menor a mayor:

```
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,  13,  14,  15,  16]
```

```
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

>>> Proceso de Brinco.

Supongamos que deseamos saber si x=55 está en la lista.

Iniciamos con:

i: 4

```
                      ▼                   ▼                   ▼
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,  13,  14,  15,  16]
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Como vec[4]=6 < 55 seguimos.

Como vec[8]=13 < 55 seguimos.

Como vec[12]=89 > 55 nos detenemos

y hacemos una búsqueda secuencial.

>>> Proceso de Búsqueda Secuencial.

Se realiza una búsqueda lineal entre vec[9] y vec[11]

```
                      ▼                   ▼   v   v   v   ▼
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,  13,  14,  15,  16]
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Si está el elemento, devolvemos el subíndice.
Si el elemento no está, devolvemos false.

Por lo tanto devolvemos el valor del súbindice 11.

▬▬▬ 5. Tercer Ejemplo.

Supongamos que deseamos saber si x=34 está en la lista.
Utilizamos un brinco de tamaño 3.

```
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55]
```

>>> Iniciamos con:

i: 3

```
                  ▼           ▼           ▼
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55]
```

Como vec[3]=5 < 34 seguimos.

Como vec[6]=8 < 34 seguimos.

Como vec[9]=21 < 34 seguimos.

Ya no podemos hacer otro salto pues, 9 + 3 = 12
Y el vector solamente tiene 11 elementos.

>>> Búsqueda Secuencial.

i: 4*3 = 12

Como 12 es mayor que el largo del vector, usamos i:11

```
                  ▼           ▼           ▼   v
index: [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]
vector:[ 0,  1,  5,  6,  7,  8,  9, 13, 21, 34, 55]
```

Como vec[10]=34 Se detiene el proceso.

▬▬▬ 6. Análisis del Algoritmo.

Observemos que en un vector de largo 16 se realizaron:

16/4 = 4 brincos

De forma general:

(n/m) brincos.

En el peor caso, suponemos que el elemento se encuentra en el último intervalo.
En el vector de 16 elementos el último intervalo está entre:
vec[12] y vec[16]

Se deben revisar:
vec[13], vec[14], vec[15]

que son 3 comparaciones.

De forma general:

(m - 1) comparaciones.

Así que el tiempo total que toma el algoritmo está dado por:

$$\left(\frac{n}{m}\right) + \left(\ m - 1\ \right)$$

●● Buscamos el valor de “m”.

Qué valor de “m” debemos utilizar para que este valor se vuelva lo más pequeño posible?

Si vemos esto como una función de “m” de la forma:

f(m) := (n/m) + (m-1);

$$f(m) := \left(\frac{n}{m}\right) + \left(\ m - 1\ \right)$$

Al derivar f(m) se obtiene:

$$f(m) = \left(\frac{n}{m}\right) + \left(\ m - 1\ \right)$$

$$f'(m) = \left(\frac{n}{m}\right)' + \left(\ m - 1\ \right)'$$

$$f'(m) = (n * m^{-1})' + (m - 1)'$$

$$f'(m) = (n * m^{-1})' + (m') - (1')$$

$$f'(m) = (-n * m^{-2}) + (1)$$

$$f'(m) = \left(\frac{-n}{m^2}\right) + \left(1\right)$$

Para encontrar el mínimo igualamos a cero:

$$f'(m) = 0$$

$$\left(\frac{-n}{m^2}\right) + \left(1\right) = 0$$

$$\left(\frac{-n}{m^2}\right) = \left(-1\right)$$

$$\frac{-n}{m^2} = -1$$

$$\frac{n}{m^2} = 1$$

$$n = m^2$$

$$\sqrt{(n)} = m$$

$$m = \sqrt{(n)}$$

Podemos estar seguros que se trata de un mínimo, pues la segunda derivada es positiva:

$$f''(m) = \left(f'(m)\right)'$$

$$f''(m) = \left(\frac{-n}{m^2} + 1\right)'$$

$$f''(m) = (-n*m^{-2} + 1)'$$

$f''(m) = (-n*m^{-2})' + (1)'$

$f''(m) = (-n * -2 * m^{-3}) + (0)$

$f''(m) = -n * -2 * m^{-3}$

$f''(m) = 2*n*m^{-3}$

$$f''(m) := \left(\frac{2*n}{m^3}\right)$$

Por lo tanto:

$$f''(m) := \left(\frac{2*n}{m^3}\right)$$

f”(m) := (2*n)/m^3;

Como “n” y “m” son valores positivos, entonces f”(m) > 0.

●● El Tiempo de Ejecución del Algoritmo.

$$f(m) = \left(\frac{n}{m}\right) + \left(m - 1\right)$$

y sustituimos $m = \sqrt{(n)}$

$$f(n) = \left(\frac{n}{\sqrt{(n)}}\right) + \left(\sqrt{(n)} - 1\right)$$

f(n) = √(n) + √(n) - 1

f(n) = 2*√(n) - 1

f(n) = O(√(n))

●● Sobre el valor de “m”.

Tal como se ha indicado:

m = √(n)

En el caso que “m” sea un valor con decimales se ajustará hacia abajo.

>>> Por ejemplo: n = 10

m = √(10) = 3.1622 = 3

m = √(10) = 3.1622 = 3

>>> Por ejemplo: n = 15

m = √(15) = 3.8729 = 3

m = √(15) = 3.8729 = 3

>>> Por ejemplo: n = 16

m = √(16) = 4

>>> Por ejemplo: n = 21

m = √(21) = 4.5825 = 4

m = √(21) = 4.5825 = 4

▬▬▬ 7. Un Comentario Adicional.

En orden de eficiencia, del más lento al más rápido tenemos

• “Linear Search”, con tiempo O(n)

• “Jump Search”, con tiempo O(√(n))

• “Binary Search”, con tiempo $O(\log_2(n))$

» Gráfico

» Funciones de Search.

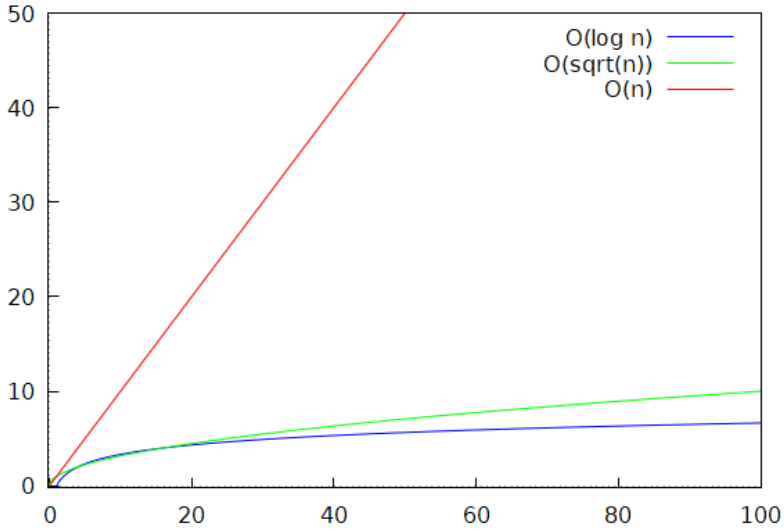
```
load(draw)$
fn(n):=n;
fsqrt(n):=sqrt(n);
flog2(n):=log(n)/log(2);


ini:0;
fin:100;
wxdraw2d
(
   xrange = [-0.20,100],
   yrange = [-0.20, 50],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(n)",
   explicit( fn(n), n, ini, fin),

   color=green,
   key="O(sqrt(n))",
   explicit( fsqrt(n), n, ini, fin),

   color=blue,
   key="O(log2 n)",
   explicit( flog2(n), n, ini, fin)
)$
```



▬▬▬ 8. Resumen.

• “Jump Search” funciona únicamente si el vector está ordenado.

• El tamaño óptimo del bloque para el “Jump Search” es de m = √(n). Al hacer eso se tiene que el tiempo de ejecución sea de O(√(n))

• El tiempo de ejecución del “Jump Search” es mejor que el “Linear Search”, pero no es tan bueno como el “Binary Search”.

• El “Binary Search” es mejor que el “Jump Search”, pero este último tiene la ventaja que se va hacia atrás en una única ocasión.

▬▬▬ 9. Ejercicios.

●● Ejercicio 1.

Si se utiliza el “jump search” con un valor de m=1. Cuál es el valor de O(?).

●● Ejercicio 2.

Implemente el algoritmo de “jump search” utilizando maxima.

●● Ejercicio 3.

Implemente el algoritmo de “jump search” utilizando otro lenguaje de su preferencia, Python, C, C++, etc.

●● Ejercicio 4.

Se obtiene un mejor resultado con el “jump search”, si una vez que se localiza el bloque donde se encuentra el elemento, se hace una búsqueda lineal inversa? Es decir, desde el subíndice mayor hacia el subíndice menor?