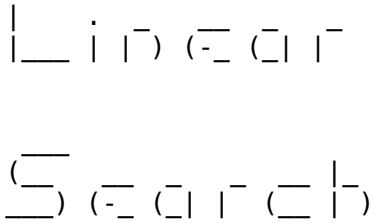
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬ 1. Introducción.

El algoritmo de “linear search” o búsqueda lineal es uno de los algoritmo más básicos. Lo usaremos como un primer caso para analizar distintas formas de implementación, formas iterativos y recursivos. Además servirá como elemento de partida para problemas más complejos.

▬▬▬ 2. Algoritmo de Búsqueda Lineal.

Suponga que se tiene un vector con 10 números ordenadas de menor a mayor. El vector inicia en la posición 1.

```
vector: [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99]
 index: [  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
```

Se desea construir un programa que recibe un número.

Si el número se encuentra dentro del vector, devuelve la posición en que se encuentra.

Si el número no se encuentra dentro del vector, devuelve “false”.

Una forma muy ineficiente, y que no utiliza el hecho que la lista está ordenada, es ir desde el inicio hacia el final buscando el número. Por ejemplo, si deseamos encontrar la posición del número 45 se realizaría el siguiente proceso:

Posición 1, valor 10. Continuar

Posición 2, valor 21. Continuar.

Posición 3, valor 34. Continuar.

Posición 4, valor 45. Return(4)

Si el número seleccionado estuviera en la posición 10, se necesitarían 10 intentos para encontrarlo. En cada intento solamente se reduce el conjunto de número en 1 número. Por ejemplo, si buscamos la posición del número 99.

Posición 1, valor 10. Continuar

Posición 2, valor 21. Continuar.

Posición 3, valor 34. Continuar.

Posición 4, valor 45. Continuar.

Posición 5, valor 50. Continuar.

Posición 6, valor 62. Continuar.

Posición 7, valor 73. Continuar.

Posición 8, valor 79. Continuar.

Posición 9, valor 91. Continuar.

Posición 10, valor 99. Return(10).

En el peor caso posible, el número se encuentra en la posición 10. Por lo que se necesitarían 10 preguntas para encontrarlo.

Si se tienen 100 elementos, en el peor caso posible se necesitarían 100 intentos de búsqueda.

Si se tienen “n” elementos, se necesitarían “n” intentos para encontrar el número.

n = número_de_iteraciones

De esta manera, este proceso de búsqueda es O(n).

▬▬▬ 3. Búsqueda Lineal En Recursión de Pila.

●● Cómo debe funcionar el programa?

En este caso tenemos:

```
lsearch(45, []);
--  lsearchAux(45, [])
--  1
false pues 1 > length([])
```

Supongamos que invocamos al programa con:

```
lsearch(45, [10, 21, 34, 45, 50, 62, 73, 79, 91, 99]);
lsearchAux(45, [10, 21, 34, 45, 50, 62, 73, 79, 91, 99]);
lsearchAux(45, [10, 21, 34, 45, 50, 62, 73, 79, 91, 99]);
1 + lsearchAux(45, [21, 34, 45, 50, 62, 73, 79, 91, 99]);
1 + 1 + lsearchAux(45, [34, 45, 50, 62, 73, 79, 91, 99]);
1 + 1 + 1 + lsearchAux(45, [45, 50, 62, 73, 79, 91, 99]);
1 + 1 + 1 + 1
1 + 1 + 2
1 + 3
4
```

Otro ejemplo:

```
lsearch(45, [10, 21, 34]);
lsearchAux(45, [10, 21, 34]);
1 + lsearchAux(45, [21, 34]);
1 + 1 + lsearchAux(45, [34]);
1 + 1 + 1 + lsearchAux(45, []);
1 + 1 + 1 + 1
1 + 1 + 2
1 + 3
```

4

false pues 4 > length([10,21,34])

●● Veamos el programa.

Veamos la siguiente implementación:

```
vec: [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99];


lsearch(num,vec):=block
(   [res],
    res: lsearchAux(num,vec),
    if res>length(vec) then
       false
    else
       res
);


lsearchAux(num,vec):=block
(
    if vec=[] then
       1
    elseif num=first(vec) then
       1
    else
       1 + lsearchAux(num,rest(vec))
);
```

●● Análisis del programa.

En este caso el análisis del programa nos da:

```
lsearch(num,vec):=block ------------------------> T(n) = n + 2
(   [res],
    res: lsearchAux(num,vec),  ----------> n+1

    if res>length(vec) then ----> 1

       false -------> 1

    else

       res ---------> 1
);



lsearchAux(num,vec):=block --> T(n) = 1 + T(n-1) = n + 1 = O(n)
(
    if is(vec=[]) then ----------------------> 1 + T(n-1)

       1 ----------------------------> 1
```

```
    elseif num=first(vec) then
       1 ------------------------------> 1
    else
       1 + lsearchAux(num,rest(vec)) --> 1 + T(n-1)
);
```

Por lo tanto tenemos para resolver lsearchAux:

T(0) = 1

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

Al resolverla se obtiene:

T(n) = n+1

▬▬▬ 4. Búsqueda Lineal En Recursión de Cola.

●● Cómo debe funcionar el programa?

Tenemos el proceso:

```
lsearch(45, []);
lsearchAux(45, [], 1);
false
```

Tenemos el proceso:

```
lsearch(45, [10, 21, 34, 45, 50, 62, 73, 79, 91, 99]);
lsearchAux(45, [10, 21, 34, 45, 50, 62, 73, 79, 91, 99], 1);
lsearchAux(45, [21, 34, 45, 50, 62, 73, 79, 91, 99],     2);
lsearchAux(45, [34, 45, 50, 62, 73, 79, 91, 99],         3);
lsearchAux(45, [45, 50, 62, 73, 79, 91, 99],             4);
4
```

Veamos otro ejemplo:

```
lsearch(45, [10, 21, 34]);

lsearchAux(45, [10, 21, 34],  1);

lsearchAux(45, [21, 34],      2);

lsearchAux(45, [34],          3);

lsearchAux(45, [],            4);

false
```

●● Implementación del programa.

Tenemos el siguiente algoritmo que implementa la búsqueda lineal en recursión de cola:

```
vec: [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99];


lsearch(num,vec):=block
(
    lsearchAux(num,vec,1)
);



lsearchAux(num,vec,index):=block
(
    if vec=[] then
       false
    elseif num=first(vec) then
       index
    else
       lsearchAux(num,
                  rest(vec),
                  index+1)

)
```

●● Análisis del programa.

En este caso el análisis nos da una relación de recurrencia.

```
lsearchAux(num,vec):=block ---> T(n) = n+1 = O(n)
(
    lsearchAux(num,vec,1) ----------------------> n+1
);


lsearchAux(num,vec,index):=block  ---> T(n) = T(n-1) + 1 = n+1
(
    if vec=[] then
```

```
       false ----------------------------------> 1
    elseif num=first(vec) then
       index  ----------------------------------> 1
    else
       lsearchAux(num,    ---------------------> T(n-1) + 1
                  rest(vec),
                  index+1)
);
```

Al resolver la relación de lsearchAux se obtiene:

T(0) = 1

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

Al resolver la relación de recurrencia encontramos:

T(n) = n + 1

▬▬▬ 5. Búsqueda Lineal Iterativo.

●● Cómo debe funcionar?

```
lsearch(45, [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99]);

i:1  vec[1]=10 es diferente 45

i:2  vec[2]=21 es diferente 45

i:3  vec[3]=34 es diferente 45

i:4  vec[4]=45 entonces index:4

i:5  vec[5]=50 es diferente 45

i:6  vec[6]=62 es diferente 45

...

i:10 vec[10]=99 es diferente 45

return(index:4)
```

●● Implementación del Programa.

A continuación se muestra una posible implementación del código de la búsqueda lineal:

```
/* Un vector de Prueba */
```

```
vec: [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99];


/* Algoritmo de Búsqueda Lineal */

lsearch(num,vec):=block
(    [index],
     index:false,

     for i:1 thru length(vec) do
     (
          if num=vec[i] then
             index:i
     ),

     return(index)
);
```

Si se llama al programa:

```
(%i) lsearch(50,vector);
(%o) 5

(%i) lsearch(45,vector);
(%o) 4

(%i) lsearch(53,vector);
(%o) false
```

●● Análisis del programa.

Veamos como se realiza el análisis de este programa:

```
/* Un vector de Prueba */

vector: [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99];


/* Algoritmo de Búsqueda Lineal */

lsearch(num,vec):=block
(    [index],
     index:0, ----------------------> 1

     for i:1 thru length(vec) do ---> 2*n
     (
          if num=vec[i] then ---> 2
             index:i --------> 1
     ),
     return(index) -----------------> 1
);
```

Por lo tanto, el tiempo total de ejecución del programa está dado por:

$T(n) = 2*n + 2 = O(n)$

▬▬▬ 6. Otra Implementación de la Búsqueda Lineal Iterativo.

●● Cómo debe funcionar?

```
lsearch(45, [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99]);

i:1  vec[1]=10 es diferente 45

i:2  vec[2]=21 es diferente 45

i:3  vec[3]=34 es diferente 45

i:4  vec[4]=45 entonces index:4

return(index:4) y se detiene el proceso
```

●● Implementación del Programa.

Aquí tenemos otra posible implementación. En este caso, cuando se encuentra con el elemento que se busca se detiene el proceso.

Otra forma del iterativos

```
/* Un vector de Prueba */

vec: [ 10, 21, 34, 45, 50, 62, 73, 79, 91, 99];


/* Algoritmo de Búsqueda Lineal */

lsearch(num,vec):=block
(    [index],
     index:false,

     for i:1 thru length(vec) do
     (
          if num=vec[i] then
          (  index:i,
             return(index)
          )
     ),

    return(index)
);
```

Observemos que funciona bien:

```
(%i) lsearch(45,vector);
(%o) 4


(%i) lsearch(50,vector);
(%o) 5


(%i) lsearch(13,vector);
(%o) false
```

●● Análisis de la segunda implementación.

```
lsearch(num,vec):=block
(    [index],
     index:false, ----------------------> 1

     for i:1 thru length(vec) do -------> 3*n
     (
          if num=vec[i] then ------> 3
          (  index:i, --------> 1
             return(index) ---> 1
          )
     ),
    return(index) ----------------------> 1
);
```

Por lo tanto tenemos que:

T(n) = 3*n + 2

Que es igual a O(n)

▬▬▬ 7. El Caso Peor, Mejor y Promedio.

●● Peor Caso.

Tal como se ha analizado anteriormente, tanto en iteración, en recursión y en recursión de cola. El peor caso está dado por:

Peor_Caso = O(n)

●● Mejor Caso.

El mejor caso, consistiría en que el elemento que se busca se encuentre al inicio de la lista. Por lo que al buscarlo bastaría una comparación para encontrarlo. Por lo tanto.

Mejor_Caso = O(1)

●● Caso Promedio.

El caso promedio es más difícil de calcular.
Si se busca el primer elemento basta 1 comparación.
Si se busca el segundo elemento se necesitan 2 comparaciones.
Y así sucesivamente hasta llegar al último elemento.

Luego hay que dividir todos estos valores por el número de elementos.
En general

$$\text{Caso\_Promedio} = \frac{\#\_\text{Comparaciones\_de\_cada\_elemento}}{\#\_\text{de\_elementos}}$$

Por lo tanto, el caso promedio estaría dado por:

$$\left(\frac{1 + 2 + 3 + \ldots + n}{n} \right)$$

$$= \left(\frac{1}{n} \right) * \left(1 + 2 + 3 + \ldots + n \right)$$

$$= \left(\frac{1}{n} \right) * \left(\frac{n*(n+1)}{2} \right)$$

$$= \left(\frac{n*(n+1)}{2*n} \right)$$

$$= \left(\frac{n + 1}{2} \right)$$

$$= \left(\frac{n}{2} + \frac{1}{2} \right)$$

De esta manera:

Caso_Promedio = O(n/2) = O(n)

Para el cálculo del caso promedio se ha mantenido la suposición que todos los casos tienen la misma probabilidad de presentarse. Este supuesto no es siempre verdadero. Es por esta razón que el caso promedio es difícil de calcular y no se utiliza regularmente.

▬▬▬ 8. Resumen.

• Linear Search es uno de los algoritmos más sencillos y fáciles de programar.

• Se puede programar de forma iterativa o recursivo. En cualquier caso su tiempo de ejecución es O(n)

• En Linear Search, si el vector está ordenado o no, el tiempo de ejecución es siempre el mismo.

• Se puede calcular el peor caso, el mejor caso y el caso promedio. Sin embargo, lo usual es utilizar únicamente el peor caso.

▬▬▬ 9. Ejercicios.

●● Ejercicio 1.

Implemente el algoritmo de búsqueda secuencial en cualquier lenguaje que usted conozca.

●● Ejercicio 2.

Si para hacer una búsqueda secuencial, los valores del vector no estuvieran ordenados. Afecta eso el valor de la gran O del algoritmo?

●● Ejercicio 3.

Se puede mejorar el algoritmo de búsqueda secuencial, al utilizar el hecho que los elementos se encuentran ordenados. Por ejemplo, el algoritmo inicia al principio del vector, va buscando el número deseado. Pero cuando el número que revisa es menor que el número buscado se puede detener.

Cómo afecta este cambio el análisis de complejidad del algoritmo?