Análisis de algoritmos

Algoritmo: Conjunto de instrucciones sencillas, claramente especificado, que se debe seguir para resolver un problema.

Una vez que se ha diseñado un algoritmo, el siguiente paso es determinar la cantidad de recursos, como tiempo y espacio, que requerirá.

Un algoritmo que resuelve un problema pero demora un año en completarlo, difícilmente será de utilidad. Por otro lado un algoritmo que necesite 1 gigabyte de memoria RAM actualmente no es de gran utilidad.

Comparación de algoritmos.

Todo algoritmo tiene operaciones fundamentales y otras que no lo son tanto. Es importante poder determinar cuales son las líneas que son claves y condicionaran el desempeño del algoritmo, porque de ellas dependerá el poder medir el comportamiento de un algoritmo.
La complejidad en algoritmos se puede definir como la medida o tamaño de un problema. Se representa como una tasa de crecimiento de la cantidad de datos de entrada.

El análisis que se hace de los algoritmos para determinar su complejidad esta basado en conceptos matemáticos, pues es un análisis teórico y por lo tanto necesita un marco formal que sustente dicho análisis. A continuación se entregan como recordatorio alguno de estos sustentos.

Exponentes:

$$x^a x^b = x^{a+b}$$

$$\frac{x^a}{x^b} = x^{a-b}, x \neq 0$$

$$\left(x^a\right)^b = x^{ab}$$

$$x^n + x^n = 2x^n$$

$$2^n + 2^n = 2^{n+1}$$

Logaritmos:

En informática generalmente todos los logaritmos son en base 2, a menos que se especifique otra cosa.

$$x^a = b \Leftrightarrow \log_x b = a$$

$$\log_a b = \frac{\log_c b}{\log_c a}; c > 0$$

$$\log_n ab = \log_n a + \log_n b$$

$$\log_n\left(\frac{a}{b}\right) = \log_n a - \log_n b$$

$$\log_n a^b = b * \log_n a$$

$$\log_n a < a; \forall a > 0$$

$$\log_n 1 = 0$$

Series:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=1}^{n} \frac{1}{i} \approx \log_e n$$

$$\sum_{i=1}^{n} f(x) = nf(n)$$

Un criterio posible para elegir el mejor algoritmo entre varios puede ser la cantidad de recursos consumidos por el algoritmo: espacio de memoria y tiempo de ejecución.

Eficiencia en memoria o complejidad espacial de un algoritmo es la cantidad de almacenamiento necesario para ejecutar el algoritmo. Eficiencia en tiempo de ejecución o complejidad temporal indica el tiempo que demorara el algoritmo si las entradas tienden a crecer.

Notación asintótica

El análisis asintótico se emplea para explorar el comportamiento de una función o de una relación entre funciones, cuando algún parámetro de la función tiende hacia un valor asintótico.

Este tipo de análisis puede ser usado para mostrar que dos funciones son aproximadamente iguales entre si o que el valor de cierta función crece asintóticamente más rápidamente que la otra función.

Un convenio de notación que se usa de forma extensiva en el análisis asintótico es la notación O. Usando la notación O tendremos un medio para expresar la cota superior asintótica de una función.

Definición.

Sean $f(n)$ y $g(n)$ dos funciones arbitrarias tales que:
$f(n), g(n): \mathrm{Z}^+ \to \mathrm{Z}^+$, se dice que $f(n)$ es *"O grande"* de $g(n)$ y se escribe:
$f(n) = \mathrm{O}(g(n))$, si existen constantes positivas c y n_0 tales que

$$f(n) \le cg(n_0) \quad , \quad \forall n \ge n_0 \, .$$

Decir que $f(n) = \mathrm{O}(g(n))$ supone que $cg(n)$ es una cota superior del tiempo de ejecución del algoritmo.

Ejemplo1

Comprobar que la función: $f(n) = 3n^3 + 2n^2$ *es* $\mathrm{O}(n^3)$. Aplicando la definición tenemos que: $f(n) = 3n^3 + 2n^2$ Consideremos $g(n) = n^3$ y haciendo $n_0 = 0$ *y* $c = 5$, se tiene que:

$$3n^3 + 2n^2 \le 5n^3 \quad \forall n \ge 0 \quad .$$

Ejemplo 2

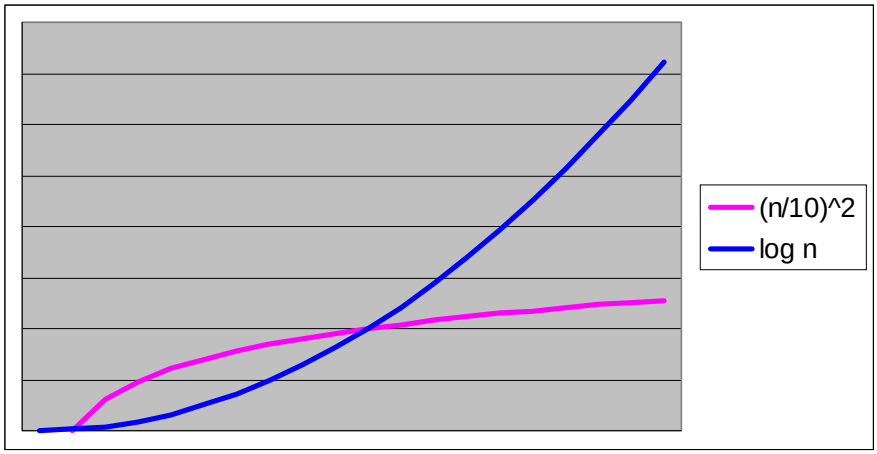
Si $f(n) = n^3 + 20n^2 + 100n$, Consideremos $g(n) = n^3$, $n_0 = 0$ y $c = 121$ Se tiene que:

$$f(n) = n^3 + 20n^2 + 100n \leq n^3 + 20n^3 + 100n^3 = 121n^3$$

Estos ejemplos demuestran que al especificar cotas de "**O grande**" para ecuaciones en las que aparecen potencias simples de n, podemos tranquilamente ignorar todos los términos excepto el de orden mas alto.

Esto se debe a que cuando n crece, el termino que contiene la mayor potencia de n domina a los otros términos en ecuaciones como la ya mencionadas.

Consideremos las funciones $f(n) = \log(n)$ $\wedge$ $g(n) = (n/4)^2$



para $a < n < b$, se tiene que $f(n) > g(n)$ sin embargo $g(n) \neq O(f(n))$, porque la desigualdad $g(n) \leq cf(n)$ no se satisface para todo $n \geq n_0$, independiente de lo grande que se escoja c.

Pero si podemos escribir $f(n) = O(g(n))$, dado que podemos hacer $n_0 = b$ y $c = 1$, el valor $cg(n)$ será mayor o igual que el de $f(n)$, $\forall n \geq n_0$.

Observación: Estos no son los únicos valores de n_0 y c para los cuales $f(n) \leq g(n)$.

La igualdad $f(n) = O(g(n))$ funciona solo en una dirección.

Dado que el termino $O(g(n))$ realmente especifica un conjunto infinito de funciones, esta sentencia dice que $f(n)$ pertenece al conjunto de funciones que pueden ser acotadas superiormente por un múltiplo fijo de $g(n)$, cuando n es suficientemente grande.

La notación $O(g(n)) = f(n)$ *no debe ser usada nunca.*

Como estamos tratando con cotas superiores asintóticas, si $f(n) = O(g(n))$ y $h(n)$ es una función cuyos valores son mayores que los de $g(n)$, cuando n es suficientemente grande, entonces $f(n) = O(h(n))$.

Ejemplo:

Si $f(n) = n^2$, es correcto establecer $f(n) = O(n^3)$, sin embargo esta cota superior no esta ajustada.

Para simplificar la notación diremos que $O(f(n)^2) = O(f(n))^2$

Resumen

Cuando se dice que $f(n) = O(g(n))$, se esta garantizando que la función $f(n)$ crece a una velocidad no mayor que $g(n)$.

Propiedades de $O()$

Para cualquier par de funciones $f(n) \wedge g(n)$ se verifican las siguientes propiedades:

$$cO(f(n)) \quad es \quad O(f(n))$$
$$O(f(n) + g(n)) \quad es \quad max(O(f(n)), O(g(n)))$$
$$O(f(n)) + O(g(n)) \quad es \quad O(f(n) + g(n))$$
$$O(f(n))O(g(n)) \quad es \quad O(f(n)g(n))$$
$$O(O(f(n))) \quad es \quad O(f(n))$$

Funciones de complejidad algorítmica más usuales ordenadas de mayor a menor eficiencia son:

- $O(1)$: Complejidad constante
- $O(\log(n))$: Complejidad logarítmica, suele aparecer en algoritmos con iteración o recursión no estructurada (búsqueda binaria).
- $O(n)$: Complejidad lineal, es en general una complejidad buena y bastante usual. Suele aparecer, en la evaluación de ciclos simples cuando la complejidad de las operaciones interiores es constante o en algoritmos de recursión estructurada.
- $O(n\log(n))$: Aparece en algoritmos con recursión no estructurada (por ejemplo Quick sort) y se considera una complejidad buena.
- $O(n^2)$: Complejidad cuadrática, aparece en ciclos o recursiones doblemente anidadas.
- $O(n^3)$: Complejidad cubica, aparece en ciclos o recursiones triples, para un valor grande de n empieza a crecer en exceso.
- $O(n^k)$: Complejidad polinómica $(n > 3)$, si k crece la complejidad del programa es bastante mala.

- $O(2^n)$: Complejidad exponencial, debe evitarse en la medida de lo posible, puede aparecer en rutinas recursivas que contengan dos o más llamadas internas. En problemas done aparece esta complejidad suele hablarse de *"explosión combinatoria"*.

Ejercicios

1. Comparar los siguientes pares de funciones usando notación asintótica.

$f(n)$	$g(n)$
$10^{-3}n^4$	$10^3 n^3$
n^2	$n\log(n)$
$\log(n)$	$\log(\log(n))$
2^{n^2}	2^{2^n}
$100n + \log_{10}(n)$	$n + (\log(n))^2$
$\log(n)$	$\log(n^2)$
$n^{\log(n)}$	n

Análisis de complejidad de algoritmos no recursivos

Para evaluar el calculo de la complejidad de un algoritmo es necesario evaluar la complejidad de las operaciones que lo conforman.

Complejidad de asignaciones y expresiones simples: El tiempo de ejecución de toda instrucción de asignación simple, de la evaluación de una expresión formada por términos simples o de toda constante es **O(1).**

Secuencia de Instrucciones: El tiempo de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecución respectivos. Para una secuencia de dos instrucciones $\mathbf{I_1}$ e $\mathbf{I_2}$ tenemos que su tiempo de ejecución viene dado por la suma de los tiempos de ejecución de $\mathbf{I_1}$ **e** $\mathbf{I_2}$, es decir:

$$\mathbf{T(I_1 ; I_2) = T(I_1) + T(I_2)}$$

Aplicando la regla de la suma su orden es:

$$\mathbf{O(T(I_1 ; I_2)) = O(T(I_1) + T(I_2))}$$
$$\mathbf{O(T(I_1 ; I_2)) = max(O(T(I_1)), O(T(I_2)))}$$

Instrucciones Condicionales: El tiempo de ejecución requerido por una instrucción condicional **IF – THEN**, es el necesario para evaluar la condición, mas el requerido para el conjunto de instrucciones que se ejecutan cuando se cumple la condición.

T(IF – THEN) = T(condición) + T(rama THEN)

El tiempo para una instrucción condicional del tipo IF – THEN – ELSE es el resultante de evaluar la condición mas el máximo entre los requeridos para ejecutar el conjunto de instrucciones de las ramas THEN y ELSE.

T(IF – THEN – ELSE)= T(condición) + max(T(rama THEN), T(rama ELSE))

Si aplicamos la regla de la suma tenemos que:

O(T(IF–THEN–ELSE))= O(T(condición)) + max(O(T(rama THEN)), O(T(rama ELSE)))

Instrucciones de iteración: La complejidad en tiempo de un ciclo **FOR** es el producto del numero de iteraciones por la complejidad de las instrucciones del cuerpo del ciclo. Para ciclos **WHILE, LOOP y REPEAT** se sigue la regla anterior considerando la estimación del numero de iteraciones para el peor caso posible.

Llamadas a procedimientos: La evaluación de la complejidad de la llamada a un procedimiento esta dada por el tiempo requerido para ejecutar el cuerpo del procedimiento, no se tiene en cuenta el tiempo necesario para efectuar el paso de los argumentos.

Ejemplos:

1) x = x + 1; (x++)
   Es O(1); Constante.

2) for(i=1; i < n; i++)
   x=x+1;

$$O(T(n)) = O(\sum_{i=1}^{n} 1)$$
$$O(T(n)) = O(n);\ Lineal$$

3) for(i=1; i < n; i++)
   for(j=1; j < n; j++)
   for(k=1; k<n; k++)
   x=x+1;

$$O(T(n)) = O(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 1)$$
$$O(T(n)) = O(n^3);\ Lineal$$

4) if ((n mod 2) == 0)
   for(i=1 i < n ; i++)
   x=x++;

$$\begin{aligned} O(T(n)) &= O(T(condicion) + O(T(ramaTHEN)) \\ &= O(O(1) + T(FOR)) \\ &= O(1) + O(\sum_{i=1}^{n} 1) \\ &= O(n); \quad Lineal \end{aligned}$$

5)
```
i=1;
while(i<n){
    x++;
   i=i+2;
}
```

$$\begin{aligned} O(T(n)) &= max(O(1), O(T(WHILE)) \\ &= max(O(1), O(\sum_{i=1}^{\text{int}(n/2)} 1)) \\ &= max(O(1), O(\text{int}(n/2)) \\ &= max(O(1), O(n)) \\ &= O(n); \quad Lineal \end{aligned}$$

6)
```
for(i=1; i<n; i++)
        for(j=1; j<i; j++)
                x++;
```

$$O(T(n)) = O(\sum_{i=1}^{n}\sum_{j=1}^{i}1)$$
$$= O(\sum_{i=1}^{n} i)$$
$$= O\left(\frac{n(n+1)}{2}\right)$$
$$= O(n^2)$$

7)
```
x=1;
while(x < n)
        x=2*x;
```

$$O(T(n)) = max(O(1), O(T(WHILE)))$$

El cálculo del número de iteraciones del ciclo while equivale a calcular el valor de la variable t en el siguiente conjunto de instrucciones:

```
x=1;
t=1;
while(x < n){
        x=2*x;
        t++
}
```

En este caso tenemos que:

$$n = 8 \Rightarrow t = 4$$
$$n = 16 \Rightarrow t = 5$$
$$n = 32 \Rightarrow t = 6$$

De donde podemos deducir que:

$$2^{t-1} \geq n; \quad \textit{despejando t nos queda}$$
$$\log_2 2^{t-1} = \log_2 n$$
$$(t-1)\log_2 2 = \log_2 n$$
$$t = \log_2 n + 1$$

Por lo tanto la complejidad del ciclo while es:

O(T(WHILE))=O(log n +1)

O(T(WHILE))=O(log n)

Obteniendo finalmente:

O(T(n))=max(O(1),O(log n))
O(T(n))=**O(log n).**

Tarea: Evalúe la complejidad del algoritmo siguiente (¿que calcula?).

```
int i, aux, menor, mayor;
if (n<=1)
        return(1)
else {
        mayor = 1;
        menor= 1;
        for(i=2; i < n; i++){
                aux=menor;
                menor=mayor;
                mayor = aux;
        }
        return(mayor);
}
```

Análisis de complejidad de algoritmos de tiempo recursivos.

Método para el cálculo de la complejidad en algoritmos recursivos.

Como Ud. debe haber visto, el análisis de la complejidad del algoritmo que calcula el factorial iterativo, tiene complejidad lineal.

Para el ejemplo del caso recursivo se plantea el problema en términos recurrentes:

$$n! = \begin{cases} 1 & si \quad n = 0 \\ n(n-1)! & si \quad n > 0 \end{cases}$$

lo que nos interesa es determinar el tiempo de ejecución del algoritmo $T(n)$, a partir de la función podemos plantear la siguiente expresión que relaciona el tiempo de ejecución en términos recurrentes:

$$T(n) = \begin{cases} 1 & si \quad n = 0 \\ T(n-1)+1 & si \quad n > 0 \end{cases}$$

Para la solución de este tipo de ecuaciones se plantean 4 métodos:

a) Sustituir las ecuaciones por su igualdad hasta llegar a cierto $T(n_0)$ conocido. Este método se llama de expansión de recurrencia.

b) Elegir una función $f(n)$ cota superior y una función $g(n)$ cota inferior del mismo orden y usar la ecuación de recurrencia para probar que $g(n) \leq T(n) \leq f(n) \quad \forall n$, a este método se le llama de acotación.

c) Suponer una solución $f(n)$ y usar la recurrencia para demostrar que $T(n) \leq f(n)$.

d) Emplear la solución general para ciertas ecuaciones de recurrencia de tipos comunes.

Volviendo al ejemplo del factorial y utilizando el método de expansión de recurrencias, tenemos que:

$$
\begin{aligned}
T(n) &= T(n-1)+1 \\
&= (T(n-2)+1)+1 \\
&= T(n-2)+2 \\
&= T(T(n-3)+1)+2 \\
&= T(n-3)+3 \\
&\vdots \\
T(n) &= T(n-k)+k \qquad \textit{para el } k-\textit{esimo término}
\end{aligned}
$$

(*En este punto debemos considerar el valor de* $T(n)$ *conocido*)

Si elegimos un $k = n$ *tenemos que*

$$T(n) = T(0)+n$$

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

$$O(T(n)) = O(n) \quad \Rightarrow \quad \textit{complejidad de orden lineal}$$

Ejemplo 2

```
int recursiva(n){
        if  (n <= 1)
return(5)
        else
return(recursiva(n-1)+recursiva(n-1))
}
```

Expresando la función en términos de tiempo de ejecución recursivo tenemos que:

$$T(n)=\begin{cases}1 & si \quad n\le 1\\ 2T(n-1)+1 & si \quad n>1\end{cases}$$

y aplicando el método de expansión de recurrencias para resolverlo tenemos que:

$$\begin{aligned}T(n)&=2T(n-1)+1\\ &=2(2T(n-2)+1)+1\\ &=2\ T(n-2)+3\\ &=2\ (2T(n-3)+1)+3\\ &=2^3T(n-3)+7\\ &\ \ \vdots\\ T(n)&=2^kT(n-k)+2^k-1\end{aligned}$$

$$\begin{aligned}Si\quad k&=n-1\\ &2^{n-1}T(1)+2^{n-1}-1\\ &2^{n-1}+2^{n-1}-1\\ &2*2^{n-1}-1\\ &2^n-1\end{aligned}$$

$O(T(n))=O(2^n) \Rightarrow$ *complejidad de orden cuadrática*

Ejemplo 3

```
int recursiva_1(n){
        if (n <= 1)
return(1)
        else
return(2*recursiva_1(n div 2))
}
```

Expresando la función en términos de tiempo de ejecución recursivo tenemos que:

$$T(n) = \begin{cases} 1 & si \quad n \le 1 \\ T(n/2)+1 & si \quad n > 1 \end{cases}$$

y aplicando el método de expansión de recurrencias para resolverlo tenemos que:

$$\begin{aligned} T(n) &= T(n/2)+1 \\ &= T(((n/2)/2)+1)+1 \\ &= T(n/4)+2 \\ &= T(((n/4)/2)+1)+2 \\ &= T(n/8)+3 \\ &\vdots \\ &= T(n/2^K)+k \end{aligned}$$

$$Si \quad k = \log \; n$$

$$T(n/2^{lol_2 n})+\log_2 n$$

$$O(T(n)) = O(\log_2 n) \quad \Rightarrow \quad complejidad \quad de \quad orden \quad \log \, arítmico$$

Ejemplo 4

```
int recursivo_2(int m, int x, int r){
    int i, r1;
    if (n==0)
r = x;
          else {
for(i=1; i<=n; i++){
    x = x+1;
    r = r1;
                }
     recursivo_2 (n-1, x, r1);
      }
}
```

Se plantea la ecuación de recurrencia

$$T(n) = \begin{cases} 1 & si \quad n = 0 \\ n + T(n-1) & si \quad n > 0 \end{cases}$$

Queda de tarea calcular la complejidad utilizando el método de expansión de recurrencias.

EJERCICIOS.

1) Un ejemplo clásico que se utiliza para demostrar la potencia de la recursión en la resolución de problemas es el de las torres de Hanoi. En este rompecabezas, se dan tres postes etiquetados A, B y C. Inicialmente, una torre de ocho discos de diferentes tamaños están apilados en el poste A en orden de tamaño decreciente. Esto es el disco más grande esta en el fondo y el más pequeño esta en la cima de la torre como se muestra en la figura. El objetivo es trasladar toda la torre a uno de los otros postes, moviendo un disco cada vez y no colocando nunca un disco más grande encima de otro más pequeño.

   a) Presente un procedimiento recursivo para resolver el problema.
   b) Si denotamos como operación básica el acto de mover un disco de un poste a otro, entonces ¿Cuál es la complejidad en tiempo del algoritmo?.
   c) Escriba un programa en lenguaje C y compare su ejecución con el punto (b).
   d) La leyenda dice que un en el inicio del tiempo, Dios colocó una torre de 64 discos de oro encima de la primera de las tres agujas de diamantes, y ordeno que un grupo de sacerdotes de Brahma debían transferirlos a la tercera aguja usando las reglas establecidas previamente para las Torres de Hanoi. Cuando los sacerdotes completasen su tarea, la torre se derrumbaría y el mundo terminaría. Si asumimos que los sacerdotes pueden transferir un disco de una aguja a otra en un segundo, ¿cuánto tiempo tardaran en completar su tarea?.

2) Usar la definición de la notación O para responder a las siguientes cuestiones:

   a) ¿Es $2^{n+1} = O(2^n)$?
   b) ¿Es $2^{2n} = O(2^n)$?
   c) Suponiendo que f(n) = O(g(n)), ¿implica ello que $2f(n)=O(2^{g(n)})$?

3) Escribir una función en seudocódigo que calcule la siguiente relación de recurrencia:

$$T(n) = \begin{cases} 1 & si \quad n = 1 \\ 2T(n-1) + n & si \quad n > 1 \end{cases}$$

4) La siguiente es una famosa relación de recurrencia conocida como la función de Ackermann:

$$A(m,n) = \begin{cases} n+1 & si \quad m = 0 \ y \ n \geq 0 \\ A(m-1,1) & si \quad m \geq 1 \ y \ n = 0 \\ A(m-1, A(m, n-1)) & si \quad m, n \geq 1 \end{cases}$$

   Observe que uno de los parámetros de esta relación de recurrencia es a su vez recursivo. Calcular los valores de A(2,2), A(2,3) y A(2,4).

5) Escriba un algoritmo que, dados $x \in \Re$ y $n \in Z_0^+$, calcule x^n. Determine el numero de multiplicaciones efectuadas por el algoritmo.

6) Diseñe un procedimiento recursivo que invierta un vector de largo $n \in N$, utilizando solo una variable escalar auxiliar. Calcule el numero de cambios realizados entre los elementos del vector.

7) Escriba un procedimiento que convierta recursivamente un entero dado en un string que contenga todos sus dígitos. Determine la complejidad de su algoritmo.

8) Búsqueda de un valor dado n en un vector de enteros v[1..max].

   a) Escribir una función recursiva que detecte la posición de n en v. En caso de no pertenecer debe retornar un cero.

   b) Repita el problema del punto anterior, pero ahora suponga que el vector v esta ordenado.

   c) Analice la complejidad para los casos peores de los puntos anteriores.

9) Sea a[1..m, 1..n] una matriz de elementos enteros ordenados de modo que:

   a[i, j] < a[i, j+1], i = 1, ..., m y j = 1, ... , (n-1)
   a[i,n] < a[i+1, 1], i = 1, ... , (m-1)

   a) Elabore un algoritmo que indique si un entero x pertenece a la matriz a y que efectúe en el pero de los casos (m+n) comparaciones.

   b) Elabore un algoritmo que efectúe menos comparaciones que el caso anterior.

10) Considere el vector x[1..n] de elementos enteros. Se desea encontrar el mayor elemento del arreglo.

    a) Escriba un algoritmo para este efecto, utilizando un algoritmo trivial, e indique la cantidad de comparaciones efectuadas con los elementos del vector.

    b) Diseñe una versión recursiva del mismo, utilizando particiones binarias sucesivas del vector y realizando la búsqueda en estas particiones.