

Complejidad algorítmica

...

Algoritmos y Estructuras de Datos
2023

¿Qué estrategias se le ocurren para encontrar un elemento en una lista? ¿Cómo determinaría cual es mejor método?

Evaluando algoritmos

- Las principales variables para evaluar el desempeño de un algoritmo son el tiempo de ejecución y el espacio en memoria que requiere.
- En general, el tiempo de ejecución de un algoritmo crece con el tamaño de los datos de entrada, aunque existen otros factores que pueden afectar (*hardware*, lenguaje de implementación, etc).
- Para determinar el tiempo de ejecución de un algoritmo es necesario conocer todas las **operaciones primitivas** que se llevan a cabo.

(algunas) Operaciones primitivas

- Asignar variables
- Operaciones aritméticas
- Comparación de variables
- Acceder a un elemento de una lista con su índice
- Llamado a una función
- Retornar valores

¿Qué tiempo medir?

2	5	23	7	87	4	17	11
---	---	----	---	----	---	----	----

- Buscar **23** en la lista -> 3 comparaciones
- Buscar **17** en la lista -> 7 comparaciones
- Buscar **2** en la lista -> 1 comparación
- Buscar **9** en la lista -> 8 comparaciones

¿Qué tiempo medir?

- En entradas del mismo tamaño, un algoritmo puede correr en distintos tiempos.
- Una opción para determinar el tiempo podría incluir el cálculo del promedio de diferentes corridas.
- Esta estrategia implicaría numerosas corridas y un análisis estadístico complejo de la distribución de las muestras analizadas.
- Una variable más sencilla de determinar es “el peor caso”.

Comportamientos más frecuentes: Constante

- El comportamiento más simple es el que requiere tiempo constante.
- Sin importar el tamaño de un objeto, las operaciones siempre van a requerir el mismo tiempo.
- Ejemplo: acceder a un valor en un objeto de la clase **list**.

```
lista1 = ["2", "5", "23", "7", "87", "4", "17", "11"]  
lista2 = ["5", "7", "4", "11"]  
lista1[2]  
lista2[2]
```

Comportamientos más frecuentes: Lineal

- Cuando se deben recorrer y realizar operaciones básicas sobre todos los elementos de una lista, el tiempo va a depender proporcionalmente al tamaño de la lista.
- Ejemplo: búsqueda de mayor o menor

```
max = float("-inf")
for i in lista1:
    if i > max:
        max = i
print(max)
```


Comportamientos más frecuentes: Cuadrática

- Cuando se definen bucles anidados, por cada elemento de una lista se recorren todos los elementos de la segunda lista.
- Para los n elementos de la primer lista se realizaran m operaciones, siendo m el número de elementos de la segunda lista (en total $n*m$ operaciones)
- Ejemplo: búsqueda de elementos repetidos en una lista

```
for i in lista1:
    for j in lista2:
        if i == j:
            print (i)
```

Comportamientos más frecuentes: Exponencial

- En los casos donde se va multiplicando el número de operaciones por cada elemento de un grupo, el crecimiento se denomina exponencial.
- Un ejemplo de crecimiento exponencial es el código de Fibonacci recursivo.
- Cada llamada a la función implica 2 llamadas recursivas y, a su vez, cada llamada recursiva implica 2 llamadas más.

```
def fibonacci(valor):  
    if valor > 1:  
        return fibonacci(valor-1) + fibonacci(valor-2)  
    else:  
        return 1
```

Comportamientos más frecuentes: Logarítmica

- El mejor de los casos es el comportamiento logarítmico.
- El tiempo de ejecución aumenta con el tamaño de los datos, pero más lentamente que un algoritmo lineal.
- Ejemplo: **búsqueda binaria** (más sobre búsqueda binaria al final de la presentación)

Notación “Big Oh”

- En un algoritmo, todas las operaciones primitivas aportan a su complejidad
- La notación “Big Oh” es una forma de representar esta complejidad.
- Para ello, se listan todas las operaciones de un algoritmo y su comportamiento.
- La notación “Big Oh” plantea que sólo se tome en cuenta el término de mayor exponente, ya que es el que más rápido crece.

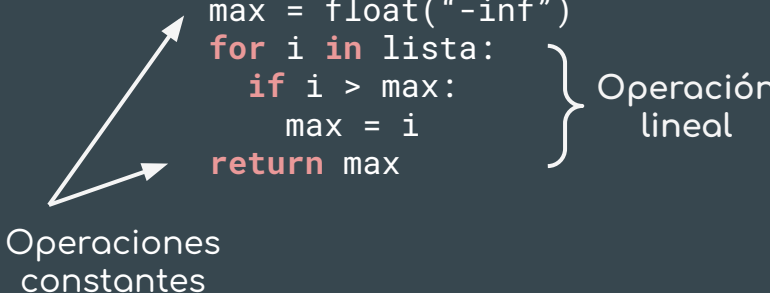
Notación “Big Oh”

- La función “maximo” incluye 3 partes:
 - la definición de la variable “max”
 - el recorrido de la lista
 - retornar el valor
- La primera y la última parte tienen complejidad constante, mientras que la segunda tiene complejidad lineal.
- Por lo tanto, se establece que la función “maximo” tiene complejidad lineal o **$O(n)$**

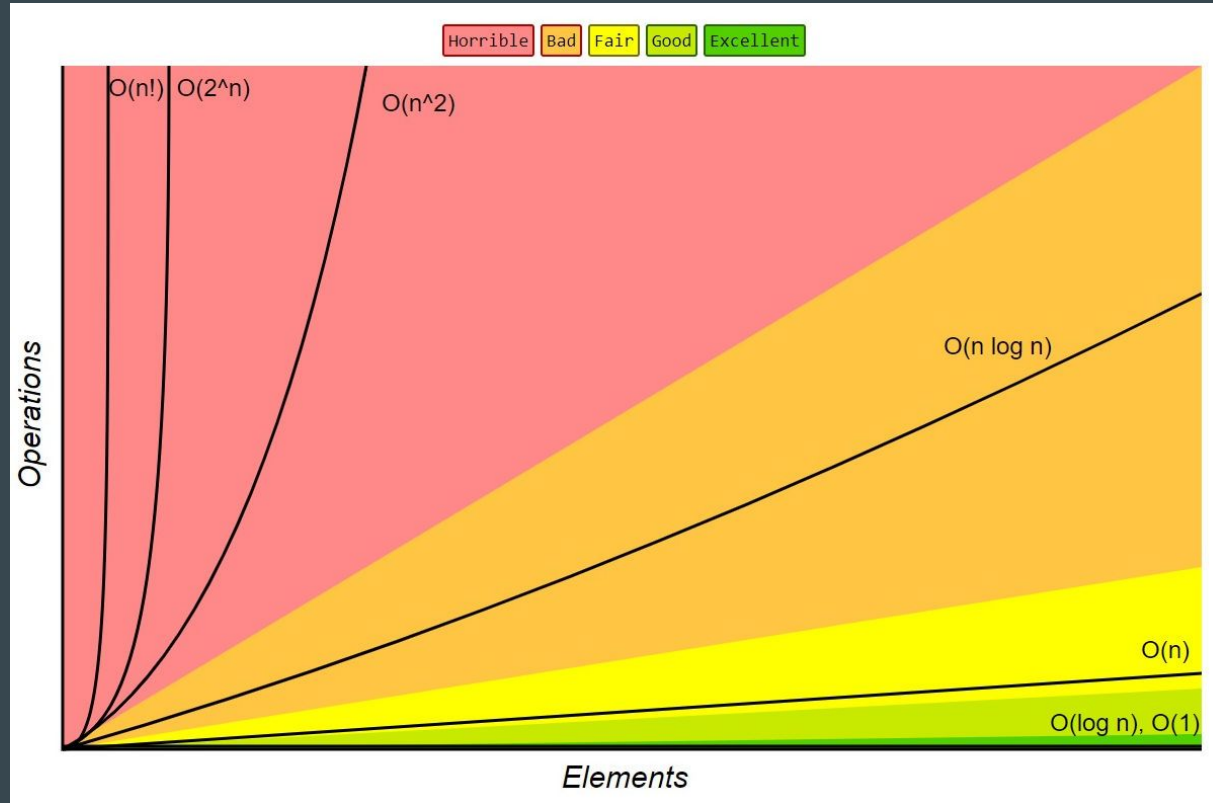
```
def maximo(lista):  
    max = float("-inf")  
    for i in lista:  
        if i > max:  
            max = i  
    return max
```

Operaciones constantes

Operación lineal



Comparación entre complejidades



Métodos de búsqueda

...

Búsqueda secuencial

2	5	23	7	87	4	17	11
---	---	----	---	----	---	----	----

- La forma más simple de buscar un elemento en una lista es compararlo con todos los elementos de dicha lista.
- Para ello es necesario recorrer la lista hasta el final o hasta encontrarlo.

```
def busqueda_secuencial(lista, e):  
    for i in lista:  
        if e == i:  
            return True  
    return False
```


Búsqueda binaria

- Una forma más eficiente de buscar un elemento es partiendo de una lista ordenada.
- Si el elemento a buscar es mayor que la mediana de la lista, solo se busca entre los elementos más grandes; si es menor, solo se buscan más pequeños

```
buscar(lista, buscado):  
    si buscado == mediana:  
        devolver True  
    si buscado < mediana:  
        buscar (lista[:pos(mediana)], buscado)  
    sino:  
        buscar (lista[pos(mediana):], buscado)
```

Búsqueda binaria

Ejemplo: buscar de manera binaria el número **17**

2	4	5	7	11	17	23	87
---	---	---	---	----	----	----	----

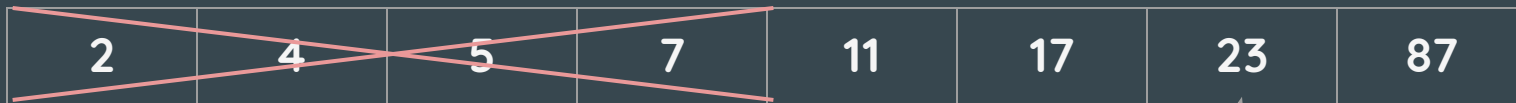


Mediana: valor del medio de la lista (o el siguiente si n es par)

```
busqueda_binaria(lista, 17):  
    si 17 == 11:  
        devolver True  
    si 17 < 11:  
        busqueda_binaria (lista[:len(lista)//2], 17)  
    sino:  
        busqueda_binaria (lista[len(lista)//2:], 17)
```

Búsqueda binaria

Ejemplo: buscar de manera binaria el número **17**



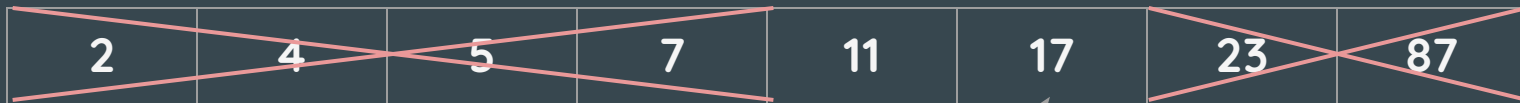
2	4	5	7	11	17	23	87
---	---	---	---	----	----	----	----

Mediana

```
busqueda_binaria(lista, 17):  
    si 17 == 23:  
        devolver True  
    si 17 < 23:  
        busqueda_binaria (lista[:len(lista)//2], 17)  
    sino:  
        busqueda_binaria (lista[len(lista)//2:], 17)
```

Búsqueda binaria

Ejemplo: buscar de manera binaria el número **17**



Mediana

```
busqueda_binaria(lista, 17):  
    si 17 == 17:  
        devolver True  
    si 17 < 17:  
        busqueda_binaria (lista[:len(lista)//2], 17)  
    sino:  
        busqueda_binaria (lista[len(lista)//2:], 17)
```

Búsqueda secuencial vs. binaria

- En el peor caso (elemento no presente), la búsqueda secuencial debe recorrer toda la lista.
- Como el tiempo de ejecución crece proporcionalmente al número de elementos de la lista, la complejidad es **lineal: $O(n)$**
- Durante la búsqueda binaria, en cada llamada recursiva se eliminan la mitad de los datos.
- En el peor caso, el algoritmo posee complejidad **logarítmica o $O(\log_2(n))$**

Implementando busquedas

```
def busqueda_secuencial (lista, valor):  
    for i in lista:  
        if valor == i:  
            return True  
    return False
```

```
def busqueda_binaria (lista, valor):  
    if valor != lista[0] and len(lista) <= 1:  
        return False  
    elif valor == lista[len(lista)//2]:  
        return True  
    elif valor > lista[len(lista)//2]:  
        return busqueda_binaria (lista[len(lista)//2:], valor)  
    elif valor < lista[len(lista)//2]:  
        return busqueda_binaria (lista[:len(lista)//2], valor)
```

Bibliografía

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Capítulo 3, 4.1.3
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press. Capítulo 3