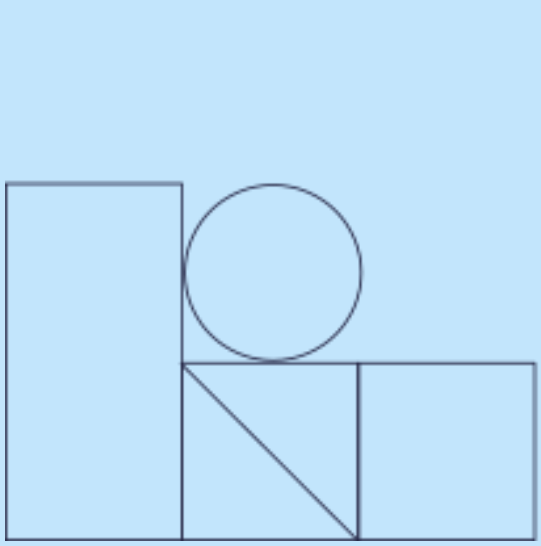
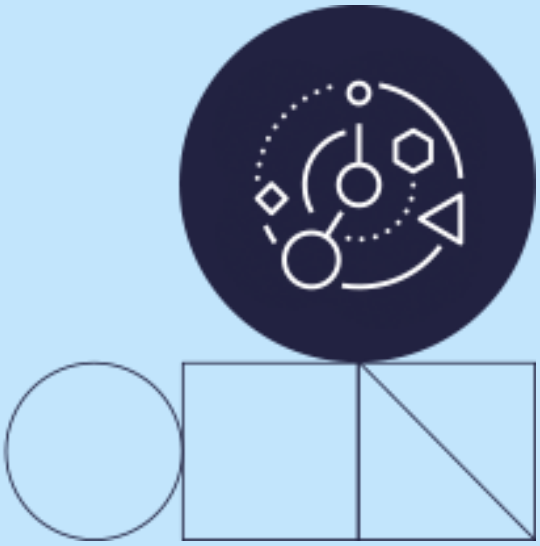
IBM SkillsBuild

IBM

Programando en Python

Algoritmos de búsqueda lineal y binaria





Índice

Introducción 3

Búsqueda lineal 4

Búsqueda binaria 5

Comparación de la eficiencia de los algoritmos 9

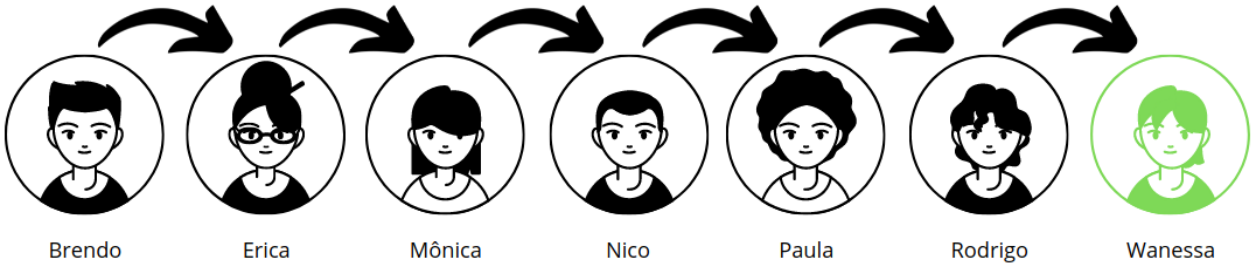
Introducción

Como hemos visto en temas anteriores, actualmente, se ha hecho más visible la necesidad de contar con algoritmos más eficientes para nuestras aplicaciones, ya sea por la cantidad de datos procesados o por la necesidad de respuestas rápidas. Esto nos lleva a uno de los principales fundamentos del desarrollo de software: analizar la complejidad de los algoritmos.

Un ejemplo práctico: imaginemos que estamos desarrollando una guía telefónica y somos responsables de crear la función de búsqueda de contactos asumiendo que todos los contactos ya están en orden alfabético.

Búsqueda lineal

Para hacerlo más fácil y evitar errores, **desplacémonos por la lista** recorriendo los contactos uno a uno en busca de un contacto en concreto. Supongamos que se solicitó el contacto de Wanessa:



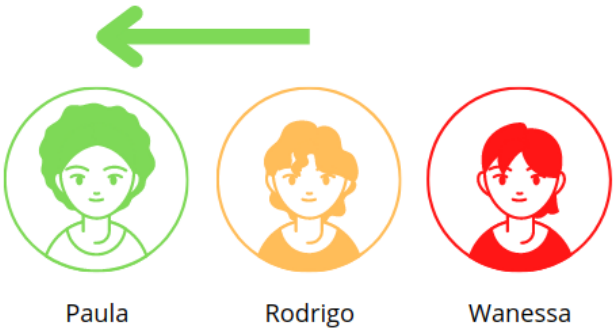
Nos encontramos con que hemos repasado toda la lista hasta encontrarlo. La solución actual funciona perfectamente y con un rendimiento aceptable. Ahora imaginemos que nuestra solución se vendió a una empresa multinacional que tiene miles o incluso millones de contactos.

En esta ocasión tendremos que desplazarnos por la lista millones de contactos y recorrerla hasta encontrar el contacto solicitado. Como vemos, para este tipo de búsqueda, la solución lineal parece no ser la más efectiva.

Búsqueda binaria

Pensemos en una solución mejor. Sabemos que la lista ya está en orden alfabético, entonces, podemos desglosarla y comparar el contacto que buscamos con el contacto del medio, comprobar la dirección que debemos tomar y eliminar el resto de la lista. Por tanto, seguiremos los pasos descritos para encontrar el contacto de Paula:

Seleccionamos el contacto del medio, Nico, para comparar con el contacto que buscamos, Paula, y descartamos la mitad de la lista en la que estamos seguros de que no estará la persona buscada.

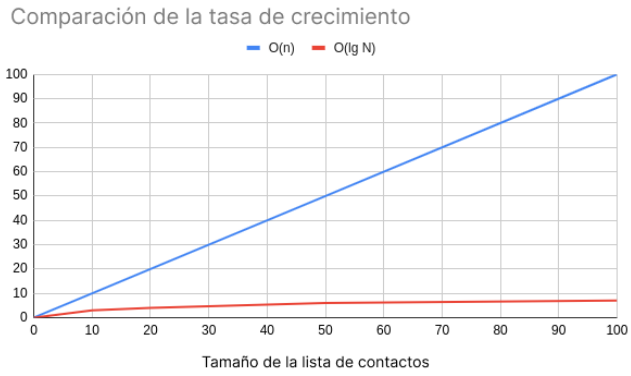


En esta iteración, elegimos nuevamente al contacto en el medio de la lista, Rodrigo, y verificamos en qué dirección debemos ir, descartando la otra mitad. Entonces encontramos a Paula.

¿Cuál sería la diferencia entre las dos soluciones?

Tengamos en cuenta que en la primera solución, independientemente del tamaño de la lista, debemos recorrer todos los contactos hasta encontrar el que queremos. Si tenemos suerte, el contacto que buscamos podría ser el primero de todos. Sin embargo, si no tenemos tanta suerte y buscamos el último contacto, tendremos que recorrer toda la lista, lo que sería el peor de los casos para el algoritmo. Con eso en mente, podemos determinar que nuestra primera solución ejecuta una función que crece linealmente con el tamaño de la lista de contactos.

En la segunda solución, en cada iteración eliminamos la mitad de la lista, por lo que no es necesario revisarla por completo. De esta forma optimizamos la búsqueda. Esta solución realiza una función que crece a una tasa logarítmica considerando el tamaño de la lista de contactos.



Ahora está claro que la solución logarítmica tiene un desempeño extremadamente más eficiente que la solución lineal, y podemos confirmarlo a través del gráfico anterior, donde el eje de la cantidad de operaciones realizadas por las dos funciones es bastante desigual.

Profundizando un poco más, en una lista con 1 millón de contactos solo es necesario realizar 20 operaciones de comparación hasta dar con el contacto deseado. Con esto podemos empezar a entender la importancia del análisis de la complejidad de los algoritmos.

Otro ejemplo, imaginemos tener que desarrollar una función de búsqueda de estudiantes de para que sea posible loguearse en un sitio web. Después de encontrar a la persona, nuestra intención es autenticarla para autorizar el acceso a la plataforma. Para ello recibimos un listado con todos los nombres de las personas registradas.

En primer lugar, usaremos la búsqueda lineal. Entonces decidimos revisar toda la lista buscando a la persona a autenticar y llegamos a la siguiente implementación:

```
from array import array

def buscar(lista, nombre_buscado):
    tamano_lista = len(lista)
    for actual in range(0, tamano_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos =
sorted(importa_lista('../data/lista_alumn
os'))
    posicion_del_alumno =
buscar(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

Hemos desarrollamos la función de búsqueda con una lista que contiene 7 estudiantes y también simulamos la búsqueda con aproximadamente 85,000. Todo funcionó bien y con un rendimiento aceptable, como muestra el siguiente algoritmo:

```
def buscar(lista, nombre_buscado):
    tamano_lista = len(lista)
    for actual in range(0, tamano_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos =
sorted(importa_lista('../data/lista_alumn
os'))
    posicion_del_alumno =
buscar(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()




def importar_lista(archivo):
    lista = []
    with open(archivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
```

```
        lista.append(line)
    return lista

def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    for actual in range(0,
tamano_de_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos =
sorted(importa_lista('../data/lista_aluno
s'))
    posicao_do_aluno =
busca(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

Ahora supongamos que el cliente nos confirma que recibiremos aproximadamente 3000 logins de personas nuevas al día siguiente.

Decidimos hacer una simulación simple para ver cómo reaccionaría nuestra búsqueda ante esta cantidad de solicitudes. Y pensando en el peor de los casos, siempre optaremos por la búsqueda de la última persona de la lista.

```
from array import array

def importar_lista(archivo):
    lista = []
    with open(arquivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
        lista.append(line)
    return lista
```

```
def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    for actual in range(0,
tamano_de_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos = ["Brendo",
"Erica", "Monica", "Nico", "Paulo",
"Rodrigo", "Wanessa"]
    for i in range(0, 3500):
        posicion_del_alumno =
buscar(lista_de_alumnos, "Wanessa")
        print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

Cambiamos la llamada de búsqueda y recorrimos un bucle para simular 3500 solicitudes de búsqueda. Al ejecutar notamos que el tiempo se ha incrementado significativamente.

¿Por qué sucedió esto? Hagamos un cálculo simple para responder a esta pregunta:

Para cada una de las 3500 solicitudes de búsqueda se realizaron 85 mil comparaciones, es decir: (3500*85000) = 297500000 operaciones.

Y si la lista de personas o solicitudes crece, el algoritmo crecerá linealmente a estas cantidades. Así que asumimos que cada uno de estos algoritmos es O(N).

Ahora vamos a optimizar el algoritmo.

Pensando en los alumnos que recibiremos, vamos a optimizar este algoritmo utilizando la técnica de divide y vencerás mediante búsqueda binaria.

```
from array import array

def importar_lista(archivo):
    lista = []
    with open(arquivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
        lista.append(line)
    return lista

def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    inicio = 0
    fim=tamano_de_lista-1

    while inicio<=fin:
        medio=(inicio+fin)//2
        if lista[medio] ==
nombre_buscado:
            return medio
        elif lista[meio] <
nombre_buscado:
            inicio=medio+1
        elif lista[medio] >
nombre_buscado:
            fin = medio-1

    return -1

def main():
    lista_de_alumnos =
sorted(importar_lista('../data/lista_alum
nos'))
    for i in range(0,3500):
        posicion_del_alumno =
buscar(lista_de_alumnos, "Zoraida")
        print("Aluno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

En primer lugar, usamos la función **sorted()** de Python para devolver la lista en orden alfabético. A partir de ahí, empezamos a buscar a la alumna Zoraida, que está al final de la lista, para simular el peor de los casos.

Refactorizamos la función de buscar para utilizar la búsqueda binaria, que consiste en comparar el valor buscado con el valor del elemento en el medio de la lista y, si son iguales, se devuelve la posición media.

```
if lista[medio] == nombre_buscado:
            return medio
```

Si el valor buscado precede al del medio, el algoritmo descarta todos los valores subsiguientes.

```
elif lista[medio] > nombre_buscado:
            fim = medio-1
```

Y si el valor buscado está después del valor medio, el algoritmo descarta todos los valores anteriores, hasta que solo quede el elemento deseado. Si el elemento restante no es lo que queremos, se devuelve un valor negativo.

```
elif lista[medio] < nombre_buscado:
            inicio=medio+1
```

Ahora bien, al realizar la simulación de las 3500 solicitudes, nos damos cuenta de que la ejecución es casi instantánea. ¿Por qué este algoritmo funciona tan rápido? Analicémoslo.

Comparación de la eficiencia de los algoritmos

Considerando, una vez más, que tenemos 85.000 alumnos y que en cada iteración de búsqueda se descarta la mitad de la lista que no es la que buscamos, podemos calcular mediante logaritmos en base 2 y concluir que con cada petición del función de búsqueda, en el peor de los casos, se realizan operaciones **lg(N)**, es decir,

lg(85000) ≃ 16 operaciones.

Pero aún no ha terminado, ya que hicimos 3500 llamadas a esta función, por lo que realizamos **(3500 * 16) ≃ 56000** operaciones.

Así, pudimos optimizar nuestro algoritmo que antes realizaba casi 300 millones de operaciones a solo 56 mil.

Una vez más hemos comprobado la importancia de pensar en cómo se comportarán nuestros algoritmos de acuerdo a la cantidad de datos recibidos, y vale la pena recalcar que aunque existan abstracciones de tales funciones, como la función **index()** que realizaría esta búsqueda con solo una llamada, es muy importante aprender estos fundamentos.

Importante recalcar que para la solución de búsqueda binaria es necesario tener la lista ordenada, por lo que usamos la función **sorted()** para ordenarla.