Análisis de la eficiencia algorítmica en búsquedas en Pyhton

Programacion I

Alumno:

Tempio Diaz, Matias.(comisión 22)

Vera, Pablo Martin. (comisión 23)

Profesor tutor:

Gubiotti, Flor Camila (comision23)

Indice

Introducción

Uno de los objetivos en el desarrollo de un algoritmo es mantener lo más bajo posible el consumo de recursos, aprovechándolos de la mejor manera, sin desperdiciarlos.

En esta investigación se pretende medir la eficiencia de un algoritmo, es decir, medir la cantidad de recursos necesarios para su ejecución.

El análisis de la eficiencia de los algoritmos presenta un problema, que consiste en que los recursos consumidos dependen de varios factores externos, y se mencionan a continuación: la computadora en la que se ejecute, la calidad del código generado por el compilador y el tamaño de los datos de entrada. Y no se debe dejar pasar la importante diferencia entre estas variables externas: al cambiar de computadora o compilador, la velocidad de ejecución y la memoria usada pueden variar de una forma constante, es decir, una maquina puede resultar diez veces más rápida que otra, o un compilador generar código tres veces más lento. En cambio, la influencia del tamaño de los datos de entrada presenta por lo general otro comportamiento. Para muchos algoritmos ocurre que al recibir datos que son unas unidades más grandes, el uso de recursos se incrementa de una manera que se duplica o se eleva al cuadrado.

En esta investigación se contrastará el método de búsqueda lineal con respecto al método de búsqueda binaria, sobre una lista de calificaciones ficticia de alumnos de la Tecnicatura en Programación a distancia que pertenecen a la Universidad Tecnológica Nacional.

Marco teórico

Para continuar, primero se debe comprender qué es un algoritmo. A lo que se puede responder que un algoritmo es un procedimiento computacional que toma un valor o un conjunto de valores como una entrada, y produce algún otro valor o conjunto de valores como salidas.

Por lo que se puede afirmar también que un algoritmo es una secuencia de pasos computacionales que transforman un dato de entrada en un dato o conjunto de datos de salida.

Y por otro lado, se puede definir al algoritmo como una herramienta para resolver un problema computacional específico.

Ahora bien, un algoritmo es más eficiente que otro si realiza las mismas tareas con menos recursos. Para justificar que un algoritmo es ineficiente debemos proporcionar otro más eficiente que él, y al mismo tiempo se debe argumentar que no existe una mejor manera de desarrollar la misma tarea.

El análisis de los algoritmos es una herramienta que los compara para hacer la evaluación de su diseño, para establecer su calidad y su eficiencia. Al comprender las diferencias que resultan de un análisis comparativo, es posible tomar decisiones más acertadas para desarrollar el algoritmo más adecuado y que refleje mejoras sustanciales en eficiencia y rendimiento, especialmente cuando se manejan grandes volúmenes de datos o se requieren respuestas rápidas.

Como ya se mencionó, en esta investigación se realizara el análisis de algoritmos de búsqueda. Por lo que se debe tener en cuenta que un algoritmo de búsqueda es una secuencia de instrucciones y reglas definidas que nos permiten encontrar un elemento de interés específico dentro de una estructura de datos como pueden ser listas, arreglos, árboles o bases de datos. Son de suma importancia dentro de las ciencias de la computación, existiendo diversas técnicas de búsqueda que se adaptan dependiendo de la naturaleza de los datos y el objetivo específico. En esta oportunidad, se desarrollaran dos algoritmos clásicos caracterizados por su sencillez y aplicabilidad: la búsqueda lineal y la búsqueda binaria.

En primer lugar, la búsqueda lineal, también conocida como búsqueda secuencial, es un método que consiste en recorrer una lista de elementos uno por uno, desde el principio hasta encontrar el objetivo buscado o hasta llegar al final. Resulta más sencillo de comprender e implementar, es eficiente para listas de un tamaño pequeño y no requiere que los elementos de la lista se encuentren previamente ordenados.

Podemos encontrarnos con distintos panoramas, por ejemplo, el mejor caso resultaría ser cuando el valor buscado es igual al primer elemento de la lista, en este caso solo se necesitaría una comparación. En el otro extremo, sería

cuando la posición del elemento rastreado es la última dentro del vector, o incluso peor aún, cuando el elemento no se encuentra dentro de la lista. En estos casos se debería recorrer la totalidad de la longitud del arreglo.

En segundo lugar, la búsqueda binaria, también conocida como búsqueda dicotómica, puede considerarse un método más eficiente, pero tiene como condición obligatoria que la lista a tratar posea sus elementos ordenados, ya sea de manera ascendente o descendente. Este método consiste en comparar el elemento de interés con el elemento que se encuentra en la posición del medio de la lista. Si no hay coincidencia se descarta una de las mitades del conjunto de datos y se vuelve a repetir el proceso, de forma iterativa o recursiva, sobre la sub-lista correspondiente, haciendo uso de la estrategia “divide y vencerás”.

Continuando con el análisis de algoritmos y la búsqueda de la eficiencia computacional de los mismos, es de suma importancia tener en consideración los conceptos de complejidad temporal y espacial, las cuales constituyen herramientas fundamentales a la hora de evaluar el rendimiento, y nos permiten estimar de una manera teórica, los recursos demandados por un algoritmo dependiendo del tamaño de los datos de entrada.

Es así, que se puede definir como complejidad temporal al tiempo de procesamiento que demanda la ejecución de un algoritmo, que no es expresada en unidad de tiempo específica, si no que se refiere al número de operaciones elementales que son necesarias al momento de la ejecución. Se expresa utilizando la notación Big-O, que describe el comportamiento del algoritmo, principalmente, en el peor caso, es decir cuando el tamaño de los datos de entrada tiende a infinito.

En cuanto a complejidad espacial, se hace referencia a la cantidad de memoria requerida por el algoritmo al momento de resolver el problema y de cómo incrementa esa demanda de espacio de memoria al extenderse el volumen de entradas. De igual forma que la complejidad temporal, también es posible expresarla a través de la notación Big-O.

Esta herramienta, como fue citado anteriormente, es útil para elección de mejores soluciones, con la característica de ser independiente del lenguaje de programación que se utilice y del hardware en donde se ejecute el algoritmo. Posee las siguientes notaciones:

- $O(1)$ tiempo constante
- $O(\log_n)$ tiempo logarítmico
- $O(n)$ tiempo lineal
- $O(n \log_n)$ tiempo lineal logarítmico
- $O(n^2)$ tiempo cuadrático
- $O(2^n)$ tiempo exponencial

Caso práctico

Se realiza una búsqueda lineal y una búsqueda binaria, sobre una lista ordenada, de calificaciones ficticias que obtuvieron los alumnos de la Tecnicatura en programación a distancia, de la Universidad Tecnológica Nacional. Los valores buscados dentro de la lista de calificaciones son las notas iguales a 10. A continuación, se presenta el código desarrollado en Python.

integrador.py

C: > Users > BANGHO > Desktop > integrador.py > busqueda

```python
# se importa el modulo random para generar aleatoriamente notas
import random

#se importa el modulo time para poder calcular la demora en la busqueda
import time

#se genera una lista de notas del 1 al 10
def alumnos(cant):
    lista_notas=[round(random.uniform(1, 10), 2) for i in range(cant)]

    lista_notas= sorted(lista_notas)
    return lista_notas# la funcion retorna una lista ordenada para que ambas busquedas esten en iguales condiciones

#se busca la calificacion objetivo de forma lineal y se retorna el tiempo de demora en encontrar el objetivo
def busqueda(list):
    inicio= time.time()
    for i in range (len(list)):
        if list[i] == 10:
            fin= time.time()
    demora= (fin - inicio)*1000
    return demora


def busqueda_binaria(list):
    #se divide la lista en dos mitades para ir comparando
    izquierda = 0
    derecha = len(list) - 1
    resultado= -1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if list[medio] >9:
            resultado= medio
            derecha= medio-1
        else:
            izquierda= medio+1
    #si no se encuentran resultados se devuelve una lista vacia
    if resultado == -1:
        return []

    return list[resultado:]


#funcion que calcula la demora de busqueda binaria
def demora_binaria(list):
    inicio= time.time()
    busqueda_binaria(list)
    fin= time.time()
    demora= (fin-inicio)*1000
    return demora


#se imprimen los valores obtenidos con la busqueda lineal
print(f"El tiempo de busqueda lineal es de {busqueda(alumnos(100000)):6f} milisegundos para 100000 alumnos ")
print(f"El tiempo de busqueda lineal es de {busqueda(alumnos(400000)):6f} milisegundos para 400000 alumnos ")
print(f"El tiempo de busqueda lineal es de {busqueda(alumnos(800000)):6f} milisegundos para 800000 alumnos ")
print(f"El tiempo de busqueda lineal es de {busqueda(alumnos(1200000)):6f} milisegundos para 1200000 alumnos ")
print(f"El tiempo de busqueda lineal es de {busqueda(alumnos(1600000)):6f} milisegundos para 1600000 alumnos ")

#se imprimen los valores obtenidos con la busqueda lineal
print(f"El tiempo de busqueda binaria es de {demora_binaria(alumnos(100000)):6f} milisegundos para 100000 alumnos")
print(f"El tiempo de busqueda binaria es de {demora_binaria(alumnos(400000)):6f} milisegundos para 400000 alumnos")
print(f"El tiempo de busqueda binaria es de {demora_binaria(alumnos(800000)):6f} milisegundos 800000 alumnos")
print(f"El tiempo de busqueda binaria es de {demora_binaria(alumnos(1200000)):6f} milisegundos 1200000 alumnos")
print(f"El tiempo de busqueda binaria es de {demora_binaria(alumnos(1600000)):6f} milisegundos 1600000 alumnos")
```

Metodología utilizada

En primer lugar, se investigó sobre los conceptos fundamentales relacionados con la temática desarrollada, consultando bibliografía disponible para obtener información confiable de respaldo.

En segundo lugar, se desarrolló el código en lenguaje Python, el cual fue dictado y ejercitado durante el transcurso del cuatrimestre actual dentro de la materia Programación I.

La lista utilizada fue generada con notas aleatorias, mediante el método *random.uniform( )*. Luego, mediante la función *sorted( )* se ordenaron sus elementos en forma ascendente.

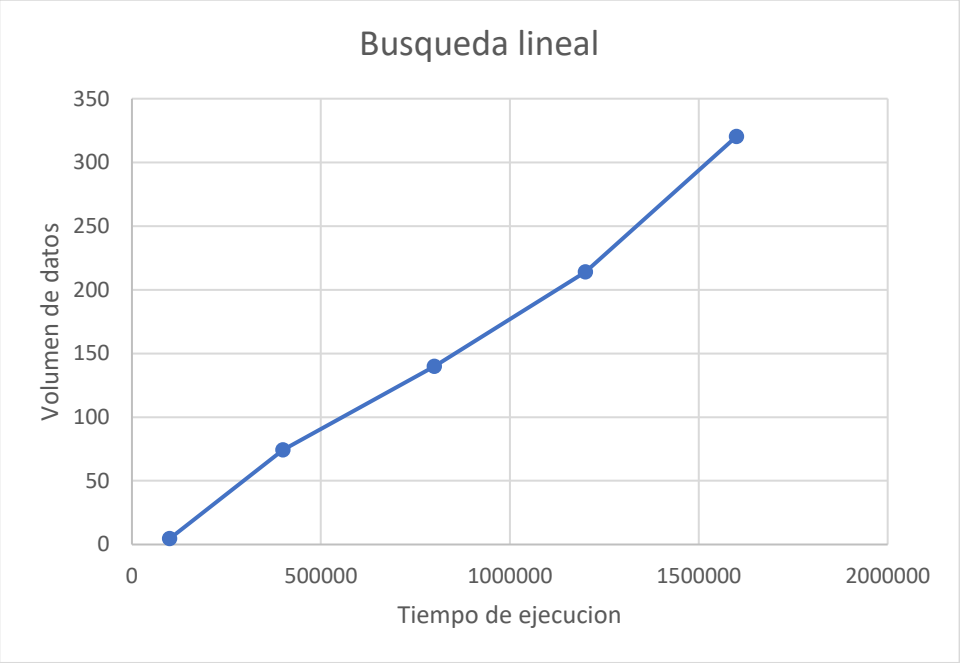
Se realizaron las búsquedas, bajo las dos metodologías ya mencionadas, y se midió el tiempo de ejecución mediante la importación del modulo *time,* obteniendo así la demora medida en milisegundos.

El entorno de desarrollo itegrado utilizado fue Visual Studio Code, y el control de versiones fue GIT, para luego alojar el repositorio remoto dentro de la plataforma GitHub.

Resultados obtenidos

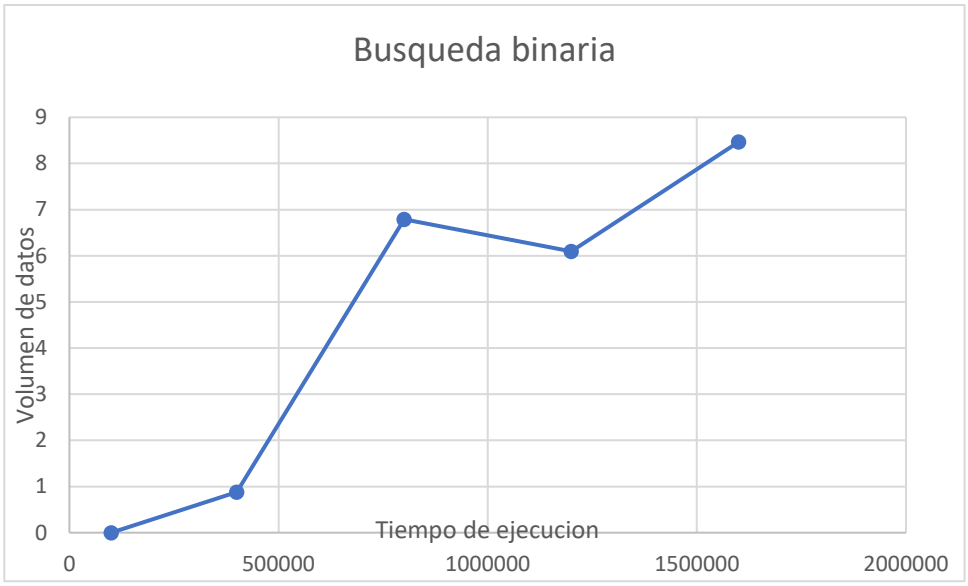
Al realizar el análisis empírico propuesto, se tuvo en cuenta que ambas funciones busquen el mismo valor objetivo, y que se trabaje con exactamente la misma lista de datos ordenada.

Se repitieron las mediciones en varias oportunidades, para sacar un promedio y evitar distorsiones por azar. Se volcaron los datos obtenidos en una planilla de cálculo Excel y se obtuvo un gráfico representativo de los valores obtenidos para cada modalidad de búsqueda.



En el eje vertical está representado el tamaño de la estructura de dato con la que se trabajo y en el eje horizontal se representa el tiempo de ejecución medido en milisegundos

Se puede observar que a medida que aumenta el tamaño de la lista en donde se realiza la búsqueda lineal, los tiempos de ejecución también se incrementan, de manera proporcional, correspondiéndose a una complejidad temporal en notación Big O de O(n).



En este grafico se puede observar que al principio la curva asciende abruptamente a medida que el volumen de datos se incrementa y a partir de un punto se desacelera generandose una especie de meseta. El tiempo de ejecución no depende un 100% del tamaño constante de la entrada ya que en cada iteración éste disminuye a la mitad, asemejándose a una complejidad temporal en notacion Big O de $O(\log_n)$.

A continuación compartimos el enlace al repositorio en GitHub:

https://github.com/Pablomartinv/Integrador_Programacion1.git

Conclusión

En resumen, el análisis de la complejidad temporal de un algoritmo se centra en el tamaño de la entrada (n) y en cómo varía el comportamiento del algoritmo a medida que cambia esta entrada. La importancia de realizar un análisis de algoritmos nos permite determinar la eficiencia de un algoritmo sobre otro, visualizar soluciones alternativas, seleccionar el algoritmo más apropiado según el objetivo y contexto para diseñar programas más adecuados y escalables.

También podemos mencionar que este trabajo resulto de mucha utilidad para afianzar conocimientos adquiridos durante todo el cuatrimestre cursado correspondiente a la materia Programación I, integrando el desarrollo de estructuras de datos, métodos de búsqueda, funciones, estructuras condicionales y repetitivas, entre otros.

Bibliografía

- Python Software Foundation- Python 3.13.5 documentation.

- Apuntes sobre el calculo de la eficiencia de los algoritmos- José L. Balcazar.

- MANUAL DE ANÁLISIS Y DISEÑO DE ALGORITMOS- Víctor Valenzuela Ruz. Docente Ingeniería en Gestión Informática INACAP Copiapó.

- https://www.w3schools.com/python/python_reference.asp

Anexos

Enlace al video explicativo en youtube.com:

https://www.youtube.com/watch?v=6Xxp3VsnQUE