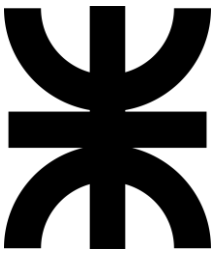
Universidad Tecnológica Nacional

Facultad Regional San Nicolás

Programación I



Trabajo Integrador

“Análisis de eficiencia de algoritmos utilizando estructuras de datos complejos y recursividad”

Comisión 20

Integrantes:

Juan Pablo Ribero Mazzoni - juampiribero@gmail.com

Raul Alberto Robino - raulrobino@gmail.com

Profesor: Nicolás Quirós

Tecnicatura Universitaria en Programación a Distancia

2025

Introducción

Para poder lograr el desarrollo de software de manera eficiente y escalable es necesaria una buena comprensión de cómo se funcionan los algoritmos y las estructuras de datos con el tamaño de los datos de entrada.

El presente trabajo práctico explora el análisis de algoritmos, haciendo foco en la eficiencia temporal y su aplicación práctica al trabajar con estructuras de datos complejas como por ejemplo listas, diccionarios y conjuntos.

También se investigará el impacto puede tener la recursividad en el rendimiento, en comparación con las soluciones iterativas.

Marco teórico

Se puede definir al análisis de algoritmos como el estudio sistemático del rendimiento de los algoritmos, los cuales son un conjunto de instrucciones precisas y ordenadas que permiten resolver un problema específico. Dicho análisis tiene los siguientes objetivos:

- Eficiencia temporal, que es el tiempo que demora la ejecución de un algoritmo en función del tamaño de la entrada.
- Eficiencia espacial, que es la cantidad de memoria que demanda un algoritmo en función del tamaño de la entrada.

Analizar los algoritmos es importante porque:

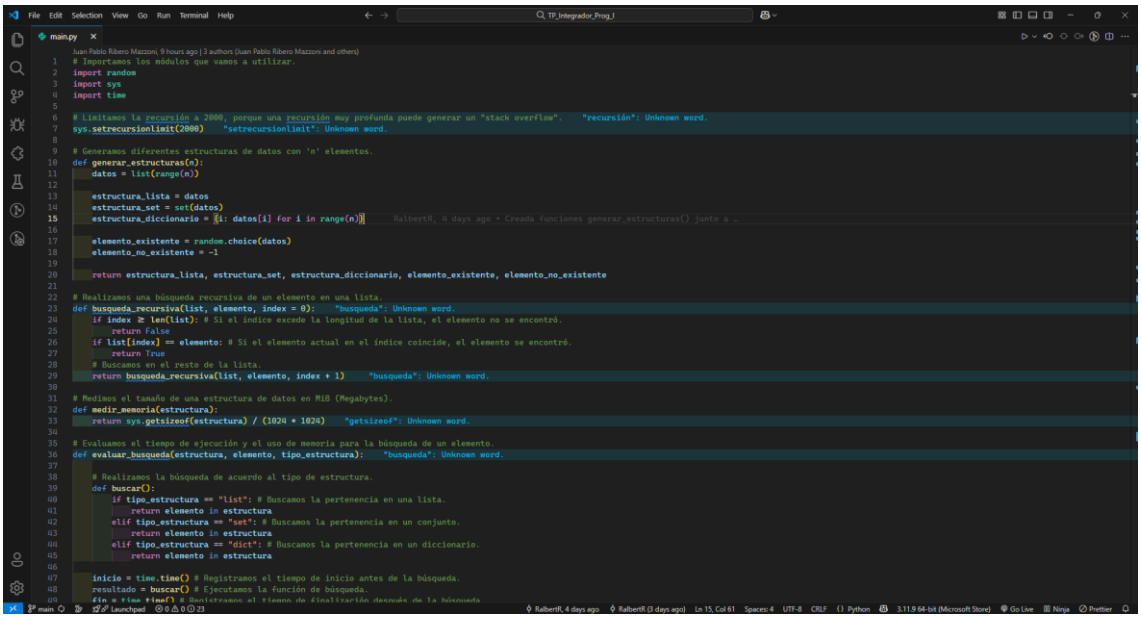
- Permite comparar la eficiencia de diferentes soluciones para un mismo problema.
- La optimización del uso de memoria y el tiempo de ejecución son cruciales para el rendimiento de las aplicaciones.
- Ayuda a elegir el algoritmo más adecuado para cada situación, garantizando la eficiencia y escalabilidad de nuestras soluciones.

Algunos de los conceptos claves que se pueden mencionar son:

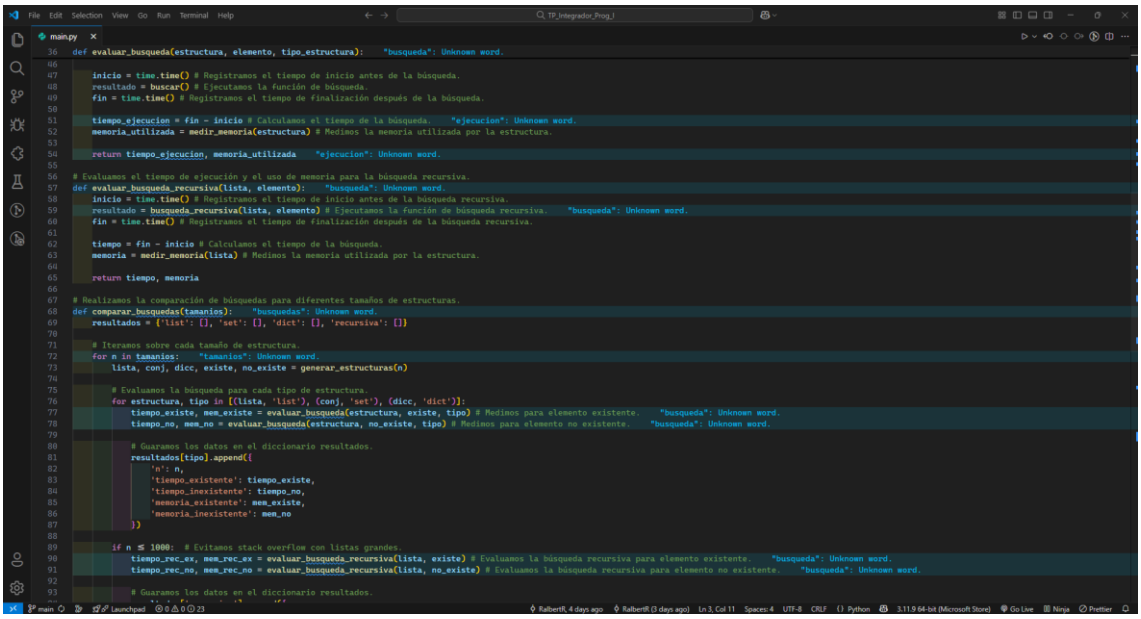
- Análisis empírico, mide el tiempo en el que se ejecuta un algoritmo. Además, permite observar el comportamiento práctico en diferentes escenarios y con distintos tamaños de entradas.
- Análisis teórico: determina la complejidad de un algoritmo analizando su código, identificando las operaciones dominantes y expresando su rendimiento con la notación Big O.
- Notación Big O (O(n)), es una forma de clasificar los algoritmos de acuerdo a cómo crece el tiempo de ejecución o espacio requerido a medida que el tamaño de la entrada (n) aumenta. Describe el peor caso del crecimiento, ignorando constantes y términos de menor orden.
- Estructuras de datos complejas:
    - Listas (list), son colecciones ordenadas y mutables de ítems. Se asemeja a un arreglo dinámico.
    - Diccionarios (dict), son colecciones de pares clave-valor, no son ordenadas y mutables. Estos ofrecen búsquedas muy rápidas.
    - Conjuntos (set), son colecciones no ordenadas de ítems únicos y mutables. Estos son eficientes para operaciones de membresía y eliminación de duplicados.
- Recursividad, es una técnica de programación donde una función se llama a sí misma para resolver un problema, dividiéndolo en problemas más pequeños del mismo tipo. Es muy importante entender su impacto en el uso de la memoria y el tiempo de ejecución.

Caso práctico

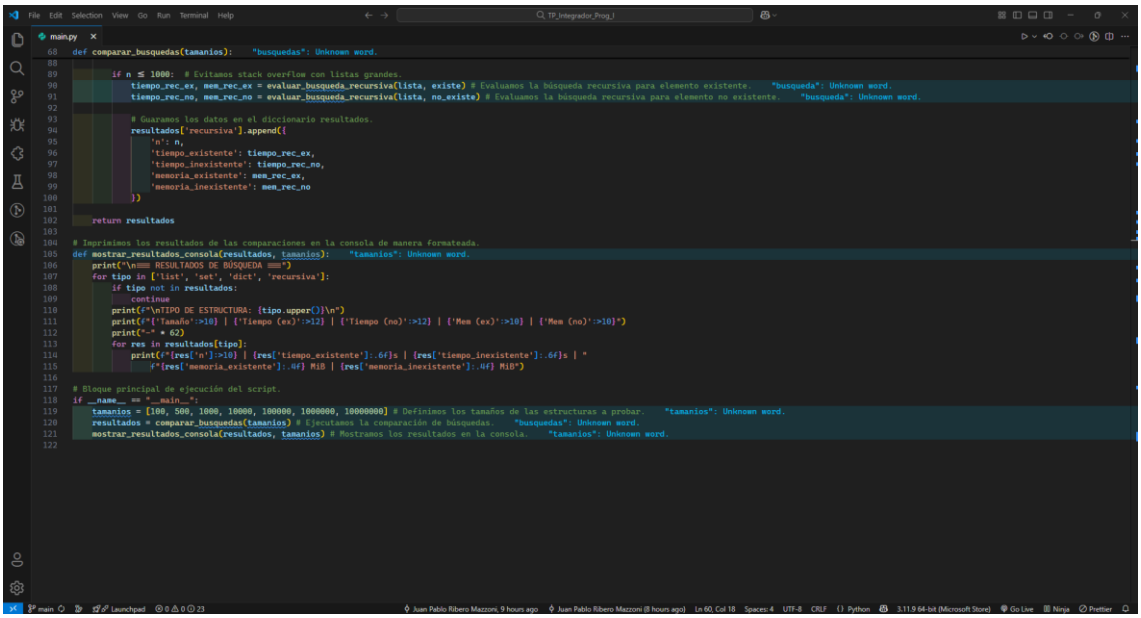
Repositorio → **https://github.com/JuampiRibero/TP_Integrador_Prog_I**



```python
# Importamos los módulos que vamos a utilizar.
import random
import sys
import time

# Limitamos la recursión a 2000, porque una recursión muy profunda puede generar un "stack overflow".
sys.setrecursionlimit(2000)

# Generamos diferentes estructuras de datos con 'n' elementos.
def generar_estructuras(n):
    datos = list(range(n))

    estructura_lista = datos
    estructura_set = set(datos)
    estructura_diccionario = {i: datos[i] for i in range(n)}

    elemento_existente = random.choice(datos)
    elemento_no_existente = -1

    return estructura_lista, estructura_set, estructura_diccionario, elemento_existente, elemento_no_existente

# Realizamos una búsqueda recursiva de un elemento en una lista.
def busqueda_recursiva(list, elemento, index = 0):
    if index >= len(list): # Si el índice excede la longitud de la lista, el elemento no se encontró.
        return False
    if list[index] == elemento: # Si el elemento actual en el índice coincide, el elemento se encontró.
        return True
    # Buscamos en el resto de la lista.
    return busqueda_recursiva(list, elemento, index + 1)

# Medimos el tamaño de una estructura de datos en MiB (Megabytes).
def medir_memoria(estructura):
    return sys.getsizeof(estructura) / (1024 * 1024)

# Evaluamos el tiempo de ejecución y el uso de memoria para la búsqueda de un elemento.
def evaluar_busqueda(estructura, elemento, tipo_estructura):

    # Realizamos la búsqueda de acuerdo al tipo de estructura.
    def buscar():
        if tipo_estructura == "list": # Buscamos la pertenencia en una lista.
            return elemento in estructura
        elif tipo_estructura == "set": # Buscamos la pertenencia en un conjunto.
            return elemento in estructura
        elif tipo_estructura == "dict": # Buscamos la pertenencia en un diccionario.
            return elemento in estructura

    inicio = time.time() # Registramos el tiempo de inicio antes de la búsqueda.
    resultado = buscar() # Ejecutamos la función de búsqueda.
```



```python
def evaluar_busqueda(estructura, elemento, tipo_estructura):

    inicio = time.time() # Registramos el tiempo de inicio antes de la búsqueda.
    resultado = buscar() # Ejecutamos la función de búsqueda.
    fin = time.time() # Registramos el tiempo de finalización después de la búsqueda.

    tiempo_ejecucion = fin - inicio # Calculamos el tiempo de la búsqueda.
    memoria_utilizada = medir_memoria(estructura) # Medimos la memoria utilizada por la estructura.

    return tiempo_ejecucion, memoria_utilizada

# Evaluamos el tiempo de ejecución y el uso de memoria para la búsqueda recursiva.
def evaluar_busqueda_recursiva(lista, elemento):
    inicio = time.time() # Registramos el tiempo de inicio antes de la búsqueda recursiva.
    resultado = busqueda_recursiva(lista, elemento) # Ejecutamos la función de búsqueda recursiva.
    fin = time.time() # Registramos el tiempo de finalización después de la búsqueda recursiva.

    tiempo = fin - inicio # Calculamos el tiempo de la búsqueda.
    memoria = medir_memoria(lista) # Medimos la memoria utilizada por la estructura.

    return tiempo, memoria

# Realizamos la comparación de búsquedas para diferentes tamaños de estructuras.
def comparar_busquedas(tamanios):
    resultados = {'list': [], 'set': [], 'dict': [], 'recursiva': []}

    # Iteramos sobre cada tamaño de estructura.
    for n in tamanios:
        lista, conj, dicc, existe, no_existe = generar_estructuras(n)

        # Evaluamos la búsqueda para cada tipo de estructura.
        for estructura, tipo in [(lista, 'list'), (conj, 'set'), (dicc, 'dict')]:
            tiempo_existe, mem_existe = evaluar_busqueda(estructura, existe, tipo) # Medimos para elemento existente.
            tiempo_no, mem_no = evaluar_busqueda(estructura, no_existe, tipo) # Medimos para elemento no existente.

            # Guaramos los datos en el diccionario resultados.
            resultados[tipo].append({
                'n': n,
                'tiempo_existente': tiempo_existe,
                'tiempo_inexistente': tiempo_no,
                'memoria_existente': mem_existe,
                'memoria_inexistente': mem_no
            })

        if n <= 1000:  # Evitamos stack overflow con listas grandes.
            tiempo_rec_ex, mem_rec_ex = evaluar_busqueda_recursiva(lista, existe) # Evaluamos la búsqueda recursiva para elemento existente.
            tiempo_rec_no, mem_rec_no = evaluar_busqueda_recursiva(lista, no_existe) # Evaluamos la búsqueda recursiva para elemento no existente.

            # Guaramos los datos en el diccionario resultados.
```



```python
def comparar_busquedas(tamanios):    "busquedas": Unknown word.

        if n ≤ 1000:  # Evitamos stack overflow con listas grandes.
            tiempo_rec_ex, mem_rec_ex = evaluar_busqueda_recursiva(lista, existe) # Evaluamos la búsqueda recursiva para elemento existente.    "busqueda": Unknown word.
            tiempo_rec_no, mem_rec_no = evaluar_busqueda_recursiva(lista, no_existe) # Evaluamos la búsqueda recursiva para elemento no existente.    "busqueda": Unknown word.

            # Guaramos los datos en el diccionario resultados.
            resultados['recursiva'].append({
                'n': n,
                'tiempo_existente': tiempo_rec_ex,
                'tiempo_inexistente': tiempo_rec_no,
                'memoria_existente': mem_rec_ex,
                'memoria_inexistente': mem_rec_no
            })

    return resultados

# Imprimimos los resultados de las comparaciones en la consola de manera formateada.
def mostrar_resultados_consola(resultados, tamanios):    "tamanios": Unknown word.
    print("\n=== RESULTADOS DE BÚSQUEDA ===")
    for tipo in ['list', 'set', 'dict', 'recursiva']:
        if tipo not in resultados:
            continue
        print(f"\nTIPO DE ESTRUCTURA: {tipo.upper()}\n")
        print(f"{'Tamaño':>10} | {'Tiempo (ex)':>12} | {'Tiempo (no)':>12} | {'Mem (ex)':>10} | {'Mem (no)':>10}")
        print("-" * 62)
        for res in resultados[tipo]:
            print(f"{res['n']:>10} | {res['tiempo_existente']:.6f}s | {res['tiempo_inexistente']:.6f}s | "
                  f"{res['memoria_existente']:.4f} MiB | {res['memoria_inexistente']:.4f} MiB")

# Bloque principal de ejecución del script.
if __name__ == "__main__":
    tamanios = [100, 500, 1000, 10000, 100000, 1000000, 10000000] # Definimos los tamaños de las estructuras a probar.    "tamanios": Unknown word.
    resultados = comparar_busquedas(tamanios) # Ejecutamos la comparación de búsquedas.    "busquedas": Unknown word.
    mostrar_resultados_consola(resultados, tamanios) # Mostramos los resultados en la consola.    "tamanios": Unknown word.
```

```
$ python3 main.py

=== RESULTADOS DE BÚSQUEDA ===

TIPO DE ESTRUCTURA: LIST

    Tamaño |  Tiempo (ex) |  Tiempo (no) |   Mem (ex) |   Mem (no)
--------------------------------------------------------------------
       100 | 0.000000s | 0.000000s | 0.0008 MiB | 0.0008 MiB
       500 | 0.000000s | 0.000000s | 0.0039 MiB | 0.0039 MiB
      1000 | 0.000000s | 0.000000s | 0.0077 MiB | 0.0077 MiB
     10000 | 0.000000s | 0.000000s | 0.0763 MiB | 0.0763 MiB
    100000 | 0.000000s | 0.001001s | 0.7630 MiB | 0.7630 MiB
   1000000 | 0.000999s | 0.003999s | 7.6294 MiB | 7.6294 MiB
  10000000 | 0.031992s | 0.036014s | 76.2940 MiB | 76.2940 MiB

TIPO DE ESTRUCTURA: SET

    Tamaño |  Tiempo (ex) |  Tiempo (no) |   Mem (ex) |   Mem (no)
--------------------------------------------------------------------
       100 | 0.000000s | 0.000000s | 0.0080 MiB | 0.0080 MiB
       500 | 0.000000s | 0.000000s | 0.0315 MiB | 0.0315 MiB
      1000 | 0.000000s | 0.000000s | 0.0315 MiB | 0.0315 MiB
     10000 | 0.000000s | 0.000000s | 0.5002 MiB | 0.5002 MiB
    100000 | 0.000000s | 0.000000s | 4.0002 MiB | 4.0002 MiB
   1000000 | 0.000000s | 0.000000s | 32.0002 MiB | 32.0002 MiB
  10000000 | 0.000000s | 0.000000s | 256.0002 MiB | 256.0002 MiB

TIPO DE ESTRUCTURA: DICT

    Tamaño |  Tiempo (ex) |  Tiempo (no) |   Mem (ex) |   Mem (no)
--------------------------------------------------------------------
       100 | 0.000000s | 0.000000s | 0.0045 MiB | 0.0045 MiB
       500 | 0.000000s | 0.000000s | 0.0177 MiB | 0.0177 MiB
      1000 | 0.000000s | 0.000000s | 0.0352 MiB | 0.0352 MiB
     10000 | 0.000000s | 0.000000s | 0.2813 MiB | 0.2813 MiB
    100000 | 0.000000s | 0.000000s | 5.0001 MiB | 5.0001 MiB
   1000000 | 0.000000s | 0.000000s | 40.0001 MiB | 40.0001 MiB
  10000000 | 0.000000s | 0.000000s | 320.0001 MiB | 320.0001 MiB

TIPO DE ESTRUCTURA: RECURSIVA

    Tamaño |  Tiempo (ex) |  Tiempo (no) |   Mem (ex) |   Mem (no)
--------------------------------------------------------------------
       100 | 0.000000s | 0.000000s | 0.0008 MiB | 0.0008 MiB
       500 | 0.000000s | 0.000000s | 0.0039 MiB | 0.0039 MiB
      1000 | 0.000000s | 0.000000s | 0.0077 MiB | 0.0077 MiB
```

Metodología utilizada

Para realizar el análisis empírico, se siguieron los siguientes pasos:

1. Generación de estructuras: se definieron tres tipos de estructuras de datos (una lista, un conjunto y un diccionario) con tamaños crecientes de entradas (10, 100, 500, 1000, 10.000 y 100.000).
2. Selección de elementos de búsqueda: se eligieron dos tipos de elementos para poder evaluar la eficiencia de búsqueda.
    - Uno donde el elemento buscado existe dentro de la estructura (numero seleccionado de manera aleatoria desde los datos generados).
    - Otro donde el valor buscado no se encuentra dentro de la estructura (valor int -1, que no se encuentra en ninguna de las estructuras).
3. Medición del rendimiento: se calculó el tiempo de ejecución de la búsqueda utilizando la función time.time(). También se cuantificó la memoria utilizada para almacenar el resultado de la búsqueda con sys.getsizeof().
4. Evaluación uniforme: para garantizar una comparación equitativa entre estructuras, se utilizó un enfoque de búsqueda uniforme (elemento in estructura), que es la manera idiomática en Python y muestra cómo cada estructura optimiza internamente este proceso.
5. Repetición para distintos tamaños: se repitió el proceso para cada tamaño de estructura, registrando tanto el tiempo como la memoria utilizada.

Resultados obtenidos

Los pruebas dieron los siguientes resultados:

- Tiempo de búsqueda:
    - La lista mostró un crecimiento casi lineal en el tiempo de búsqueda, debido a que recorre todos los elementos.

    - Tanto el conjunto como el diccionario mostraron tiempos de búsqueda bajos y casi constantes, sin importar el tamaño de la estructura, gracias a su implementación basada en tablas hash.
- Uso de memoria:
    - El diccionario fue la estructura con más consumo de memoria debido al almacenamiento en pares clave-valor.
    - El conjunto utilizó más memoria que la lista, pero menos que el diccionario.
    - La lista fue la estructura más eficiente en consumo de memoria, pero con el peor rendimiento en tiempo de búsqueda.

Conclusiones

Las estructuras de datos no solo determinan cómo se guardan los elementos, sino que también tienen un impacto significativo en la eficiencia de los algoritmos que se utilizan con ellas. Aunque las listas son simples y utilizan menos memoria, no son ideales para realizar búsquedas en grandes volúmenes de datos.

Los conjuntos (set) y los diccionarios (dict) son pueden ser los indicados cuando se necesita rapidez en las búsquedas, porque su rendimiento se mantiene constante incluso con una gran cantidad de información.

Este caso práctico permitió experimentar y medir de manera empírica la eficiencia algorítmica en Python, reforzando así la importancia que tiene la selección de la estructura de datos adecuada dependiendo del contexto del problema.

Bibliografía

- Python Software Foundation. (n.d.). time — Time access and conversions. En Python 3.x Documentation. https://docs.python.org/3/library/time.html
- Python Software Foundation. (n.d.). Big O Cheat Sheet. https://www.bigocheatsheet.com/
- Python Software Foundation. (n.d.). Python Data Structures. En The Python Tutorial. https://docs.python.org/3/tutorial/datastructures.html

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4.ª ed.).
- Bhargava, A. (2016). Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People. Manning Publications.

Anexos

- Capturas de pantallas de resultados de la ejecución del algoritmo (por ejemplo, imagen de la terminal mostrando los tiempos de ejecución para los diferentes tamaños de entradas).
- Enlace al repositorio del proyecto en GitHub: https://github.com/JuampiRibero/TP_Integrador_Prog_I (código y un README.md describiendo el proyecto, cómo ejecutarlo, y los resultados principales).
- Enlace a la video presentación: https://www.youtube.com/watch?v=0QvqdDkETm0