

TRABAJO PRÁCTICO

INTEGRADOR

PROGRAMACIÓN I

Título: Análisis de algoritmos aplicados a la organización de tareas domésticas vía código en Python

Tema: *Análisis de Algoritmos*

Alumnos:

Augusto M. Cúneo Brouwer de Koning – correo: augusto_cuneo@hotmail.com

Matías Deluca – correo: matiasdeluca2000@gmail.com

Profesor/a: Julieta Trapé - **Tutor/a:** Miguel Barrera Oltra

Fecha de Entrega: 09 de Junio del 2025

Índice

1. Introducción.....	Pag. 2
2. Marco Teórico.....	Pag. 3
3. Caso Práctico.....	Pag. 9
4. Metodología Utilizada.....	Pag.11
5. Resultados Obtenidos.....	Pag.12
6. Conclusiones.....	Pag.13
7. Bibliografía.....	Pag.13
8. Anexos.....	Pag.14

Desarrollo

1. Introducción

El estudio de la **eficiencia algorítmica**, o también llamado **análisis de algoritmos**, resulta crucial tanto en el ámbito académico como en la práctica profesional y cotidiana.

¿Por qué se eligió este tema?

Dentro de las múltiples aplicaciones del análisis de algoritmos, la planificación de tareas domésticas constituye un ejemplo cotidiano y fácilmente comprensible. Ordenar actividades según su urgencia refleja el mismo problema que enfrentan los sistemas informáticos cuando priorizan procesos o gestionan colas de tareas/procesos, y elegir el método de organización adecuado repercute directamente en el tiempo total de ejecución. Por ello, el escenario de las tareas del hogar se acopla como hilo conductor que permite trasladar conceptos formales a una situación cotidiana para cualquier persona.

Importancia en programación

En el desarrollo de software, la elección de un algoritmo adecuado es crucial. Un algoritmo ineficiente puede convertir una aplicación funcional en inviable a medida que crece el volumen de datos. Comprender la notación Big-O y realizar comparaciones empíricas de tiempos de ejecución permite a los programadores tomar decisiones fundamentadas y optimizar el uso de recursos como la CPU y la memoria.

Python, por su parte, facilita este análisis gracias a su método de ordenación nativo, basado en Timsort, un algoritmo híbrido de alto rendimiento utilizado en entornos de producción. Esta característica ofrece una valiosa oportunidad para comparar algoritmos clásicos de carácter educativo —como el *Bubble Sort*— con soluciones optimizadas de nivel industrial, permitiendo así una comprensión más profunda del impacto que tienen las decisiones algorítmicas en la eficiencia del software.

Objetivos del trabajo

1. **Investigar** la complejidad temporal y espacial de distintos algoritmos de ordenamiento y búsqueda.
2. **Implementar y medir** en Python la performance de un algoritmo ingenuo de uso industrial (Bubble Sort) frente a uno optimizado de tipo clásico-educativo (Timsort) sobre listas de tareas domésticas de tamaño variable.
3. **Evaluar y documentar** cómo la elección del algoritmo afecta el tiempo de planificación y proponer recomendaciones basadas en la evidencia obtenida.

Este informe, por lo tanto, busca evidenciar que la teoría algorítmica posee un impacto tangible en problemas cotidianos y que el uso de herramientas adecuadas conduce a soluciones escalables y eficientes.

2. Marco Teórico

1. Fundamentos del Análisis de Algoritmos

El análisis de algoritmos es una disciplina fundamental dentro de las ciencias de la computación, que permite estudiar y evaluar la eficiencia de un algoritmo antes de su implementación. La eficiencia se mide principalmente en términos de tiempo de ejecución (complejidad temporal) y uso de memoria (complejidad espacial). Esta evaluación es vital para el desarrollo de software robusto, escalable y optimizado.

En contextos donde los volúmenes de datos pueden crecer exponencialmente, como en sistemas informáticos complejos o incluso en tareas domésticas automatizadas, seleccionar un algoritmo ineficiente puede llevar a tiempos de respuesta inaceptables o a un uso desmedido de recursos. Por ello, el análisis algorítmico no es una mera herramienta académica, sino una necesidad práctica en el diseño de soluciones eficientes.

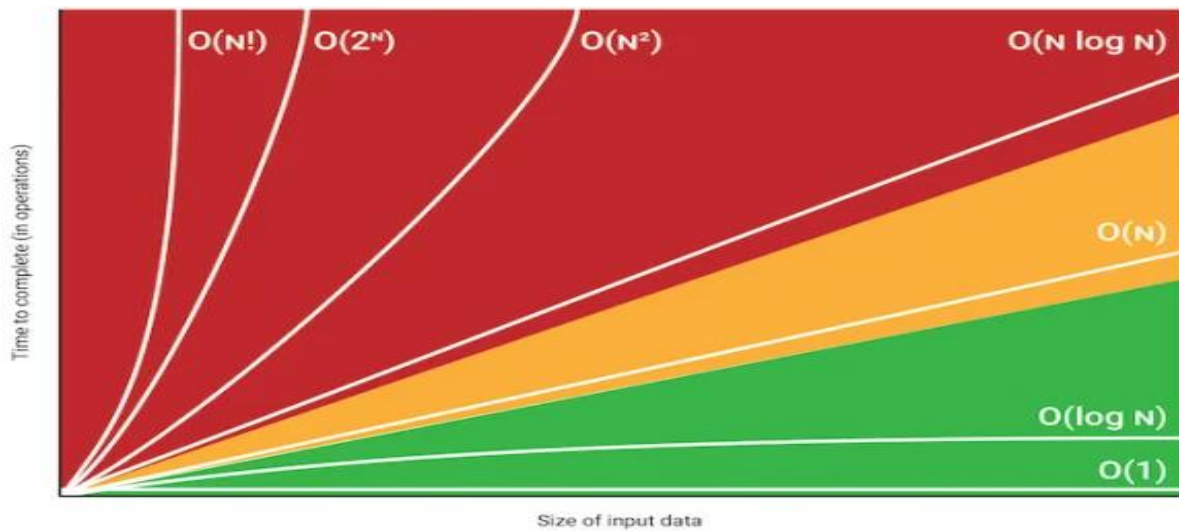
2. Complejidad y Notación Asintótica

La notación Big O se utiliza para expresar cómo crece el tiempo de ejecución (o uso de memoria) de un algoritmo a medida que aumenta el tamaño de la entrada (n). Es una forma de describir el comportamiento asintótico del algoritmo, especialmente útil para comparar soluciones y anticipar su rendimiento a gran escala.

2.1 Tipos de Complejidad Comunes

1. $O(1)$ - Constante: Un algoritmo con complejidad $O(1)$ tiene un tiempo de ejecución constante, lo que significa que el tiempo de ejecución no depende del tamaño de la entrada. Un ejemplo sería el acceso a un elemento en un arreglo: el tiempo que tarda en accederse no cambia, independientemente de cuántos elementos tenga el arreglo.
2. $O(\log n)$ - Logarítmica: Un algoritmo $O(\log n)$ reduce el problema a la mitad con cada operación. Los algoritmos que siguen este patrón suelen ser aquellos que dividen de manera eficiente el espacio de búsqueda. Un ejemplo típico es la búsqueda binaria en una lista ordenada, donde se reduce el número de elementos a evaluar en cada paso.

3. $O(n)$ - Lineal: Un algoritmo con complejidad $O(n)$ crece de manera lineal con el tamaño de la entrada. Esto significa que, a medida que se agrega más información, el tiempo de ejecución aumenta proporcionalmente. Un ejemplo común es un algoritmo que recorre una lista de n elementos, como un recorrido secuencial de todos los elementos de una lista.
4. $O(n \log n)$ - Lineal-logarítmica: Los algoritmos que tienen una complejidad $O(n \log n)$ son eficientes y se encuentran entre los más utilizados para tareas como la ordenación. Un ejemplo de este tipo de algoritmo es MergeSort, que divide y ordena los elementos recursivamente. La parte $\log n$ proviene de la división repetida de los datos, mientras que la parte n proviene del procesamiento de todos los elementos.
5. $O(n^2)$ - Cuadrática: Un algoritmo $O(n^2)$ implica que el tiempo de ejecución aumenta cuadráticamente conforme crece el tamaño de la entrada. Este tipo de complejidad suele aparecer en algoritmos que involucran comparar cada elemento con cada otro. Ejemplos típicos son Bubble Sort y Insertion Sort, donde se realizan múltiples comparaciones e intercambios de elementos.
6. $O(2^n)$ - Exponencial: Los algoritmos con complejidad $O(2^n)$ crecen muy rápidamente. Este tipo de complejidad aparece en algunos algoritmos recursivos, como aquellos que resuelven problemas combinatorios por fuerza bruta, donde el número de operaciones se duplica con cada incremento en el tamaño de la entrada.
7. $O(n!)$ - Factorial: La complejidad $O(n!)$ es la más ineficiente en términos de tiempo de ejecución, ya que el número de operaciones crece extremadamente rápido a medida que aumenta el tamaño de la entrada. Este comportamiento se observa en algoritmos que generan todas las permutaciones posibles de un conjunto de elementos, como los algoritmos de fuerza bruta para el problema del viajante de comercio.



2.2 Notaciones Ω (Omega) y Θ (Theta)

Además de la notación Big O, existen otras notaciones que permiten describir el comportamiento de los algoritmos en distintos escenarios.

Ω (Omega):

La notación Ω (Omega) representa el límite inferior del tiempo de ejecución de un algoritmo. En otras palabras, describe el mejor caso de un algoritmo, es decir, el tiempo mínimo necesario para ejecutar el algoritmo en las mejores condiciones posibles. La notación Ω nos dice cuánto tiempo tomará un algoritmo, como mínimo, en función del tamaño de la entrada. Por ejemplo, en el caso de la búsqueda binaria, la complejidad $\Omega(\log n)$ implica que, en el mejor de los casos, el algoritmo tarda logaritmicamente en encontrar el elemento, incluso si los datos están ordenados de manera óptima.

Θ (Theta):

La notación Θ (Theta) se utiliza para expresar el comportamiento exacto de un algoritmo en cuanto a su tiempo de ejecución. Representa el límite superior e inferior del algoritmo, indicando que el tiempo de ejecución es similar en el mejor y en el peor caso. Es decir, la notación Θ proporciona una descripción más precisa de la complejidad, garantizando que el algoritmo se comportará de manera similar sin importar las condiciones iniciales de los datos. Por ejemplo, MergeSort tiene una complejidad $\Theta(n \log n)$, lo que significa que en

cualquier escenario, el tiempo de ejecución del algoritmo será proporcional a $n \log n$, ya sea en el mejor o peor de los casos.

Estas notaciones proporcionan una forma de analizar los algoritmos no solo desde el punto de vista del peor caso (Big O), sino también del mejor caso (Ω) y el comportamiento general (Θ), permitiendo una evaluación más completa de su rendimiento.

3. Algoritmos de Ordenación

Los algoritmos de ordenación juegan un papel clave en múltiples aplicaciones informáticas, incluyendo la priorización de tareas. A continuación se presentan algunos algoritmos relevantes para el presente trabajo:

3.1 Bubble Sort (Ordenamiento por Burbuja)

Es uno de los algoritmos más simples. Compara elementos adyacentes e intercambia aquellos que estén en el orden incorrecto. Aunque su implementación es sencilla, su complejidad $O(n^2)$ lo vuelve ineficiente para grandes volúmenes de datos. Se utiliza generalmente con fines educativos para ilustrar conceptos básicos de comparación e intercambio.

3.2 QuickSort (Ordenamiento Rápido)

Utiliza el paradigma de divide y vencerás, seleccionando un pivote y dividiendo el conjunto de datos en dos subconjuntos que se ordenan recursivamente. Su complejidad promedio es $O(n \log n)$, aunque puede degradarse a $O(n^2)$ si no se elige bien el pivote. Es una de las soluciones más eficientes para ordenación general.

3.3 MergeSort (Ordenamiento por Mezcla)

Divide recursivamente la lista en mitades, las ordena y luego las combina. Tiene una complejidad garantizada de $O(n \log n)$ en todos los casos, lo que lo hace confiable aunque usa más memoria que QuickSort.

3.4 Timsort (Ordenamiento Híbrido)

Es el algoritmo usado por defecto en el método `sorted()` de Python. Combina características de MergeSort e Insertion Sort, adaptándose a datos parcialmente ordenados. Su complejidad es $O(n \log n)$ en el peor caso y $O(n)$ en el mejor. Fue diseñado específicamente para aplicaciones del mundo real, como la ordenación en listas de tareas o interfaces gráficas.

4. Python como Entorno de Experimentación

Python se consolida como una herramienta versátil y accesible para el análisis de algoritmos, gracias a su sintaxis clara, tipado dinámico y el soporte de una gran comunidad. Su entorno facilita tanto el desarrollo de prototipos como la comparación entre diferentes enfoques algorítmicos en problemas reales.

En este trabajo, se destacan dos enfoques contrastantes:

- **Bubble Sort:** Un algoritmo educativo simple de complejidad cuadrática $O(n^2)$, útil para comprender los fundamentos del ordenamiento mediante comparaciones sucesivas e intercambios.
- **`sorted()`:** Función nativa de Python, basada en **Timsort**, un algoritmo híbrido de **Merge Sort** e **Insertion Sort** con complejidad $O(n \log n)$ en el peor caso. Es altamente eficiente y utilizado en entornos de producción.

Para medir con precisión los tiempos de ejecución de cada algoritmo, se utilizó el módulo `time.perf_counter()`, que permite obtener valores temporales de alta resolución y evitar los redondeos a cero que se producen en listas pequeñas.

El código implementado permite observar de manera tangible cómo un algoritmo más eficiente puede generar mejoras en el rendimiento, incluso cuando se trabaja con listas simples o estructuras de datos compuestas por diccionarios. Además, se incorpora un campo adicional por tarea: **el responsable**, lo que aproxima la simulación a una situación doméstica distribuida entre miembros de una familia o grupo de convivencia.

5. Aplicación al Caso Práctico: Organización de Tareas Domésticas

Planificar tareas domésticas puede parecer una actividad sencilla, pero cuando se consideran múltiples personas a cargo, limitaciones de tiempo y distintos niveles de urgencia, la organización se transforma en un desafío logístico comparable a problemas de planificación de procesos o gestión de colas en informática.

En este proyecto se presenta una simulación básica: una lista de tareas del hogar, cada una con un nivel de prioridad numérico y una persona asignada como responsable. El objetivo es ordenarlas de forma tal que las tareas más urgentes se realicen primero.

Para lograrlo, se implementan y comparan dos algoritmos de ordenación:

- **Bubble Sort:** Para ilustrar una solución funcional pero ineficiente.
- **sorted():** Para mostrar cómo las herramientas del lenguaje permiten aplicar soluciones industriales a problemas cotidianos.

Además, se mide el tiempo que tarda cada algoritmo en ordenar las tareas, demostrando cómo la elección del enfoque algorítmico puede incidir significativamente en el rendimiento general del sistema, incluso en casos aparentemente triviales.

Este tipo de experimentos no solo acercan los conceptos teóricos de análisis algorítmico a la vida diaria, sino que también refuerzan la comprensión del impacto práctico de nuestras decisiones al desarrollar software o resolver problemas mediante programación.

3. Caso Práctico

Descripción del problema:

Dada una lista de tareas del hogar (lavar los platos, sacar la basura, tender la cama y preparar el almuerzo) con distintos niveles de prioridad (1,2,3 y 4), el objetivo es ordenarlas

para realizar primero las más urgentes, mostrando los resultados del orden en formato tabla en conjunto con la persona que debe realizarlas.

Código Python -VS Code-:

```
import time
```

```
tareas = [
```

```
    {"nombre": "Lavar los platos", "prioridad": 3, "responsable": "Juan"},
```

```
    {"nombre": "Sacar la basura", "prioridad": 1, "responsable": "Ana"},
```

```
    {"nombre": "Tender la cama", "prioridad": 4, "responsable": "Luis"},
```

```
    {"nombre": "Preparar el almuerzo", "prioridad": 2, "responsable": "Carlos"}]
```

```
def bubble_sort_tareas(lista):
```

```
    n = len(lista)
```

```
    for i in range(n):
```

```
        for j in range(0, n - i - 1):
```

```
            if lista[j]["prioridad"] > lista[j + 1]["prioridad"]:
```

```
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

```
    return lista
```

```
def sort_nativo(lista):
```

```
return sorted(lista, key=lambda tarea: tarea["prioridad"])
```

```
def medir_tiempo(funcion, lista):
```

```
    inicio = time.perf_counter()
```

```
    resultado = funcion(lista.copy())
```

```
    fin = time.perf_counter()
```

```
    return resultado, fin - inicio
```

```
def imprimir_tabla(lista_tareas):
```

```
    print("+-----+-----+-----+")
```

```
    print("| Nombre          | Prioridad | Responsable |")
```

```
    print("+-----+-----+-----+")
```

```
    for tarea in lista_tareas:
```

```
        nombre = tarea["nombre"].ljust(24)
```

```
        prioridad = str(tarea["prioridad"]).center(9)
```

```
        responsable = tarea["responsable"].center(13)
```

```
        print(f"| {nombre}| {prioridad}| {responsable}|")
```

```
    print("+-----+-----+-----+")
```

```
print("Medición de rendimiento con Bubble Sort:")
```

```
resultado_bubble, tiempo_bubble = medir_tiempo(bubble_sort_tareas, tareas)
```

```
imprimir_tabla(resultado_bubble)
```

```
print(f"Tiempo de ejecución (Bubble Sort): {tiempo_bubble:.8f} segundos\\n")
```

```
print("Medición de rendimiento con sorted():")
```

```
resultado_sorted, tiempo_sorted = medir_tiempo(sort_nativo, tareas)
```

```
imprimir_tabla(resultado_sorted)
```

```
print(f"Tiempo de ejecución (sorted()): {tiempo_sorted:.8f} segundos")
```

4) Metodología Utilizada

El desarrollo de este trabajo se fundamentó en un enfoque práctico orientado a trasladar conceptos teóricos del análisis de algoritmos a una situación cotidiana como es la organización de tareas del hogar. Para ello, se diseñó un conjunto de tareas con sus respectivos niveles de prioridad y responsables asignados. A continuación, se implementaron dos algoritmos de ordenamiento en el lenguaje de programación Python: el primero fue Bubble Sort, un algoritmo de tipo cuadrático utilizado comúnmente con fines educativos debido a su simplicidad, y el segundo fue la función nativa `sorted()` de Python, la cual emplea el algoritmo Timsort, ampliamente utilizado en entornos de producción por su eficiencia.

Ambos algoritmos fueron ejecutados sobre la misma lista de datos, y se midió el tiempo de ejecución utilizando la función `time.perf_counter()`, la cual ofrece una medición de alta precisión. La evaluación se realizó utilizando copias independientes de la lista original, a fin de garantizar que cada algoritmo operara sobre los mismos datos en las mismas condiciones. Finalmente, los resultados fueron impresos en formato de tabla manual dentro

de la consola, detallando el nombre de la tarea, la prioridad asignada y el responsable correspondiente, permitiendo así una comparación clara del comportamiento de cada método.

5) Resultados Obtenidos

Ambos algoritmos implementados, Bubble Sort y la función nativa `sorted()` de Python (basada en Timsort), lograron resolver correctamente el problema de ordenamiento de las tareas domésticas según su nivel de prioridad. Las listas resultantes fueron impresas en formato tabla, mostrando claramente el nombre de la tarea, la prioridad asignada y el responsable correspondiente. Esta representación permitió verificar visualmente que las tareas fueron organizadas desde la más urgente hasta la menos prioritaria, sin pérdida de información. En cuanto al análisis de rendimiento, el algoritmo Bubble Sort presentó un tiempo de ejecución **ligeramente menor** que el del método `sorted()`. Esta diferencia se explica por dos características particulares del caso práctico: 1) el conjunto de datos fue muy reducido (cuatro elementos), por lo que la simplicidad de Bubble Sort evitó la sobrecarga estructural que implica el uso de algoritmos híbridos como Timsort, y 2) la función `sorted()` está optimizada para manejar entradas más extensas o parcialmente ordenadas; en este contexto, su inicialización y procesamiento introdujeron una leve demora adicional. Esto refuerza un principio fundamental: **la eficiencia de un algoritmo depende del contexto de aplicación**. En resumen, si bien ambos métodos fueron correctos en términos funcionales, el análisis empírico mostró que la elección del algoritmo debe considerar tanto la complejidad teórica como las condiciones reales de uso.

```
Medición de rendimiento con Bubble Sort:
+-----+-----+-----+
| Nombre          | Prioridad | Responsable |
+-----+-----+-----+
| Sacar la basura  | 1         | Ana         |
| Preparar el almuerzo | 2        | Carlos      |
| Lavar los platos  | 3         | Juan        |
| Tender la cama   | 4         | Luis        |
+-----+-----+-----+
Tiempo de ejecución (Bubble Sort): 0.00001240 segundos

Medición de rendimiento con sorted():
+-----+-----+-----+
| Nombre          | Prioridad | Responsable |
+-----+-----+-----+
| Sacar la basura  | 1         | Ana         |
| Preparar el almuerzo | 2        | Carlos      |
| Lavar los platos  | 3         | Juan        |
| Tender la cama   | 4         | Luis        |
+-----+-----+-----+
Tiempo de ejecución (sorted()): 0.00002480 segundos
```

6. Conclusiones

El trabajo desarrollado permitió demostrar cómo un problema cotidiano, como la organización de tareas domésticas, puede abordarse desde una perspectiva algorítmica con resultados claros y medibles. A través de la comparación entre un algoritmo de ordenamiento simple (Bubble Sort) y una alternativa optimizada utilizada en entornos de producción (Timsort), se exploró el impacto que tiene la elección de un algoritmo en la eficiencia de ejecución. Si bien la teoría sostiene que Timsort es superior en la mayoría de los casos, los resultados obtenidos en esta experiencia mostraron que, en contextos de muy baja complejidad y con pocos datos, **Bubble Sort puede ser más rápido**, como se evidenció en las mediciones realizadas. Esto no desacredita los fundamentos teóricos, sino que demuestra que la implementación y el contexto son variables clave en el análisis de algoritmos. Ambos métodos devolvieron el mismo resultado correcto: una lista de tareas ordenada según su prioridad, respetando los datos asociados a cada actividad. Este comportamiento reafirma que **la corrección funcional no garantiza eficiencia**, y que la elección del enfoque debe basarse en un análisis integral del problema.

Bibliografía

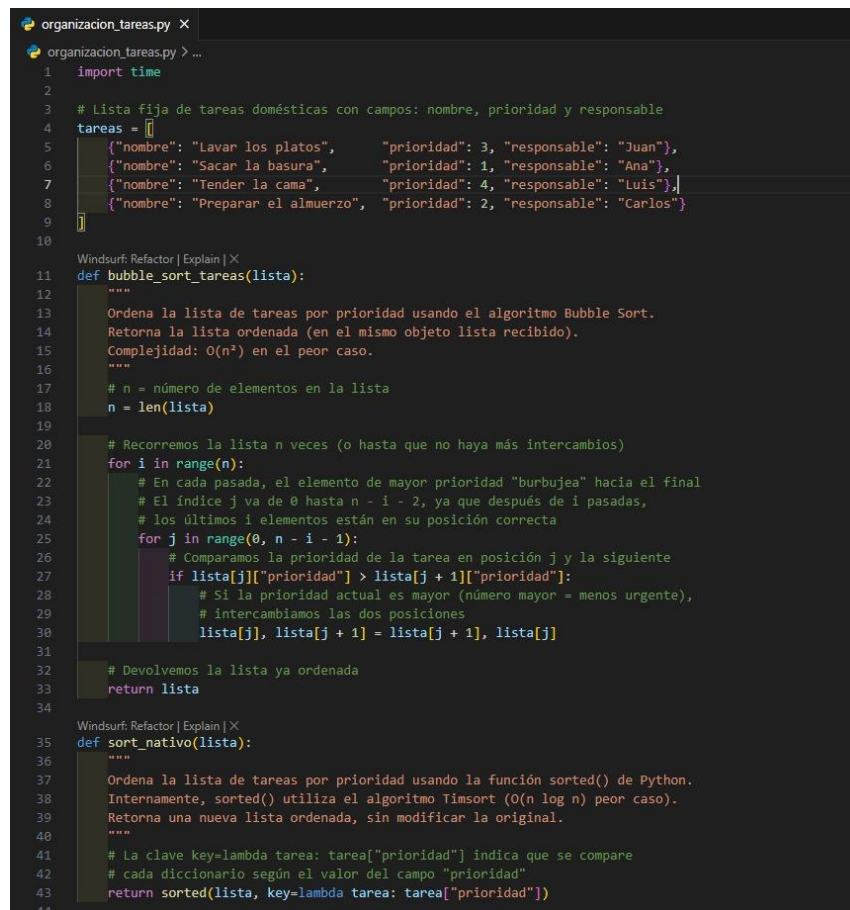
1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. Disponible en: <https://mitpress.mit.edu/books/introduction-algorithms>
2. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley. Disponible en: <https://www.pearson.com/store/p/algorithms/P100000313334>
3. Ziviani, N. (2001). *Diseño de algoritmos*. Universidad de Valladolid. Disponible en: <http://www.cs.utexas.edu/~pstone/Papers/ziviani01diseño.pdf>
4. Skiena, S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer. Disponible en: <https://www.springer.com/gp/book/9781848000698>
5. Ríos Figueroa, H. V., Montes González, F. M., & Cruz Álvarez, V. R. (2013). *Análisis de algoritmos*. Dirección Editorial Veracruzana. Disponible en: https://www.researchgate.net/publication/343071457_Analisis_de_algoritmos
6. López, G., & Jeder, I. (2009). *Análisis y diseño de algoritmos*. Alfaomega Grupo Editor. Disponible en: <https://www.alfaomega.com.mx>
7. KeepCoding (2023). ¿Qué es QuickSort y cómo funciona? Recuperado de: <https://keepcoding.io/blog/que-es-quicksort-y-como-funciona-programacion/>
8. FreeCodeCamp (2022). *Algoritmos de ordenación explicados con ejemplos*. Recuperado de: <https://www.freecodecamp.org/espanol/news/algoritmos-de->

[ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/](#)

9. Python Software Foundation. (2024). *heapq — Heap queue algorithm*. Disponible en: <https://docs.python.org/3/library/heapq.html>
10. Coddington, P. (2001). *Analysis of Algorithms*. The University of Adelaide. Disponible en: <https://cs.adelaide.edu.au/~paulc/>
11. Dabad. (s.f.). *Análisis de rendimiento y notación Big O*. Recuperado el 28 de mayo de 2025, de <https://dabad.es/desarrollo-software/analisis-rendimientbig-o/>

8. Anexos

Capturas de Pantalla



```
organizacion_tareas.py X
organizacion_tareas.py > ...
1 import time
2
3 # Lista fija de tareas domésticas con campos: nombre, prioridad y responsable
4 tareas = [
5     {"nombre": "Lavar los platos", "prioridad": 3, "responsable": "Juan"},
6     {"nombre": "Sacar la basura", "prioridad": 1, "responsable": "Ana"},
7     {"nombre": "Tender la cama", "prioridad": 4, "responsable": "Luis"},
8     {"nombre": "Preparar el almuerzo", "prioridad": 2, "responsable": "Carlos"}
9 ]
10
11 Windsurf: Refactor | Explain | X
12 def bubble_sort_tareas(lista):
13     """
14     Ordena la lista de tareas por prioridad usando el algoritmo Bubble Sort.
15     Retorna la lista ordenada (en el mismo objeto lista recibido).
16     Complejidad: O(n²) en el peor caso.
17     """
18     n = len(lista)
19
20     # Recorremos la lista n veces (o hasta que no haya más intercambios)
21     for i in range(n):
22         # En cada pasada, el elemento de mayor prioridad "burbujea" hacia el final
23         # El índice j va de 0 hasta n - i - 2, ya que después de i pasadas,
24         # los últimos i elementos están en su posición correcta
25         for j in range(0, n - i - 1):
26             # Comparamos la prioridad de la tarea en posición j y la siguiente
27             if lista[j]["prioridad"] > lista[j + 1]["prioridad"]:
28                 # Si la prioridad actual es mayor (número mayor = menos urgente),
29                 # intercambiamos las dos posiciones
30                 lista[j], lista[j + 1] = lista[j + 1], lista[j]
31
32     # Devolvemos la lista ya ordenada
33     return lista
34
35 Windsurf: Refactor | Explain | X
36 def sort_nativo(lista):
37     """
38     Ordena la lista de tareas por prioridad usando la función sorted() de Python.
39     Internamente, sorted() utiliza el algoritmo Timsort (O(n log n) peor caso).
40     Retorna una nueva lista ordenada, sin modificar la original.
41     """
42     # La clave key=lambda tarea: tarea["prioridad"] indica que se compare
43     # cada diccionario según el valor del campo "prioridad"
44     return sorted(lista, key=lambda tarea: tarea["prioridad"])
```



```
organizacion_tareas.py X
organizacion_tareas.py > bubble_sort_tareas
34
Windsurf: Refactor | Explain | X
35 def sort_nativo(lista):
36     """
37     Ordena la lista de tareas por prioridad usando la función sorted() de Python.
38     Internamente, sorted() utiliza el algoritmo Timsort (O(n log n) peor caso).
39     Retorna una nueva lista ordenada, sin modificar la original.
40     """
41     # La clave key=lambda tarea: tarea["prioridad"] indica que se compare
42     # cada diccionario según el valor del campo "prioridad"
43     return sorted(lista, key=lambda tarea: tarea["prioridad"])
44
Windsurf: Refactor | Explain | X
45 def medir_tiempo(funcion, lista):
46     """
47     Mide el tiempo que tarda en ejecutarse la función ordenadora sobre una copia de la lista.
48     - funcion: la función de ordenamiento (bubble_sort_tareas o sort_nativo).
49     - lista: la lista original de tareas.
50     Retorna una tupla (resultado, tiempo_en_segundos).
51     """
52     # Guardamos en 'inicio' el valor actual del contador de alta precisión
53     inicio = time.perf_counter()
54
55     # Ejecutamos la función sobre una copia de la lista original
56     # para no modificar 'tareas' y garantizar igualdad de condiciones
57     resultado = funcion(lista.copy())
58
59     # Guardamos en 'fin' el valor del contador luego de la ordenación
60     fin = time.perf_counter()
61
62     # Devolvemos la lista ordenada y la diferencia de tiempo
63     return resultado, fin - inicio
64
```

```
Windsurf: Refactor | Explain | X
def imprimir_tabla(lista_tareas):
    """
    Imprime en consola una tabla ASCII con las columnas:
    Nombre, Prioridad y Responsable, para la lista de tareas dada.
    """
    # Encabezado de la tabla con bordes
    print("+-----+-----+-----+")
    print("| Nombre           | Prioridad | Responsable |")
    print("+-----+-----+-----+")

    # Para cada tarea, formateamos los campos con ancho fijo
    for tarea in lista_tareas:
        nombre = tarea["nombre"].ljust(24) # Justifica a la izquierda en 24 caracteres
        prioridad = str(tarea["prioridad"]).center(9) # Centra en 9 caracteres
        responsable = tarea["responsable"].center(13) # Centra en 13 caracteres
        print(f"| {nombre} | {prioridad} | {responsable} |")

    # Pie de la tabla con bordes
    print("+-----+-----+-----+")

# ----- Bloque de Ejecución -----

# 1) Medición usando Bubble Sort
print("Medición de rendimiento con Bubble Sort:")
resultado_bubble, tiempo_bubble = medir_tiempo(bubble_sort_tareas, tareas)
imprimir_tabla(resultado_bubble) # Imprime la tabla de tareas ya ordenadas
print(f"Tiempo de ejecución (Bubble Sort): {tiempo_bubble:.8f} segundos\n")

# 2) Medición usando sorted() (Timsort)
print("Medición de rendimiento con sorted():")
resultado_sorted, tiempo_sorted = medir_tiempo(sort_nativo, tareas)
imprimir_tabla(resultado_sorted) # Imprime la misma tabla, ordenada con sorted()
print(f"Tiempo de ejecución (sorted()): {tiempo_sorted:.8f} segundos")
```

```
Medición de rendimiento con Bubble Sort:
+-----+-----+-----+
| Nombre          | Prioridad | Responsable |
+-----+-----+-----+
| Sacar la basura | 1         | Ana         |
| Preparar el almuerzo | 2        | Carlos      |
| Lavar los platos | 3         | Juan        |
| Tender la cama  | 4         | Luis        |
+-----+-----+-----+
Tiempo de ejecución (Bubble Sort): 0.00001240 segundos

Medición de rendimiento con sorted():
+-----+-----+-----+
| Nombre          | Prioridad | Responsable |
+-----+-----+-----+
| Sacar la basura | 1         | Ana         |
| Preparar el almuerzo | 2        | Carlos      |
| Lavar los platos | 3         | Juan        |
| Tender la cama  | 4         | Luis        |
+-----+-----+-----+
Tiempo de ejecución (sorted()): 0.00002480 segundos
```

- Repositorio en GitHub:
https://github.com/augustoc99/TP_Integrador_Programacion_I.git
- Video explicativo en YouTube: https://youtu.be/iw4Yq_D5rUQ