UNIVERSIDAD TECNOLÓGICA NACIONAL

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN
A DISTANCIA

TRABAJO INTEGRADOR:
ANALISIS DE ALGORITMOS – BUSQUEDA Y ORDENAMIENTO

Programación I

Profesor: Ariel Enferrel

Tutor: Franco Gonzalez

Alumnos: Matías FERREYRA
(matiferreyraacade@gmail.com)
Atilano Roberto FERNÁNDEZ
(arfernandez@gmail.com)

Fecha de entrega: 09/06/2025

Índice:

1.- Introducción

En este trabajo se busca comprender la importancia que tiene el **análisis de algoritmos** en la evaluación de distintas estrategias para resolver un mismo problema con el fin de que resulte más eficiente, es decir, que se resuelva en el menor tiempo posible y utilizando la menor cantidad de recursos.

Nuestro objetivo, es tomar un problema hipotético en el cual probaremos distintos algoritmos de búsqueda y de ordenamiento y haremos un análisis de los algoritmos utilizados.

Para contextualizar este ejercicio, supondremos un almacén y definiremos tres conjuntos (listas) de artículos del rubro. Los conjuntos contaran con 10, 100 y 500 artículos y sobre estas listas trabajaremos.

Abordaremos el problema desde un enfoque **empírico** y evaluaremos los algoritmos de **búsqueda lineal** y de **búsqueda binaria** y ordenaremos estas listas utilizando los algoritmos **Bubble Sort** y **Quick Sort**.

2.- Marco Teórico

Para comenzar se definirán algunos conceptos que darán mayor entendimiento a los temas tratados:

- **Algoritmo**: Conjunto de instrucciones bien definidas aplicadas para resolver un problema específico.
- **Análisis Empírico**: Es un enfoque práctico para determinar la eficiencia de un algoritmo. Se trata de ejecutar el mismo y medir el tiempo que dura dicha ejecución al recibir entradas de distinto tamaño.
  Consta de cuatro pasos:
  1. Implementación del algoritmo: Se escribe el código en un lenguaje de programación. En este caso Python.
  2. Instrumentación: Se utilizan instrucciones que permiten medir el tiempo. En este trabajo se utiliza `timeit()`.
  3. Ejecución con diferentes tamaños de entrada: Se obtienen los tiempos para cada tamaño de entrada.

4. Visualización de resultados: Se explayan los resultados en una hoja de cálculo donde se analiza el tiempo de ejecución en función del tamaño de entrada, a través de un gráfico de dispersión (X-Y).

- **Notación Big-O**: La notación Big-O es una herramienta para describir el comportamiento asintótico de una función, es decir, cómo crece cuando el tamaño de la entrada tiende a infinito. Se utiliza para simplificar la comparación de algoritmos eliminando constantes y términos de menor orden.

- **Asintótico**: Es una forma matemática de describir el comportamiento del tiempo de ejecución o del uso de memoria de un algoritmo cuando el tamaño de la entrada crece mucho. No se enfoca en detalles exactos como segundos o bytes, sino en cómo escala un algoritmo: qué tan rápido crece su costo a medida que aumenta el tamaño del problema.

- **Búsqueda**: Es la localización de un elemento en un conjunto de datos. A continuación, se explican los dos métodos más comunes, que serán utilizados en esta oportunidad:
  - Búsqueda lineal: Es un algoritmo que busca un valor dentro de un conjunto de datos, uno por uno, hasta encontrar el valor buscado o hasta que se recorran todos los elementos.

    Ventaja: Funciona en listas desordenadas.

    Desventaja: Es un proceso lento para listas grandes.

    Complejidad Temporal:
    - Peor caso: O(n), donde n es el número de elementos en la lista. Cuando el elemento buscado está al final o no está presente.
    - Mejor caso: O(1), cuando el elemento se encuentra en la primera posición.
    - Caso promedio: O(n), ya que en promedio el algoritmo revisa la mitad de los elementos antes de encontrar el objetivo (o no encontrarlo).

    Código en Python:

```python
def buscar_por_nombre_lineal(lista, nombre):

    for producto in lista:

        if producto["nombre"] == nombre:
            return producto

    return None
```

  - Búsqueda binaria: La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada. Funciona dividiendo repetidamente por la mitad la

parte de la lista que podría contener el elemento, hasta reducir las posibles ubicaciones a una sola.

Compara el valor buscado con el elemento del medio. Si no es igual, descarta la mitad donde en la que el valor no puede estar y repite el proceso. Así hasta encontrar el elemento o hasta determinar que no se encuentra dentro del conjunto.

Ventaja: Mucho más rápida que la búsqueda lineal.

Requisito: La lista debe estar ordenada.

Complejidad Temporal:

- Peor caso: O(log n), donde n es el número de elementos en la lista. Ocurre cuando se requiere dividir varias veces hasta llegar a un único elemento.
- Mejor caso: O(1), si el elemento está justo en la posición central en la primera comparación.
- Caso promedio: O(log n), porque reduce el espacio de búsqueda a la mitad en cada paso.

Código en Python:

```python
def buscar_por_nombre_binario(lista, nombre):

    inicio = 0
    final = len(lista) - 1

    while inicio <= final:
        medio = (inicio + final) // 2 # Encontramos el punto medio de la lista
        nombre_medio = lista[medio]["nombre"] # Obtenemos el nombre del producto que esta en el medio.
        nombre_buscado = nombre

        if nombre_medio == nombre_buscado:
            return lista[medio] # Devuelve el producto encontrado en la posición medio
        elif nombre_medio < nombre_buscado:
            inicio = medio + 1 # Corremos el inicio hacia la derecha (el buscado esta en la mitad de la derecha)
        else:
            final = medio - 1 # Corremos el final hacia la izquierda (el buscado esta en la mitad de la izquierda)

    return None
```

Comparación de ambos métodos de búsqueda por complejidad temporal.

Algoritmo	Mejor Caso	Peor Caso	Caso promedio	Eficiencia	Uso Recomendado
Lineal	O(1)	O(n)	O(n)	Baja	Listas pequeñas o no ordenadas
Binaria	O(1)	O(log n)	O(log n)	Alta	Listas grandes y ordenadas

- **Ordenamiento**: Es la organización de los datos a partir de un criterio específico. En este caso se definirán y se utilizarán dos métodos entre los diversos existentes:

- Bubble Sort (Ordenamiento de Burbuja): Es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado.

  Ventajas: Fácil de comprender y programar.

  Desventajas: No es eficiente para listas grandes. Puede llegar a realizar comparaciones innecesarias.

  Complejidad Temporal:

  - Peor caso: $O(n^2)$, donde n es el número de elementos en la lista. Esto ocurre cuando la lista está en orden inverso.
  - Mejor caso: $O(n^2)$, incluso si la lista ya está completamente ordenada, el algoritmo recorre todas las pasadas y hace todas las comparaciones posibles, porque no tiene forma de saber que ya está ordenada. Existe una optimización que detecta si no hubo intercambios, pero en este trabajo se opta por utilizar su versión clásica.
  - Caso promedio: $O(n^2)$.

  Código en Python:

```python
def bubble_sort(lista,key):
    for i in range(len(lista)):
        for j in range(len(lista) - 1 - i):
            if lista[j][key] > lista[j + 1][key]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]

    return lista
```

- Quick Sort (Ordenamiento Rápido): Es un algoritmo eficiente que se basa en el concepto de “Divide y vencerás”. Trabaja de la siguiente forma:

- “Elije” un elemento del conjunto de elementos a ordenar, al que se denomina pivote.
- Reubica los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repite este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso los elementos estarán ordenados.

  Ventajas: Es más rápido que Bubble Sort en la mayoría de los casos. Funciona bien para listas grandes.

  Desventajas: Para su “peor caso” su rendimiento se degrada considerablemente.

Complejidad Temporal:

- Peor caso: $O(n^2)$, cuando el pivote seleccionado es el menor o mayor elemento en cada partición (por ejemplo, si la lista ya está ordenada).
- Mejor caso: O(n log n), cuando el pivote divide la lista en dos partes aproximadamente iguales.
- Caso promedio: O(n log n).

Código en Python:

```python
def quick_sort(lista, key):
    # caso base
    if len(lista) <= 1:
        return lista

    # paso recursivo

    # Usamos el último elemento como pivote
    pivote = lista.pop()
    lista1 = []
    lista2 = []

    for e in lista:
        if e[key] <= pivote[key]:
            lista1.append(e)
        else:
            lista2.append(e)

    lista1 = quick_sort(lista1, key)
    lista2 = quick_sort(lista2, key)

    # pivote se concatena como una lista de un elemento
    return lista1 + [pivote] + lista2
```

Comparación de ambos métodos de ordenamiento por complejidad temporal.

Algoritmo	Mejor Caso	Peor Caso	Caso promedio	Eficiencia	Uso Recomendado
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	Baja	Listas pequeñas o casi ordenadas
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	Alta	Listas grandes (evitar peor caso)

El **Análisis de Algoritmos**, es el estudio de cómo se comporta un algoritmo, en términos de eficiencia y uso de recursos, especialmente tiempo de ejecución y uso de memoria, a medida que crece el tamaño de la entrada.

El objetivo de este tipo de análisis es medir el rendimiento de un algoritmo y para ello se determina la eficiencia en términos de **complejidad temporal** es decir, cuánto tiempo tarda un algoritmo en resolver un problema y **complejidad espacial**, cuánta memoria y recursos se utilizan.

Este análisis es necesario ya que, en programación, un mismo problema puede tener varias soluciones diferentes, por lo cual es importante poder comparar dichas soluciones y elegir la más eficiente. Esto puede marcar la diferencia a la hora de desarrollar aplicaciones que manejen grandes volúmenes de datos o que requieran alta demanda de procesamiento.

Para abordar este análisis, se utilizan dos aproximaciones: **Análisis Teórico** y **Análisis Empírico**.

En el siguiente cuadro se resumen las principales diferencias:

Aspecto	Análisis Teórico	Análisis Empírico
Definición	El análisis teórico es un enfoque matemático para evaluar la eficiencia de un algoritmo sin necesidad de implementarlo. Este método se basa en el pseudocódigo del algoritmo y permite calcular una **función temporal T(n)**, que representa el número de operaciones que realiza el algoritmo para una entrada de tamaño n.	El análisis empírico es un enfoque práctico para medir la eficiencia de un algoritmo mediante la observación directa de su tiempo de ejecución. Este método implica implementar el algoritmo y medir cuánto tiempo tarda en resolver un problema para diferentes tamaños de entrada.
Propósito	Estimar cómo crecerá el costo computacional según el tamaño de entrada, sin ejecutarlo.	Observar el rendimiento real del algoritmo en una implementación específica.
Dependencia del hardware	**No** depende del hardware ni del lenguaje de programación.	**Sí**, depende del hardware, compilador, sistema operativo, etc.
Precisión	Generaliza el comportamiento (en el peor/mejor/promedio caso).	Es precisa para entradas específicas en una máquina concreta.
Ventajas	- Universal - Independiente del entorno - Útil para comparar algoritmos conceptualmente.	- Refleja el rendimiento **real** - Detecta cuellos de botella no evidentes teóricamente.
Limitaciones	- No muestra costos reales - No considera constantes ocultas ni sobrecarga del sistema.	- Resultados **no generalizables** - Requiere múltiples pruebas para obtener conclusiones confiables.

Los **algoritmos de búsqueda y ordenamiento** son elementos esenciales en el desarrollo de software, ya que permiten gestionar la información de forma eficiente, precisa y escalable, lo que favorece la creación de programas más rápidos, estructurados y confiables.

A efectos de comprobar la eficiencia de estos procesos se realizarán diversas pruebas de implementación de los distintos algoritmos de búsqueda y de ordenamiento, y se llevará a cabo un análisis de tipo empírico en el que se medirán los tiempos de ejecución de estos.

Las muestras para procesar por los algoritmos serán listas de diccionarios de diferentes tamaños.

La **búsqueda** es una operación fundamental en programación que permite encontrar un elemento específico dentro de un conjunto de datos.

Entre los diversos algoritmos de búsqueda podemos encontrar: búsqueda lineal, búsqueda binaria, búsqueda de interpolación, entre otros, y cada uno presenta ventajas y desventajas. Algunos ejemplos de uso son:

- Búsqueda de registros en bases de datos.
- Routing en redes.
- Buscar y Reemplazar en editores de texto.

El **ordenamiento** permite organizar los datos de acuerdo con un criterio, como por ejemplo de menor a mayor o alfabéticamente.

Esta técnica es de suma importancia ya que nos permite estructurar y organizar los datos, lo que lleva a una optimización de otros procesos como búsquedas, análisis de datos, detección de duplicados, etc.

Ejemplos de esto, son vistos a diario en aplicaciones web que nos permiten ordenar por precio, relevancia, puntaje, etc.

Todos estos conceptos serán aplicados en un caso práctico desarrollado en Python, en el cual, a través de un análisis empírico, se evaluarán los tiempos de ejecución de los distintos algoritmos, tanto de búsqueda como de ordenamiento.
Para llevar a cabo este proceso se utilizarán entradas de datos de diferentes tamaños a efectos de evaluar cuál es el desempeño de cada algoritmo para cada caso y también se corroborará si

los resultados son coincidentes con la complejidad temporal que presenta cada algoritmo en la teoría.

Fuentes:

- https://es.wikipedia.org/
- https://www.geeksforgeeks.org/
- https://www.khanacademy.org/
- Apuntes de la materia: “Análisis de Algoritmos” y “Búsqueda y Ordenamiento”. De La Tecnicatura Universitaria en Programación a Distancia (UTN) - 2015

3.- Caso Práctico

En esta práctica se determinará qué tipo de algoritmo de búsqueda es más eficiente para distintas muestras de datos de diferentes tamaños.
Asimismo se representará la complejidad temporal de los mismos determinando peor caso y mejor caso.

Se analizarán primeramente los dos algoritmos de búsqueda más comunes: Búsqueda Lineal y Búsqueda Binaria, los cuales buscarán un producto dentro de la lista a partir del nombre que ingrese el usuario.
Luego se procederá con dos algoritmos de ordenamiento: Bubble Sort y Quick Sort, los cuales fueron seleccionados ya que se presume que pueden presentar diferencias significativas.
Estos organizarán la lista de acuerdo al criterio seleccionado por el usuario por consola: “nombre”, “precio” o “vencimiento”.

Cada uno de los cuatro será aplicado en una función que recibirá en como parámetro una lista de productos. En su primera ejecución se analizará una lista de 10 productos, luego otra de 100 productos y por último otra de 500 productos.

Cada producto es un diccionario compuesto por cuatro claves (key): “id”, “nombre”, “precio” y “vencimiento”.

Para medir los tiempos de ejecución, en una primera instancia se utilizó la función **time()** pero dado que los tiempos a medir eran muy bajos, muchos resultados devolvían cero(0).
Por lo tanto se optó por cambiar a la función **perf_counter()** la cuál sí arrojó resultados.
Igualmente los tiempos medidos eran demasiado bajos debido a los tamaños de las listas.

En este punto se evaluó la posibilidad de generar listas de mayor tamaño de forma automática, pero la intención desde un principio fue representar un caso cercano a la realidad. Por esto, para mantener las listas de productos, nos encontramos con la necesidad de ejecutar la medición repetidas veces para obtener resultados más observables.

En este caso se implementa de forma definitiva la función **timeit()** la cual *"proporciona una forma sencilla de cronometrar pequeños fragmentos de código Python"(Fuente: https://docs.python.org/3/library/timeit.html).*
Además esta función brinda la posibilidad de pasar la función a evaluar y la cantidad de repeticiones directamente como parámetros.

Asimismo se encontró el problema de que los tiempos de ejecución variaban entre las distintas pruebas para un mismo conjunto de datos analizado por un mismo algoritmo.
Dado este inconveniente se considera conveniente promediar los tiempos medidos. Con este propósito se crea la función **medir_tiempo_promedio()** que calcula el tiempo total y lo divide entre las repeticiones ejecutadas.

Una vez ejecutado el programa se mostrarán por consola los siguientes datos:

- Resultados de las búsquedas:
  - El producto completo, con sus claves y valores. En caso de no ser encontrado se devolverá un mensaje que lo informe.
  - El tiempo de ejecución del algoritmo.
  - **Esta información se verá representada para cada algoritmo y para los tres tamaños de lista por separado.**

- Resultados de los ordenamientos:
  - La lista de productos ordenada según el criterio seleccionado por el usuario.
  - El tiempo de ejecución del algoritmo.
  - **Esta información se verá representada para cada algoritmo y para los tres tamaños de lista por separado.**

Código fuente: Se muestran capturas del código escrito en Python de las funciones principales y el programa principal.
El archivo del programa completo se podrá revisar en el repositorio de GitHub incluido en los Anexos.

Función de búsqueda general.

```
def busqueda_general(nombre_busqueda, funcion_busqueda, lista, nombre_producto):
    print(f"{nombre_busqueda}")

    if funcion_busqueda.__name__ == "buscar_por_nombre_binaria": # Si es búsqueda
binaria ordenamos la lista
        lista = sorted(lista, key=clave_ordenamiento)

    resultado = funcion_busqueda(lista, nombre_producto) # Buscarmos el producto
    mostrar_producto(resultado) # Lo mostramos formateado

    tiempo = medir_tiempo_promedio(lambda: funcion_busqueda(lista,
nombre_producto)) # Medición del tiempo promedio
    print(f"Tiempo promedio: {tiempo:.6f} ms\n")
```

Función de búsqueda lineal.

```
def buscar_por_nombre_lineal(lista, nombre):

    for producto in lista:

        # Si el valor de la clave "nombre" == nombre(ingresado por el usuario)
        if producto["nombre"] == nombre:
            return producto

    return None
```

Función de búsqueda binaria.

```
def buscar_por_nombre_binaria(lista, nombre):

    inicio = 0
    final = len(lista) - 1

    while inicio <= final:
        medio = (inicio + final) // 2 # Encontramos el punto medio de la lista
        nombre_medio = lista[medio]["nombre"] # Obtenemos el nombre del producto
que esta en el medio.
        nombre_buscado = nombre

        if nombre_medio == nombre_buscado:
            return lista[medio] # Devuelve el producto encontrado en la posición
medio
        elif nombre_medio < nombre_buscado:
            inicio = medio + 1 # Corremos el inicio hacia la derecha (el buscado
esta en la mitad de la derecha)
        else:
```

```
            final = medio - 1 # Corremos el final hacia la izquierda (el buscado
esta en la mitad de la izquierda)

    return None
```

Función de ordenamiento general.

```
def ordenamiento_general(nombre_ordenamiento, funcion_ordenamiento, lista,
clave):
    print(f"{nombre_ordenamiento}")

    # Medimos el tiempo tiempo
    tiempo = medir_tiempo_promedio(lambda: funcion_ordenamiento(lista, clave))

    # Ordenamos y mostramos primeros productos
    lista_ordenada = funcion_ordenamiento(lista, clave)
    mostrar_lista_ordenada(lista_ordenada)

    print(f"Tiempo promedio: {tiempo:.6f} ms\n")
```

Función Bubble Sort.

```
def bubble_sort(lista,key): # key indica al algoritmo por cuál clave ordenar
    for i in range(len(lista)):
        for j in range(len(lista) - 1 - i):
            if lista[j][key] > lista[j + 1][key]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]

    return lista # Retorna la lista ordenada
```

Función Quick Sort.

```
def quick_sort(lista, key):
    # caso base
    if len(lista) <= 1:
        return lista

    # paso recursivo

    pivote = random.choice(lista)
    # Creamos listas vacías
    lista1 = []
    lista2 = []

    for e in lista[:-1]: # Se recorre c/ elemento de la lista
        # Se indica tanto al elemento como al pivote cuál es la clave del
diccionario a considerar
        if e[key] <= pivote[key]:
            lista1.append(e) # Se agrega el elemento a la lista1
        else:
```

```
            lista2.append(e) # Se agrega el elemento a la lista2

    lista1 = quick_sort(lista1, key) # Se aplica recursividad en ambas listas
    lista2 = quick_sort(lista2, key)

    return lista1 + [pivote] + lista2 # pivote se concatena como una lista de un
elemento
```

Función para medir los tiempos.

```
def medir_tiempo_promedio(funcion, repeticiones=10000):

    tiempo_total = timeit.timeit(funcion, number=repeticiones)

    return (tiempo_total / repeticiones) * 1000  # Devuelve el tiempo promedio en
ms
```

Programa Principal.

```
from utils_busqueda import buscar_por_nombre_lineal, buscar_por_nombre_binaria,
busqueda_general
from utils_ordenamiento import ordenamiento_general, bubble_sort, quick_sort

# Función para pasar nombres a minúsculas
def productos_en_minusculas(lista):
    for producto in lista:
        producto["nombre"] = producto["nombre"].lower()
    return lista


def preguntar_opcion():

    opcion = 0

    while opcion < 1 or opcion > 4:
        opcion = int(input("Por favor selecciona una opción de búsqueda:\n"
                "1. Nombre\n"
                "2. Precio.\n"
                "3. Vencimiento.\n"
                "4. Finalizar programa.\n"
            ))
        if opcion < 1 or opcion > 4:
            print("La opción seleccionada es incorrecta")
    if opcion == 1:
        opcion = "nombre"
    elif opcion == 2:
        opcion = "precio"
    elif opcion == 3:
        opcion = "vencimiento"
```

```
    elif opcion == 4:
        exit()

    return opcion

# Programa Principal

# Listas de distintos tamaños
lista_productos_10 = [
    {"id": "0001", "nombre": "Aceite", "precio": 2500.00, "vencimiento": "2025-
11-05"},
    {"id": "0002", "nombre": "Arroz", "precio": 1200.50, "vencimiento": "2026-01-
15"},
    {"id": "0003", "nombre": "Leche", "precio": 1800.75, "vencimiento": "2025-11-
20"},
    {"id": "0004", "nombre": "Pan", "precio": 900.25, "vencimiento": "2025-11-
10"},
    {"id": "0005", "nombre": "Huevos", "precio": 2200.00, "vencimiento": "2025-
12-01"},
    {"id": "0006", "nombre": "Azucar", "precio": 1500.90, "vencimiento": "2026-
03-01"},
    {"id": "0007", "nombre": "Cafe", "precio": 3000.10, "vencimiento": "2026-04-
25"},
    {"id": "0008", "nombre": "Fideos", "precio": 1100.80, "vencimiento": "2026-
02-12"},
    {"id": "0009", "nombre": "Sal", "precio": 700.30, "vencimiento": "2026-05-
30"},
    {"id": "0010", "nombre": "Harina", "precio": 1000.65, "vencimiento": "2026-
05-01"}
]

lista_productos_100 = [
    {"id": "0001", "nombre": "Aceite", "precio": 2500.00, "vencimiento": "2025-
11-05"},
    {"id": "0002", "nombre": "Arroz", "precio": 1200.50, "vencimiento": "2025-11-
15"},
    {"id": "0003", "nombre": "Leche", "precio": 1800.75, "vencimiento": "2025-11-
20"},
    {"id": "0004", "nombre": "Pan", "precio": 900.25, "vencimiento": "2025-11-
25"},…
]

lista_productos_500 = [
    {"id": "0001", "nombre": "Aceite", "precio": 2500.00, "vencimiento": "2025-
11-05"},
    {"id": "0002", "nombre": "Arroz", "precio": 1200.50, "vencimiento": "2025-11-
10"}, …

]
```

```
# Listas con los nombres de los productos en minúsculas
lista_10_minusculas = productos_en_minusculas(lista_productos_10)
lista_100_minusculas = productos_en_minusculas(lista_productos_100)
lista_500_minusculas = productos_en_minusculas(lista_productos_500)

# El usuario ingresa el producto a buscar
nombre_producto = input("Buscar producto por nombre: ").lower()

# Búsquedas

busqueda_general("---Búsqueda Lineal para 10 productos---",
buscar_por_nombre_lineal, lista_10_minusculas, nombre_producto)

busqueda_general("---Búsqueda Lineal para 100 productos---",
buscar_por_nombre_lineal, lista_100_minusculas, nombre_producto)

busqueda_general("---Búsqueda Lineal para 500 productos---",
buscar_por_nombre_lineal, lista_500_minusculas, nombre_producto)

busqueda_general("---Búsqueda Binaria para 10 productos---",
buscar_por_nombre_binaria, lista_10_minusculas, nombre_producto)

busqueda_general("---Búsqueda Binaria para 100 productos---",
buscar_por_nombre_binaria, lista_100_minusculas, nombre_producto)

busqueda_general("---Búsqueda Binaria para 500 productos---",
buscar_por_nombre_binaria, lista_500_minusculas, nombre_producto)


# Ordenamientos

criterio = preguntar_opcion()

ordenamiento_general("---Bubble Sort para 10 productos---", bubble_sort,
lista_10_minusculas, criterio)

ordenamiento_general("---Bubble Sort para 100 productos---", bubble_sort,
lista_100_minusculas, criterio)

ordenamiento_general("---Bubble Sort para 500 productos---", bubble_sort,
lista_500_minusculas, criterio)

ordenamiento_general("---Quick Sort para 10 productos---", quick_sort,
lista_10_minusculas, criterio)

ordenamiento_general("---Quick Sort para 100 productos---", quick_sort,
lista_100_minusculas, criterio)
```

```
ordenamiento_general("---Quick Sort para 500 productos---", quick_sort,
lista_500_minusculas, criterio)
```

4.- Metodología

Las tareas principales fueron repartidas entre ambas partes del equipo. Una parte se encargó de desarrollar los algoritmos de Búsqueda lineal y Bubble Sort, mientras que la otra parte desarrolló los algoritmos de Búsqueda Binaria y Quick Sort.
Luego en forma conjunta por medio de reuniones virtuales se llevaron a cabo las tareas para unificar el código, implementando las funciones de medición del tiempo, formateos de las salidas por consola y el desarrollo del programa principal.

La información sobre la forma de trabajar de los algoritmos se obtuvo del material otorgado en la unidad de la materia y por medio de videos explicativos cuyos links se anexarán en el punto correspondiente.

El desarrollo del programa se realizó en el lenguaje de programación Python y su ejecución en el IDE Visual Studio Code.

Se utilizaron como datos de entrada dos listas de diccionarios que representan un listado de productos. La primera lista contiene 10 productos y la segunda 100 productos. Dichas listas fueron creadas por medio de la Inteligencia Artificial de Google, Gemini.

Dentro del proyecto se utilizó el módulo **timeit** que, como se explicó anteriormente permitió medir los tiempos de ejecución de los algoritmos con mayor exactitud que otros como time o perf_counter.

Las pruebas se llevaron a cabo en una Notebook Dell Inspiron 15 5000 series, con un procesador Ryzen 5 2500U de 64 bits y 16GB de memoria RAM.
El sistema operativo del equipo es Windows 10 Home.

Pruebas realizadas:

Para cada una de las pruebas mencionadas a continuación se utilizaron listas de diccionarios de distintos tamaños: 10, 100 y 500 elementos.

- Búsqueda lineal representando Peor caso y Mejor caso.

- Búsqueda binaria representando Peor caso y Mejor caso.
- Bubble Sort representando Peor caso y Mejor caso.
- Quick Sort representando Peor caso y Mejor caso.
- Comparación: Búsqueda lineal y búsqueda binaria para un elemento cercano al inicio de la lista.
- Comparación: Búsqueda lineal y búsqueda binaria para un elemento cercano al final de la lista o que no se encuentra en ella.
- Comparación entre Bubble Sort y Quick Sort.

En cada prueba se midieron los tiempos de ejecución y los resultados fueron volcados en una hoja de cálculo a través de la cual se generaron los gráficos de dispersión correspondientes.

5.- Resultados Obtenidos

En este apartado se presentan los resultados arrojados por las pruebas mencionadas en el apartado anterior.

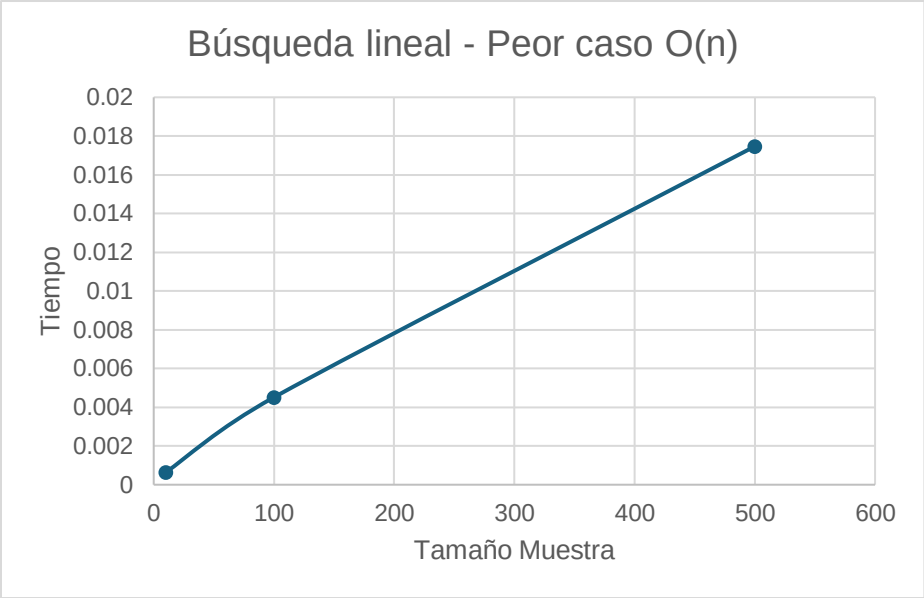
Datos utilizados como entrada:

Se utilizaron 3 listas de diccionarios cada uno de los cuales, como se mencionó anteriormente contienen las claves (keys) “id”, “nombre”, “precio” y “vencimiento”.

- Lista de 10 productos.
- Lista de 100 productos.
- Lista de 500 productos.

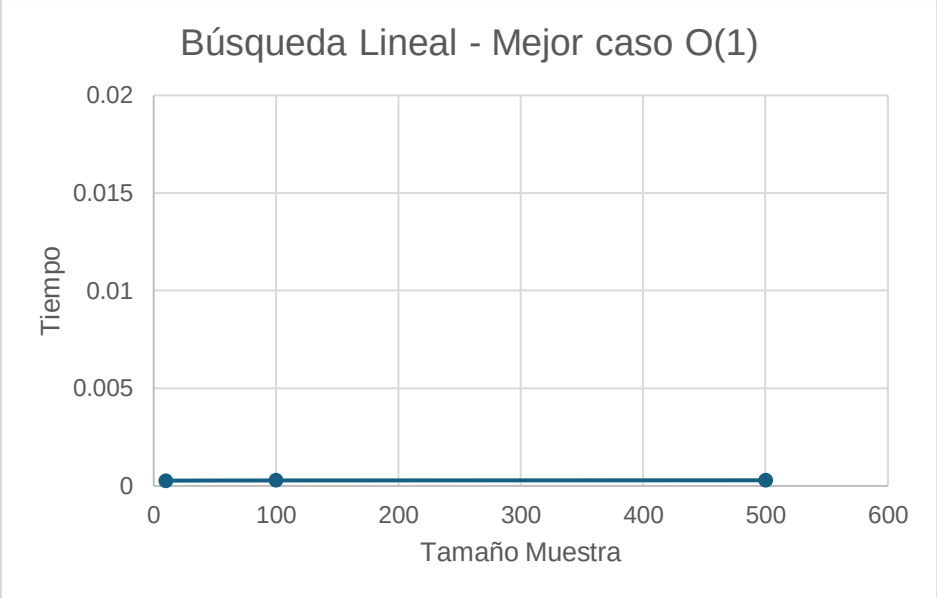
Búsqueda lineal representando Peor caso:

Tamaño Lista	Tiempo B Lineal (ms)	Condiciones
10	0,000639	Se representa el peor caso O(n) - El elemento no se encuentra en la lista
100	0,004512	
500	0,017467	



Búsqueda lineal representando Mejor caso:

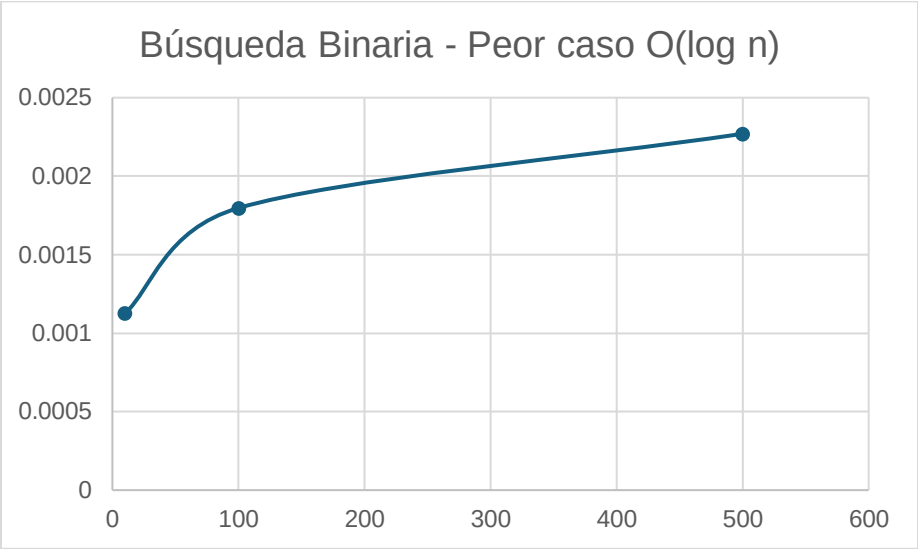
Tamaño Lista	Tiempo B Lineal (ms)	Condiciones
10	0,000273	Se representa el mejor caso O(1) - El elemento se encuentra primero en la lista
100	0,000282	
500	0,00029	



Búsqueda Binaria representando Peor Caso:

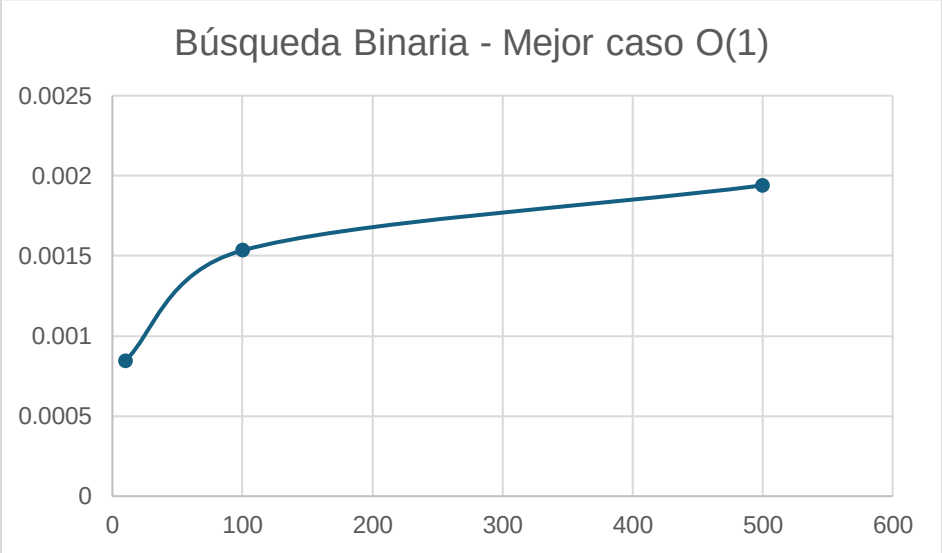
Tamaño Lista	Tiempo B Binaria (ms)	Condiciones
10	0,001125	Se representa el peor caso O(n) - El elemento no se encuentra en la lista
100	0,001797	

500	0,00227	



Búsqueda Binaria representando Mejor Caso:

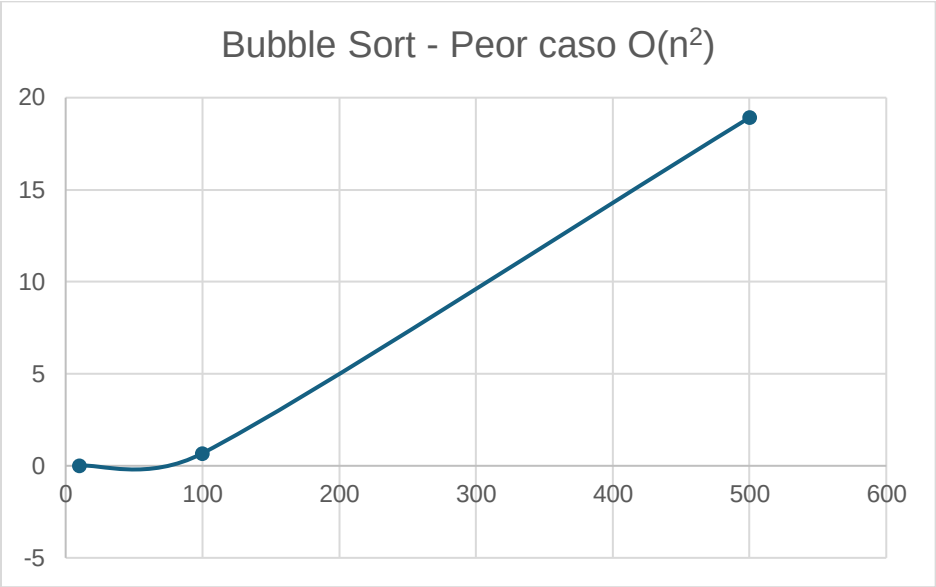
Tamaño Lista	Tiempo B Binaria (ms)	Condiciones
10	0,000846	Se representa el mejor caso O(1) - El elemento se encuentra en el centro de la lista
100	0,001534	
500	0,00194	



Bubble Sort representando Peor Caso:

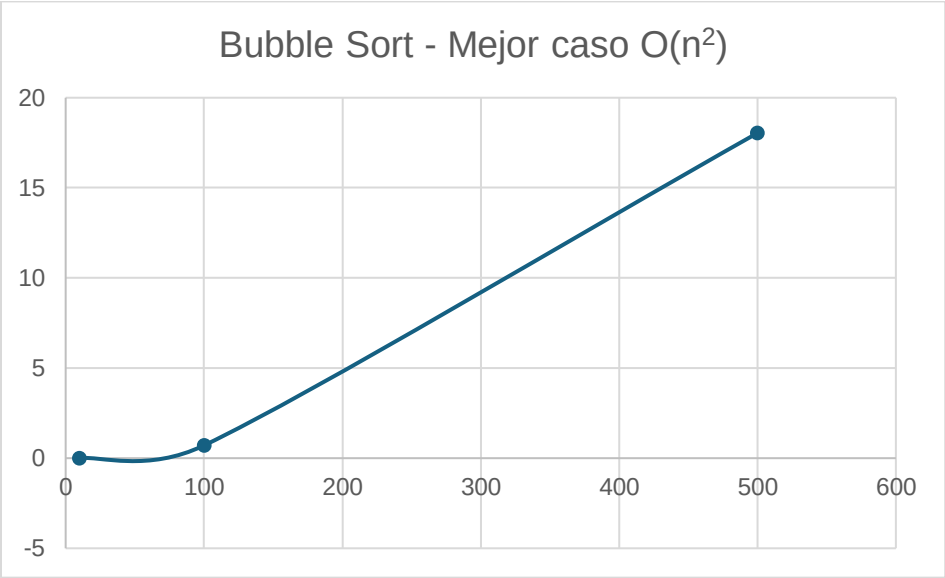
Tamaño Lista	Tiempo Bubble S (ms)	Condiciones
10	0,011437	Se representa el peor caso O(n^2) - La lista está en orden inverso
100	0,680786	

500	18,916333	



Bubble Sort representando Mejor Caso:

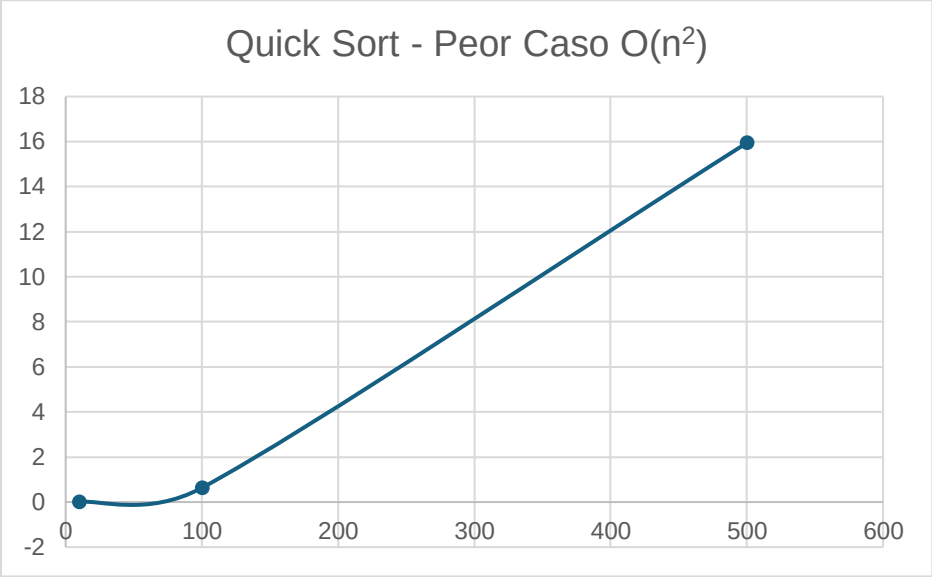
Tamaño Lista	Tiempo Bubble S (ms)	Condiciones
10	0,009695	Se representa el mejor caso O(n^2) - La lista se encuentra ordenada (Sin optimización)
100	0,707512	
500	18,042563	



Quick Sort representando Peor Caso:

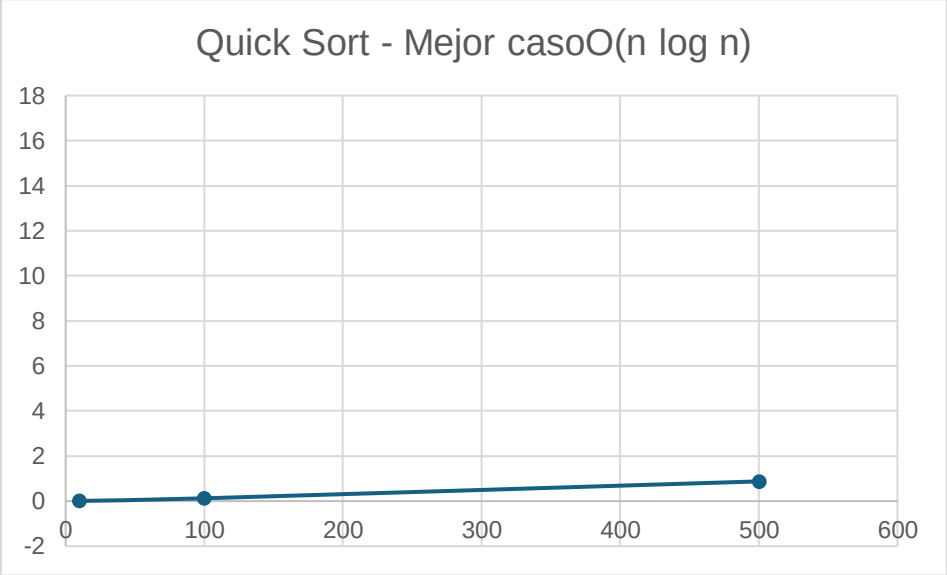
Tamaño Lista	Tiempo Quick S (ms)	Condiciones
10	0,01182	

100	0,631385	Se representa el peor caso $O(n^2)$ - El pivote es el menor o el mayor elemento en cada partición. Por ej en una lista ordenada
500	15,942763	



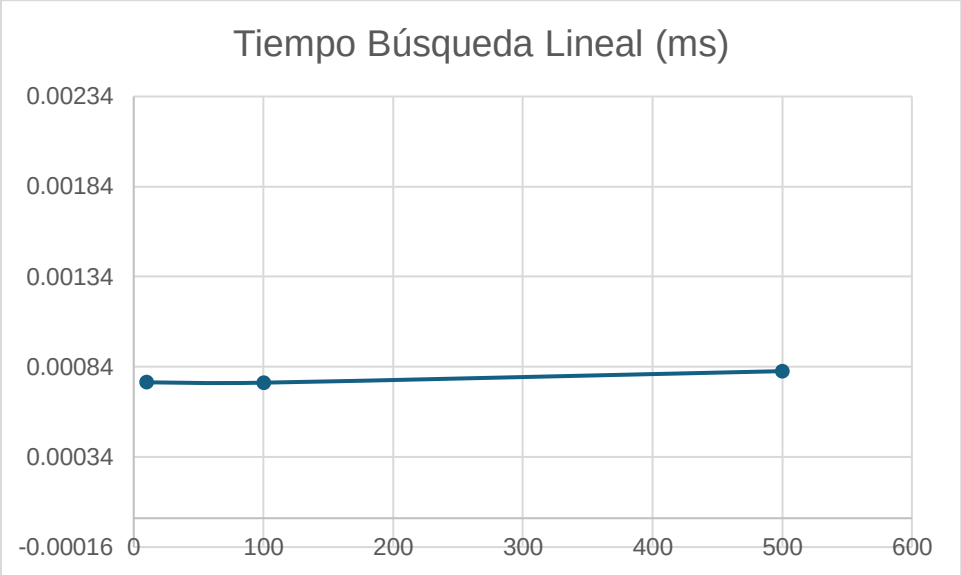
Quick Sort representando Mejor caso:

Tamaño Lista	Tiempo Quick S (ms)	Condiciones
10	0,007771	Se representa el mejor caso O(n log n) - El pivote divide la lista en dos partes iguales o casi iguales
100	0,123424	
500	0,87446	

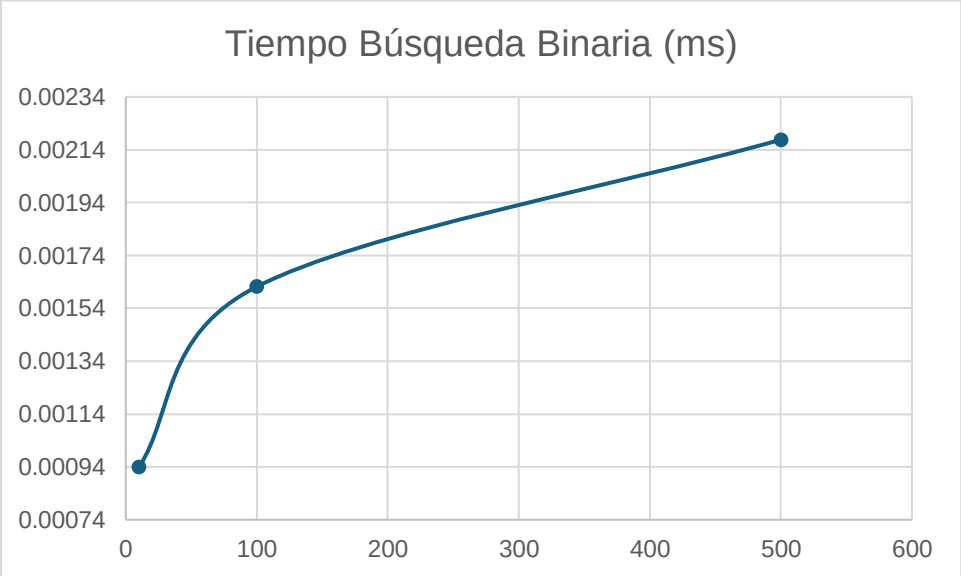


Comparación: Búsqueda lineal y búsqueda binaria para un elemento cercano al inicio de la lista.

Tamaño Lista	Tiempo B Lineal (ms)	Observaciones
10	0,000754	Para un elemento cercano al inicio de la lista
100	0,000752	
500	0,000816	

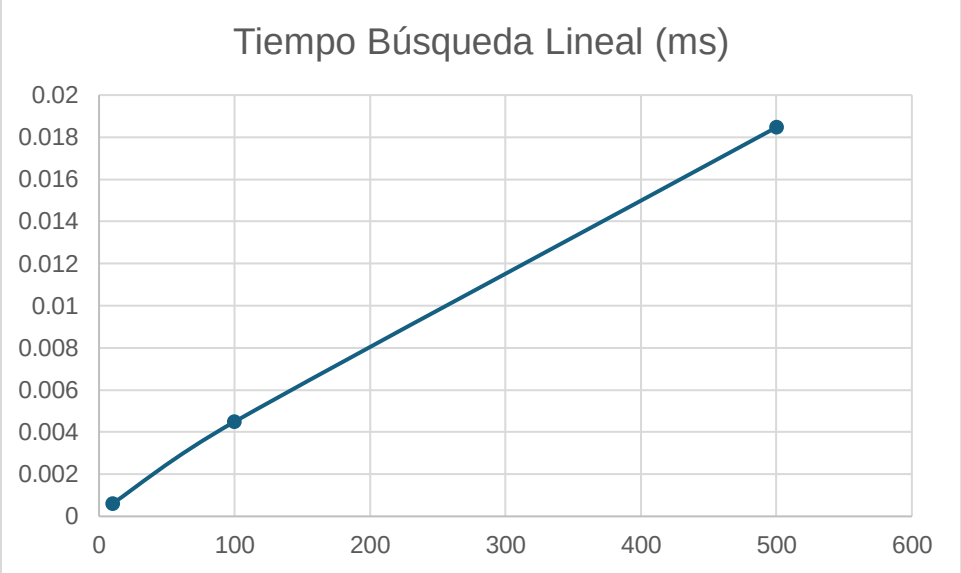


Tamaño Lista	Tiempo B Binaria (ms)	Observaciones
10	0,000938	Para un elemento cercano al inicio de la lista
100	0,001622	
500	0,002178	

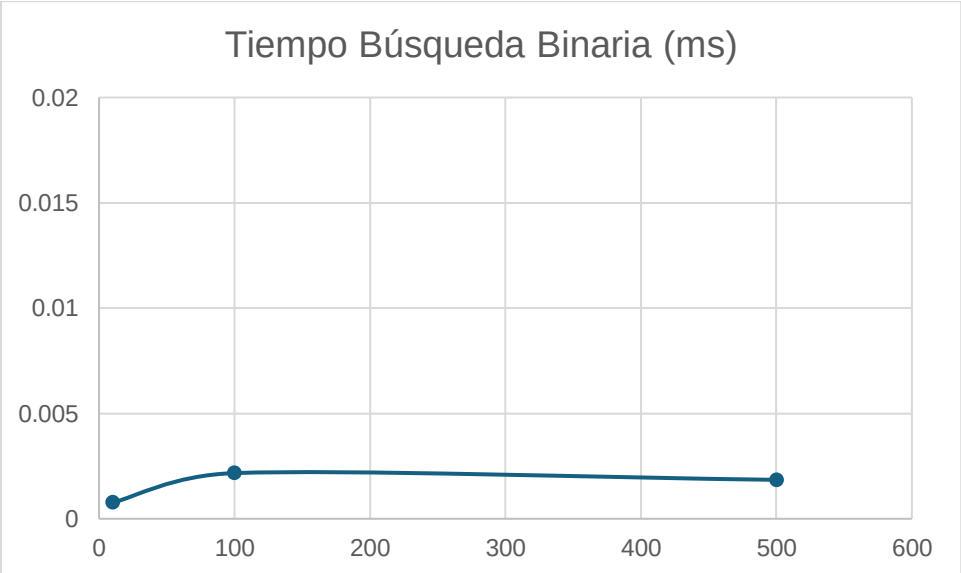


Comparación: Búsqueda lineal y búsqueda binaria para un elemento cercano al final de la lista o que no se encuentra en ella.

Tamaño Lista	Tiempo B Lineal (ms)	Observaciones
10	0,000609	Para un elemento cercano al final de la lista o que no se encuentra
100	0,004492	
500	0,018468	

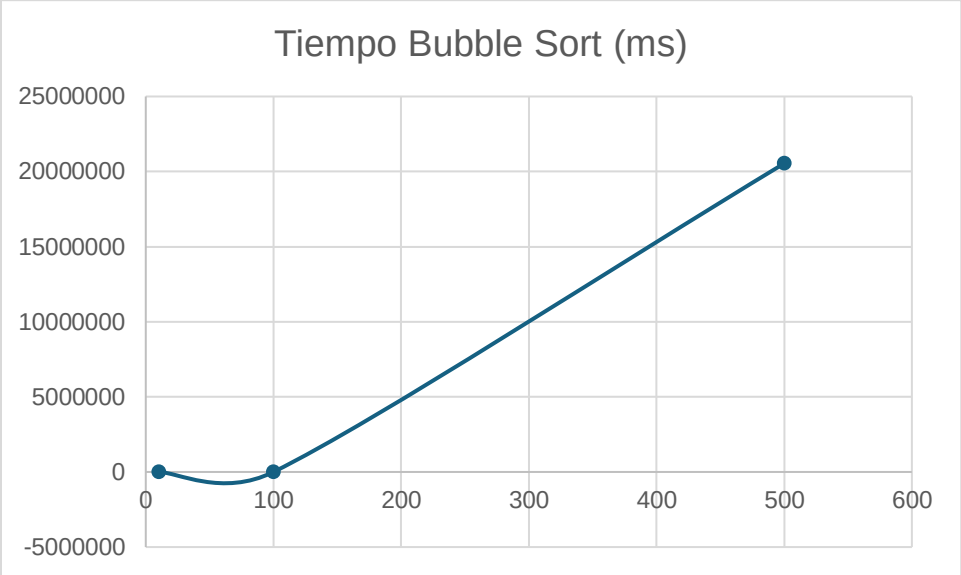


Tamaño Lista	Tiempo B Binaria (ms)	Observaciones
10	0,000768	Para un elemento cercano al final de la lista o que no se encuentra
100	0,00217	
500	0,00185	

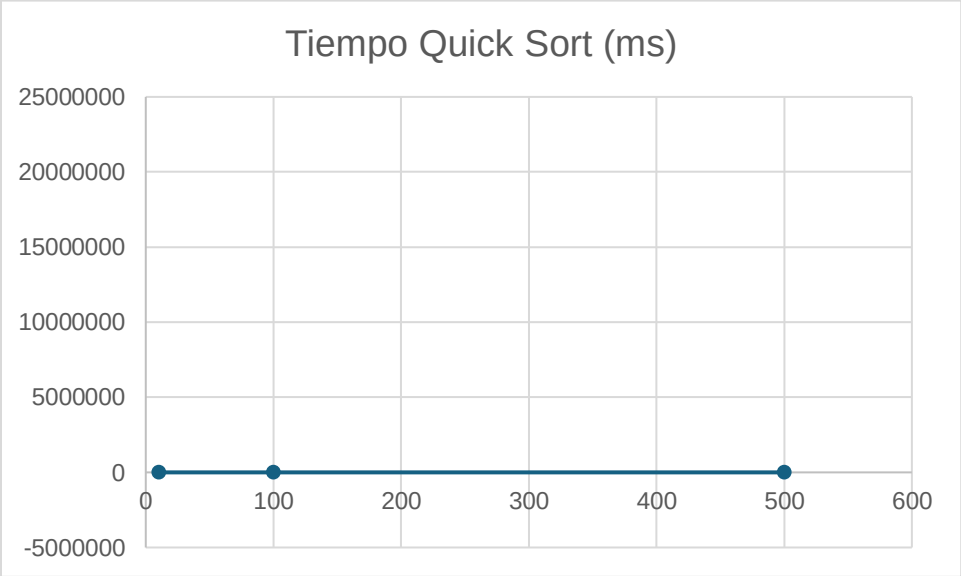


Comparación entre Bubble Sort y Quick Sort.

Tamaño Lista	Tiempo Bubble S (ms)
10	0,009382
100	0,768096
500	20.541.130



Tamaño Lista	Tiempo Quick S (ms)
10	0,011177
100	0,142636
500	0,94069



A través de los resultados obtenidos podemos deducir que:

- Se corrobora el pero caso O(n) para la búsqueda lineal, donde el elemento no se encuentra en la lista. Se observa claramente como el tiempo de ejecución aumenta a medida que aumenta el tamaño de la muestra, ya que se recorren todos los elementos de la lista.
- Se corrobora el mejor caso O(n) para la búsqueda lineal, donde el elemento se encuentra al inicio de la lista. Se observa como el tiempo es prácticamente el mismo para los distintos tamaños de muestra.
- Se corrobora el peor caso O(log n) para búsqueda binaria, donde el elemento no se encuentra en la lista. Se observa el aumento sostenido del tiempo de ejecución a medida que aumenta el tamaño de la muestra. De todas formas, aun siendo el peor caso, dicho aumento no es exageradamente significativo como en otros casos.
- Se corrobora el mejor caso O(1) para búsqueda binaria, donde el elemento se encuentra en la lista. Se observa como los tiempos se reducen respecto del peor caso, ya que el elemento se encuentra en la primera iteración.
- Se corrobora el peor caso $O(n^2)$ para Bubble Sort, donde la lista está en orden inverso. Se observa un aumento notorio del tiempo de ejecución al analizar listas de gran tamaño, ya que se genera el máximo número de comparaciones y de intercambios.
- Se corrobora el mejor caso $O(n^2)$ para Bubble Sort, donde la lista ya se encuentra ordenada. Como se mencionó anteriormente en este trabajo se utiliza el algoritmo sin optimización. Por lo cual se observa que los tiempos de ejecución son casi idénticos a los obtenidos en el peor caso. Esto se da porque el algoritmo recorre toda la lista y realiza todas las comparaciones ya que no tiene forma de saber que la lista ya está ordenada.
- Se corrobora el peor caso $O(n^2)$ para Quick Sort, donde el pivote es el mayor elemento en cada partición, por ejemplo en una lista ordenada. Al quedar las sublistas desbalanceadas, se observa un aumento exponencial del tiempo de ejecución debido a que la recursividad se aplica sobre una sola sublista que resulta conteniendo el total de los elementos (menos el pivote).
- Se corrobora el mejor caso O(n log n) para Quick Sort, donde el pivote divide la lista lo mas cercano a la mitad. Se observa que el tiempo de ejecución varia de forma ínfima a medida que aumenta el tamaño de la muestra. En este caso se debe a que, contrario al peor caso, las sublistas se generan de manera equilibrada.
- En la comparación llevada a cabo entre la búsqueda lineal y la búsqueda binaria se observa que:

- Para un elemento cercano al inicio de la lista la búsqueda lineal es más eficiente, dado que lo encuentra en la primera comparación. Por el contrario la búsqueda binaria debe realizar las divisiones y descartes hasta llegar al elemento buscado.
- Para un elemento cercano al final de la lista o que no se encuentra en ella la búsqueda lineal resulta considerablemente mas eficiente a medida que aumenta el tamaño de la muestra. Esto se da porque la búsqueda lineal debe recorrer la lista completa.

Considerando los comportamientos mencionados se recomienda el uso de búsqueda binaria, dado que resulta más eficiente en los casos en que se desconoce la posición del elemento buscado.

- En la comparación realizada entre Bubble Sort y Quick Sort se observa que para el primer algoritmo el tiempo aumenta considerablemente cuando mayor es el tamaño de la muestra. Mientras que para Quick Sort el tiempo se mantiene estable aunque aumente el tamaño de la muestra. Por lo tanto este última se recomiendo para ordenar grandes listas.

6.- Conclusiones

Durante la confección de este Trabajo Integrador se adquirieron los conocimientos necesarios para llevar a cabo el análisis de algoritmo de forma empírica, es decir, a través de la práctica.

Esto nos ayudará como futuros desarrolladores a entender la importancia que tiene conocer los diferentes algoritmos de búsqueda y ordenamiento para lograr desarrollar programas o aplicaciones que sean eficientes, escalables y precisas en la gestión de datos e información. Asimismo comprender como llevar a cabo estos análisis, nos permitirá saber elegir el procedimiento adecuado dependiendo el caso, ya que arrojan información verdaderamente útil sobre la eficiencia de los distintos algoritmos.

7.- Anexos

Repositorio GitHub (contiene capturas de pantalla y planilla de Excel con los datos recolectados): https://github.com/matiacade55/UTN-TUPaD-P1/tree/2f3a9e3cb49ba5d181fd6d589037021470c72dd7/TP-Integrador

Videos explicativos Youtube: https://www.youtube.com/watch?v=O7gwvLy695c

https://www.youtube.com/watch?v=xNid6g4H46A

Enlace al video de presentación: https://www.youtube.com/watch?v=UiqU_8FZ36c

Capturas de salidas por consola

```
---Bubble Sort para 10 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0002 - Nombre: arroz - Precio: $1200.5 - Vencimiento: 2026-01-15
Id: 0006 - Nombre: azucar - Precio: $1500.9 - Vencimiento: 2026-03-01
...
Id: 0003 - Nombre: leche - Precio: $1800.75 - Vencimiento: 2025-11-20
Id: 0004 - Nombre: pan - Precio: $900.25 - Vencimiento: 2025-11-10
Id: 0009 - Nombre: sal - Precio: $700.3 - Vencimiento: 2026-05-30
Tiempo promedio: 0.009382 ms
```

```
---Bubble Sort para 100 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0065 - Nombre: aceitunas - Precio: $1200.0 - Vencimiento: 2026-10-01
Id: 0044 - Nombre: acondicionador - Precio: $1700.0 - Vencimiento: 2026-06-15
...
Id: 0035 - Nombre: vino - Precio: $3000.0 - Vencimiento: 2026-05-01
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-10
Tiempo promedio: 0.768096 ms
```

```
---Bubble Sort para 500 productos---
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0172 - Nombre: aceite de bebe - Precio: $800.0 - Vencimiento: 2028-03-10
...
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-03
Id: 0300 - Nombre: zapatillas - Precio: $5000.0 - Vencimiento: 2029-12-10
Id: 0304 - Nombre: zapatos - Precio: $6000.0 - Vencimiento: 2029-12-30
Tiempo promedio: 20.541130 ms
```

```
---Bubble Sort para 10 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0002 - Nombre: arroz - Precio: $1200.5 - Vencimiento: 2026-01-15
Id: 0006 - Nombre: azucar - Precio: $1500.9 - Vencimiento: 2026-03-01
...
Id: 0003 - Nombre: leche - Precio: $1800.75 - Vencimiento: 2025-11-20
Id: 0004 - Nombre: pan - Precio: $900.25 - Vencimiento: 2025-11-10
Id: 0009 - Nombre: sal - Precio: $700.3 - Vencimiento: 2026-05-30
Tiempo promedio: 0.009695 ms
```

```
---Bubble Sort para 100 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0065 - Nombre: aceitunas - Precio: $1200.0 - Vencimiento: 2026-10-01
Id: 0044 - Nombre: acondicionador - Precio: $1700.0 - Vencimiento: 2026-06-15
...
Id: 0035 - Nombre: vino - Precio: $3000.0 - Vencimiento: 2026-05-01
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-10
Tiempo promedio: 0.707512 ms
```

```
---Bubble Sort para 500 productos---
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0172 - Nombre: aceite de bebe - Precio: $800.0 - Vencimiento: 2028-03-10
...
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-03
Id: 0300 - Nombre: zapatillas - Precio: $5000.0 - Vencimiento: 2029-12-10
Id: 0304 - Nombre: zapatos - Precio: $6000.0 - Vencimiento: 2029-12-30
Tiempo promedio: 18.042563 ms
```

```
---Bubble Sort para 10 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0002 - Nombre: arroz - Precio: $1200.5 - Vencimiento: 2026-01-15
Id: 0006 - Nombre: azucar - Precio: $1500.9 - Vencimiento: 2026-03-01
Id: 0003 - Nombre: leche - Precio: $1800.75 - Vencimiento: 2025-11-20
Id: 0004 - Nombre: pan - Precio: $900.25 - Vencimiento: 2025-11-10
Id: 0009 - Nombre: sal - Precio: $700.3 - Vencimiento: 2026-05-30
Tiempo promedio: 0.011437 ms
```

```
---Bubble Sort para 100 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0065 - Nombre: aceitunas - Precio: $1200.0 - Vencimiento: 2026-10-01
Id: 0044 - Nombre: acondicionador - Precio: $1700.0 - Vencimiento: 2026-06-15
Id: 0035 - Nombre: vino - Precio: $3000.0 - Vencimiento: 2026-05-01
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-10
Tiempo promedio: 0.680786 ms
```

```
---Bubble Sort para 500 productos---
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0172 - Nombre: aceite de bebe - Precio: $800.0 - Vencimiento: 2028-03-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-03
Id: 0300 - Nombre: zapatillas - Precio: $5000.0 - Vencimiento: 2029-12-10
Id: 0304 - Nombre: zapatos - Precio: $6000.0 - Vencimiento: 2029-12-30
Tiempo promedio: 18.916333 ms
```

```
Buscar producto por nombre: pan
---Búsqueda Binaria para 10 productos---
Id: 0004 - Nombre: pan - Precio: $900.25 - Vencimiento: 2025-11-10
Tiempo promedio: 0.000825 ms
```

```
---Búsqueda Binaria para 100 productos---
Id: 0049 - Nombre: lentejas - Precio: $1000.0 - Vencimiento: 2026-07-10
Tiempo promedio: 0.001534 ms
```

```
---Búsqueda Binaria para 500 productos---
Id: 0249 - Nombre: after shave - Precio: $850.0 - Vencimiento: 2029-03-30
Tiempo promedio: 0.001940 ms
```

```
Buscar producto por nombre: hola
---Búsqueda Binaria para 10 productos---
Producto no encontrado
Tiempo promedio: 0.001125 ms

---Búsqueda Binaria para 100 productos---
Producto no encontrado
Tiempo promedio: 0.001797 ms

---Búsqueda Binaria para 500 productos---
Producto no encontrado
Tiempo promedio: 0.002270 ms
```

```
Buscar producto por nombre: aceite
---Búsqueda Lineal para 10 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Tiempo promedio: 0.000273 ms

---Búsqueda Lineal para 100 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Tiempo promedio: 0.000282 ms

---Búsqueda Lineal para 500 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Tiempo promedio: 0.000290 ms
```

```
Buscar producto por nombre: hola
---Búsqueda Lineal para 10 productos---
Producto no encontrado
Tiempo promedio: 0.000639 ms

---Búsqueda Lineal para 100 productos---
Producto no encontrado
Tiempo promedio: 0.004512 ms

---Búsqueda Lineal para 500 productos---
Producto no encontrado
Tiempo promedio: 0.017467 ms
```

```
---Quick Sort para 10 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0002 - Nombre: arroz - Precio: $1200.5 - Vencimiento: 2026-01-15
...
Id: 0005 - Nombre: huevos - Precio: $2200.0 - Vencimiento: 2025-12-01
Id: 0003 - Nombre: leche - Precio: $1800.75 - Vencimiento: 2025-11-20
Id: 0003 - Nombre: leche - Precio: $1800.75 - Vencimiento: 2025-11-20
Tiempo promedio: 0.011177 ms

---Quick Sort para 100 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0065 - Nombre: aceitunas - Precio: $1200.0 - Vencimiento: 2026-10-01
...
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-10
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-10
Tiempo promedio: 0.142636 ms

---Quick Sort para 500 productos---
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
...
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-03
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-03
Id: 0300 - Nombre: zapatillas - Precio: $5000.0 - Vencimiento: 2029-12-10
Tiempo promedio: 0.940690 ms

---Quick Sort para 10 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0002 - Nombre: arroz - Precio: $1200.5 - Vencimiento: 2026-01-15
Id: 0002 - Nombre: arroz - Precio: $1200.5 - Vencimiento: 2026-01-15
...
Id: 0005 - Nombre: huevos - Precio: $2200.0 - Vencimiento: 2025-12-01
Id: 0003 - Nombre: leche - Precio: $1800.75 - Vencimiento: 2025-11-20
Id: 0004 - Nombre: pan - Precio: $900.25 - Vencimiento: 2025-11-10
Tiempo promedio: 0.007867 ms

---Quick Sort para 100 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0044 - Nombre: acondicionador - Precio: $1700.0 - Vencimiento: 2026-06-15
Id: 0044 - Nombre: acondicionador - Precio: $1700.0 - Vencimiento: 2026-06-15
...
Id: 0035 - Nombre: vino - Precio: $3000.0 - Vencimiento: 2026-05-01
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-10
Tiempo promedio: 0.123424 ms

---Quick Sort para 500 productos---
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
...
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-04
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-04
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-03
Tiempo promedio: 0.874460 ms
```

```
---Quick Sort para 10 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0002 - Nombre: arroz - Precio: $1200.5 - Vencimiento: 2026-01-15
Id: 0006 - Nombre: azucar - Precio: $1500.9 - Vencimiento: 2026-03-01
Id: 0003 - Nombre: leche - Precio: $1800.75 - Vencimiento: 2025-11-20
Id: 0004 - Nombre: pan - Precio: $900.25 - Vencimiento: 2025-11-10
Id: 0009 - Nombre: sal - Precio: $700.3 - Vencimiento: 2026-05-30
Tiempo promedio: 0.011820 ms

---Quick Sort para 100 productos---
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0065 - Nombre: aceitunas - Precio: $1200.0 - Vencimiento: 2026-10-01
Id: 0044 - Nombre: acondicionador - Precio: $1700.0 - Vencimiento: 2026-06-15
Id: 0035 - Nombre: vino - Precio: $3000.0 - Vencimiento: 2026-05-01
Id: 0031 - Nombre: yogur - Precio: $950.0 - Vencimiento: 2026-04-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-10
Tiempo promedio: 0.631385 ms

---Quick Sort para 500 productos---
Id: 0182 - Nombre: abrelatas - Precio: $500.0 - Vencimiento: 2028-04-29
Id: 0001 - Nombre: aceite - Precio: $2500.0 - Vencimiento: 2025-11-05
Id: 0172 - Nombre: aceite de bebe - Precio: $800.0 - Vencimiento: 2028-03-10
Id: 0019 - Nombre: zanahoria - Precio: $500.0 - Vencimiento: 2026-02-03
Id: 0300 - Nombre: zapatillas - Precio: $5000.0 - Vencimiento: 2029-12-10
Id: 0304 - Nombre: zapatos - Precio: $6000.0 - Vencimiento: 2029-12-30
Tiempo promedio: 15.942763 ms
```