UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

INGENIERÍA EN COMPUTACIÓN

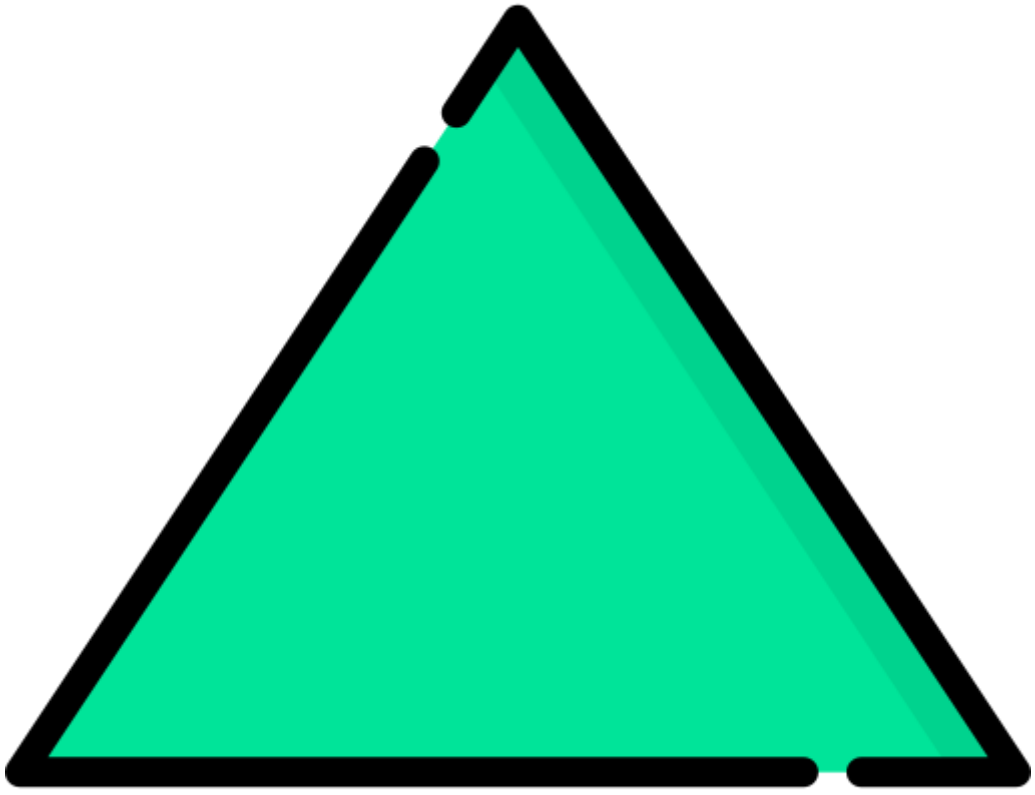
DISEÑO Y ANÁLISIS DE ALGORITMOS

GRUPO 1558 2026 – 1

PROFESOR JESÚS HERNÁNDEZ CABRERA

ALUMNO PEDRO CÉSAR JUÁREZ NÚÑEZ

PROYECTO FINAL



EJERCICIO 3

EXPLICACIÓN Y EJECUCIÓN

Se ejecuta un archivo llamado “archivo_ejecutable.py” el cual importa los métodos (algoritmos).

El objetivo del ejercicio fue desarrollar un programa capaz de medir y comparar el tiempo de ejecución de distintos algoritmos de ordenamiento vistos en clase. Los algoritmos evaluados fueron: Bubble Sort, Merge Sort, Selection Sort y Bogo Sort. Además, se debían realizar pruebas con listas de diferentes tamaños (10, 100, 1000, 10 000 y 100 000 elementos) y registrar los tiempos obtenidos en una tabla. Finalmente, se debía identificar la cota superior (Big-O) de cada algoritmo.

El programa inicia mostrando un menú donde el usuario elige qué archivo desea ordenar. Cada archivo contiene una cantidad distinta de números (10, 100, 1000, 10 000 o 100 000). Esta etapa permite observar cómo el tiempo de cada algoritmo cambia según el tamaño del problema.

Una vez seleccionado el archivo, el programa lee su contenido y convierte los valores en una lista de números enteros. Esta lista se utiliza como entrada para todos los algoritmos.

Para evaluar cada método de ordenamiento, el programa mide el tiempo que tarda en ordenar la lista original. Para ello se generan copias independientes de la lista, de modo que cada algoritmo reciba los mismos elementos sin alteraciones previas.

Tabla del Tiempo de ejecución (en segundos)

N	Bubble sort	Merge sort	Selection	Bogo sort (stupid sort)
10	1.9788742065429688e-05	4.7206878662109375e-05	2.0265579223632812e-05	1.3681271076202393
100	0.0005540847778320312	0.00026702880859375	0.0003910064697265625	-
1,000	0.0875091552734375	0.005067110061645508	0.06641197204589844	-
10,000	4.837161540985107	0.029291629791259766	2.709260940551758	-
100,000	787.600994348526	0.31217312812805176	250.20277571678162	-
O(¿?)	$O(n^2)$	$O(n \log n)$	$O(n^2)$	Promedio: $O(n * n!)$ Peor caso: $O(\infty)$

10 NÚMEROS:

```
¿Qué archivo quieres ordenar?
Diez números: ingresa "1" en consola
Cien números: ingresa "2" en consola
Mil números: ingresa "3" en consola
Diez mil números: ingresa "4" en consola
Cien mil números: ingresa "5" en consola
Ingresa el número: 1
BUBBLE SORT
     Tiempo de ejecución: 1.9788742065429688e-05 segundos
MERGE SORT
     Tiempo de ejecución: 4.7206878662109375e-05 segundos
SELECTION SORT
     Tiempo de ejecución: 2.0265579223632812e-05 segundos
BOGO SORT
     Intentos necesarios: 462374
     Lista ordenada: [1, 2, 3, 4, 4, 6, 8, 9, 10, 56]
     Tiempo de ejecución: 1.3681271076202393 segundos
```

CIEN NÚMEROS:

```
¿Qué archivo quieres ordenar?
Diez números: ingresa "1" en consola
Cien números: ingresa "2" en consola
Mil números: ingresa "3" en consola
Diez mil números: ingresa "4" en consola
Cien mil números: ingresa "5" en consola
Ingresa el número: 2
BUBBLE SORT
     Tiempo de ejecución: 0.0005540847778320312 segundos
MERGE SORT
     Tiempo de ejecución: 0.00026702880859375 segundos
SELECTION SORT
     Tiempo de ejecución: 0.0003910064697265625 segundos
BOGO SORT
     No es posible ejecutar el algoritmo Bogo Sort porque el número de elementos es muy grande
```

MIL NUMEROS:

```
¿Qué archivo quieres ordenar?
Diez números: ingresa "1" en consola
Cien números: ingresa "2" en consola
Mil números: ingresa "3" en consola
Diez mil números: ingresa "4" en consola
Cien mil números: ingresa "5" en consola
Ingresa el número: 3
BUBBLE SORT
     Tiempo de ejecución: 0.0875091552734375 segundos
MERGE SORT
     Tiempo de ejecución: 0.005067110061645508 segundos
SELECTION SORT
     Tiempo de ejecución: 0.06641197204589844 segundos
BOGO SORT
     No es posible ejecutar el algoritmo Bogo Sort porque el número de elementos es muy grande
```

DIEZ MIL NÚMEROS:

```
¿Qué archivo quieres ordenar?
Diez números: ingresa "1" en consola
Cien números: ingresa "2" en consola
Mil números: ingresa "3" en consola
Diez mil números: ingresa "4" en consola
Cien mil números: ingresa "5" en consola
Ingresa el número: 4
BUBBLE SORT
     Tiempo de ejecución: 4.837161540985107 segundos
MERGE SORT
     Tiempo de ejecución: 0.029291629791259766 segundos
SELECTION SORT
     Tiempo de ejecución: 2.709260940551758 segundos
BOGO SORT
     No es posible ejecutar el algoritmo Bogo Sort porque el número de elementos es muy grande
```

CIEN MIL NÚMEROS

```
¿Qué archivo quieres ordenar?
Diez números: ingresa "1" en consola
Cien números: ingresa "2" en consola
Mil números: ingresa "3" en consola
Diez mil números: ingresa "4" en consola
Cien mil números: ingresa "5" en consola
Ingresa el número: 5
BUBBLE SORT
     Tiempo de ejecución: 787.600994348526 segundos
MERGE SORT
     Tiempo de ejecución: 0.31217312812805176 segundos
SELECTION SORT
     Tiempo de ejecución: 250.20277571678162 segundos
BOGO SORT
     No es posible ejecutar el algoritmo Bogo Sort porque el número de elementos es muy grande
```

CONCLUSIÓN

Es increíble ver como merge sort es muy eficiente, fue muy rápido, se vio más su eficiencia con el archivo de cien mil números. Esto ocurre ya que siempre garantiza O(n log n).

Bubble Sort y Selection Sort son simples de implementar, pero su rendimiento es pobre para listas grandes debido a su complejidad $O(n^2)$.

Bogo Sort no es práctico para ningún caso real. Su complejidad (n!) crece tan rápido que incluso con 10 – 12 elementos pueden tardar muchísimo. Solo se usa con fines educativos o humorísticos.

EJERCICIO 4

EXPLICACIÓN Y JUSTIFICACIÓN

FUERZA BRUTA

Se ejecuta el archivo “código_ejecutable_fuerza_bruta.py”.

Es un código muy corto y sencillo, sin embargo, muy poco eficiente ya que su complejidad es de:

$$O(n^3)$$

Cuya finalidad es probar todas las combinaciones posibles de 3 puntos para encontrar el triángulo de mayor área.

Los tres bucles juntos generan todas las combinaciones posibles de triples (i, j, k), lo que matemáticamente es equivalente a:

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$$

Este código no es recomendable porque es demasiado lento cuando la cantidad de puntos es grande.

- Con 500 puntos (nuestro caso) → ~20 millones de triángulos (muy lento)
- Con 1 000 puntos → ~166 millones de triángulos
- Con 10 000 puntos → ~166 000 millones de triángulos (inviable)

Para hallar el triángulo de mayor área en un conjunto de puntos existen soluciones mejores como usar el casco convexo y técnicas como "rotating calipers".

MEJORA (casco convexo y rotating calipers)

Se ejecuta el archivo “código_ejecutable_optimizado.py”.

El triángulo de mayor área siempre está formado por puntos que forman parte del casco convexo del conjunto. Por lo tanto, hacemos lo siguiente:

- Calculamos el casco convexo de los puntos (método de Andrew Monotone).
- A los puntos del casco (muchos menos que 500), les aplicamos el método de los tres punteros (pointer calipers) para encontrar el área máxima de un triángulo.

¿Qué es el casco convexo?

Es la figura más pequeña convexa que contiene todos los puntos. Estos puntos que forman parte del casco son importantes porque el triángulo de mayor área siempre estará formado por 3 puntos del casco convexo (puntos más al extremo, más a la orilla). Nunca por puntos que están en el interior.

¿Cómo funciona el método de Andrew Monotone?

Es el algoritmo más sencillo para obtener el casco convexo.

PASOS:

1. Ordena todos los puntos por: “x” de menor a mayor y, si hay empate, por “y”
2. Se construye la “parte inferior” del casco
    a. Se empieza con una lista vacía (lower).
    b. Se recorre los puntos ordenados de izquierda a derecha
    c. En cada paso:
        i. Se mete el punto
        ii. Pero si en lower los últimos tres puntos forman una vuelta derecha, se elimina del del centro (para mantener la figura convexa). Esto evita hendiduras.
3. Se construye la “parte superior” del casco.
    a. Lo mismo que el punto 2 pero ahora se recorre los puntos de derecha a izquierda.
        i. Se crea una lista upper
        ii. Se agrega los puntos
        iii. Si hay una vuelta derecha, se elimina el punto del medio.
4. Se una la parte superior + inferior
    a. El casco convexo es lower + upper, quitando el primer y último punto de upper porque se repiten.

¿Cuándo es una vuelta a la derecha y vuelta a la izquierda?

- Se tiene tres puntos: P1, P2 y P3
- Queremos saber si al ir de P1 -> P2 -> P3 la trayectoria:
    - Gira la izquierda
    - Gira a la derecha
    - O es línea recta

Se calcula con el “producto cruzado 2D”:

$$vuelta = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$$

Resultado de vuelta	Significado
> 0	Vuelta izquierda
< 0	Vuelta derecha
= 0	Línea recta

Estas vueltas son importantes porque cuando se construye el casco:

- Vuelta izquierda -> la forma se mantiene convexa (se deja el punto).
- Vuelta derecha -> se forma una hendidura (eliminamos el punto medio).

¿Qué es y cómo funciona el método de los tres punteros?

Es una técnica para recorrer puntos del casco convexo sin revisar todas las combinaciones posibles.

Lo que hace:

- Se coloca tres punteros en puntos del casco.
- Se va rotando estos punteros alrededor del casco como si fueran las manecillas del reloj.
- Ésto permite encontrar el triángulo de mayor área.

Complejidad

La combinación de:

1. Construir el casco convexo usando Andrew Monotone Chain, y
2. Aplicar Rotating Calipers sobre el casco,

tiene una complejidad total de:

$$O(n \log n)$$

¿Por qué?

El método Andrew Monotone Chain ordena todos los puntos por coordenada x (y como desempate) y esta ordenación cuesta O(n log n). La construcción del casco (el barrido inferior y superior) es lineal: O(n), lo que al final es O(n log n).

El método rotating calipers es de complejidad O(h) ya que se ejecuta sobre los h puntos del casco convexo. No revisita puntos. Esta complejidad se suma con la del método Monotone Chain y se desprecia al mismo tiempo.

Explicación de los archivos

Dentro del archivo “_01_leer_coordenadas.py” tenemos el método que permite leer el archivo campo.in y extraer su contenido.

En el archivo “_02_casco_convexo.py” se tiene el método que ejecuta el método de Andrew Monotone. Hay otros algoritmos que permiten encontrar el casco convexo, pero éste es de los más sencillos y mejores de implementar.

El archivo “_03_tres_punteros.py” contiene el método que recorre los puntos del casco convexo y busca aquellos que el área sea el mayor.

Todo esto se ejecuta en el archivo “codigo_optimizado.py” en el que se importan todos los demás métodos, esto para una mejor organización del código.

EJECUCIÓN

POR FUERZA BRUTA:

```
PS C:\Users\kit21\Documents\5to semestre\algoritmos\tarea_final>
Python/Python313/python.exe "c:/Users/kit21/Documents/5to semestr
ejecutable_fuerza_bruta.py"
archivo campo.out creado con éxito por fuerza bruta
     Tiempo de ejecución: 11.271148443222046 segundos
PS C:\Users\kit21\Documents\5to semestre\algoritmos\tarea_final>
```

POR EL MÉTODO ANDREW MONOTONE Y POINTER CALIPERS:

```
Las coordenadas del triángulo es: ((1, 22), (996, 26), (212, 995))
El área es de: 483645.5

archivo campo.out creado con éxito con optimización
```

CONCLUSIÓN

Intentar resolver este problema por fuerza bruta resulta muy complejo cuando se intenta muchas coordenadas.

Aunque el método de fuerza bruta es sencillo de implementar y conceptualmente fácil de entender, su complejidad $O(n^3)$ lo vuelve completamente impráctico cuando el número de puntos crece, llegando a requerir millones o incluso miles de millones de operaciones. Esto lo hace inviable para cualquier aplicación real.

En contraste, la solución optimizada basada en el casco convexo y los rotating calipers reduce drásticamente el tiempo de cómputo a una complejidad $O(n \log n)$, permitiendo procesar cientos o miles de puntos de forma eficaz. El uso del casco convexo garantiza que solo se analizan los puntos relevantes —aquellos en el borde exterior— y el método de los tres punteros permite buscar el triángulo de mayor área sin revisar todas las combinaciones posibles.

En conclusión, la versión optimizada no solo es más rápida, sino que también es la única realmente adecuada para conjuntos de datos grandes, mientras que el enfoque de fuerza bruta solo es útil como referencia teórica o para casos muy pequeños.

BIBLIOGRAFÍA

PROBLEMA 3

Adolfo Neto. (2023, mayo 5). *Bogosort: The Stupid Sorting Algorithm*. DEV Community. https://dev.to/adolfont/bogosort-the-stupid-sorting-algorithm-168f

GeeksforGeeks. (2025, 3 octubre). *Merge Sort*. Recuperado de https://www.geeksforgeeks.org/dsa/merge-sort/

El Libro De Python. (s. f.). *Bubble Sort en Python*. Recuperado de https://ellibrodepython.com/bubble-sort

Mentores Tech. (s. f.). *Selection Sort*. Recuperado de https://www.mentorestech.com/resource-algorithms-selection-sort.php

PROBLEMA 4

The MathWorks, Inc. (s. f.). *Computing the convex hull*. MATLAB & Simulink Documentation. https://www.mathworks.com/help/matlab/math/computing-the-convex-hull.html

GeeksforGeeks. (s. f.). *Convex Hull Algorithm*. https://www.geeksforgeeks.org/dsa/convex-hull-algorithm/

CP-Algorithms. (s. f.). *Convex Hull*. https://cp-algorithms.com/geometry/convex-hull.html

GeeksforGeeks. (2025, 11 julio). *Convex Hull | Monotone chain algorithm*. https://www.geeksforgeeks.org/dsa/convex-hull-monotone-chain-algorithm/

alaneos777. (s. f.). *Rotating calipers*. HackMD. https://hackmd.io/@alaneos777/Bk3mt8P8d