Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 2 y 3
Cerradura Convexa

Integrante: Nicolás Escobar Zarzar

Profesor: Nancy Hitschfeld K.

Ayudantes: Catalina Gajardo
Gustavo Medel

Fecha de entrega: 1 de Julio de 2025
Santiago, Chile

Implementación y supuestos

La máquina utilizada para los tests posee las siguientes características relevantes:

- CPU: Intel Core i5 10400 @2.90 GHz, Turbo 4.00 GHz (6 núcleos físicos)
- RAM: 16 GB DDR4 a 2666 MT/s
- OS: Windows 11

Para los algoritmos desarrollados se intentó hacer una solución lo más general posible que funcione tanto para números enteros como reales, sin embargo, en la práctica, los algoritmos implementados en esta tarea solo funcionan de manera confiable con tipos de datos `long long` o `double` que representan números enteros y "reales" (de punto flotante) de 64 bits respectivamente. Esto es debido a la forma en que se generan los puntos para probar los algoritmos y el método utilizado para formar la cerradura. Ambos algoritmos hacen uso de la operación `productoCruz` de la clase vector para determinar qué puntos forman parte de la cerradura, esta operación es

$$A \times B = x_A y_B - x_B y_A \tag{1}$$

Donde A y B son vectores resultado de la resta entre un punto de referencia p y otros dos puntos a y b de los cuales se desea determinar cual forma parte de la cerradura. Esta operación involucra dos multiplicaciones y una sustracción, lo cual puede ser problemático tanto para números enteros como flotantes dependiendo de cuales sean los límites establecidos para estos valores, habiendo posibilidad de *overflow* o *underflow* ya sea en la multiplicación como en la resta, y en el caso de los números de punto flotante, la multiplicación inserta errores de redondeo que pueden dar resultados erróneos al intentar determinar la colinealidad de los puntos. Los valores de las coordenadas x e y de los puntos están dentro de límites preestablecidos en los archivos presentes en `src/performance_testers` y sus subdirectorios, los cuales van de -200000 a 200000 para los tests con datos de 64 bits, estos límites son bastantes generosos para evitar cualquier tipo de desborde y son lo suficientemente lejanos (en general) para prevenir una colinealidad excesiva de los puntos. Mientras más lejos puedan estar los puntos entre sí, es menos probable que estos sean colineales.

Para el caso del tipo `int`, los límites deben estar entre -16383 y 16383 (es decir $\pm 2^{14} - 1$) para prevenir desbordamiento, sin embargo, debido a lo acotado que son estos límites, el conjunto termina con demasiados puntos colineales lo que induce errores en los algoritmos en muchos casos. En el caso del tipo de dato `float`, el error acumulado en las multiplicaciones es lo suficientemente "malo" como para dar resultados erróneos casi siempre.

Además de *GiftWrapping*, el algoritmo de orden $O(n \log n)$ implementado fue el algoritmo *Divide and Conquer*, este algoritmo es particularmente poco robusto ante casos degenerados (muchos tripletes de puntos colineales), por lo que es crucial limitar la cantidad de tripletes de puntos colineales dentro de lo posible. Igualmente se hace uso de ciertos "trucos", para manejar casos de puntos colineales en ambos algoritmos. En el caso del algoritmo *GiftWrapping*, si se encuentra un punto colineal con el mejor punto candidato a ser parte de la cerradura convexa, se decide cual escoger basado en la distancia desde el punto de referencia (el último punto que si es parte de la cerradura) y cada punto candidato, prefiriendo siempre el que esté más lejos. Para evitar cálculos computacionalmente caros (como calcular una raíz cuadrada), esta distancia se determina con la operación `magnitud2` de un vector, que corresponde a llamar `productoPunto` consigo mismo, lo qué es equivalente al cuadrado de su longitud, valor suficiente para desambiguar cual punto colineal utilizar.

$$A \cdot A = \|A\|^2 \tag{2}$$

Para el caso de *Divide and Conquer*, cuando se hace la unión de las sub cerraduras, al momento de elegir el punto más a la derecha de la cerradura izquierda, también se revisa si se han encontrado múltiples puntos con la misma coordenada x que estén "más a la derecha", y se guarda un punto "derecho superior" y "derecho inferior", puntos colineales en x con la mayor y menor coordenada y respectivamente, esto es análogo para el punto más a la izquierda de la cerradura derecha. Estos puntos

superiores e inferiores se usan como base para determinar la tangente superior e inferior que unirá ambas cerraduras convexas.

Para la generación de puntos con una cantidad configurable de puntos en la cerradura según un porcentaje se hace uso del generador en `src/generators/hull_percentage_strategy.cpp`, el cual genera puntos en un círculo, asegurándose de generar primero aquellos puntos en la circunferencia (según el parámetro de porcentaje entregado), y luego los puntos interiores, los cuales tienen un límite de coordenadas ligeramente desplazado para que siempre estén estrictamente dentro del círculo. Si bien esta forma de generar puntos funciona de manera correcta para asegurar que la cerradura convexa del conjunto será la misma circunferencia exterior, el radio del círculo debe ser lo suficientemente grande para que los puntos que describen la circunferencia no sean demasiado colineales, en caso contrario, es imposible generar una cerradura convexa de esta forma sin puntos colineales en un tiempo razonable, ya que si no se tiene un patrón como el de este círculo habría que determinar la no colinealidad de cada triplete de puntos en el conjunto, el cual es un problema 3-SUM hard y no es factible de resolver para conjuntos de puntos grandes[1]. Para generar puntos colineales en la cerradura a propósito, se insertan puntos muy cerca uno del otro en el lado derecho de la circunferencia, tantos como el usuario solicite, para los tests se utilizaron los valores 0, 3, y 10. Este método de generación es notablemente malo para puntos de tipo `int` o `float` llevando a errores en todos los casos para el tipo `float`, y para el tipo `int` comienza a haber inconsistencias una vez que el porcentaje de puntos en la cerradura llega al 10%.

Además del método previamente mencionado, también se tiene un método que genera los puntos de manera completamente aleatoria restringidos dentro de un cuadrado y otro que también es aleatorio, pero se asegura de que haya al menos un triplete de puntos colineales en el conjunto, en cualquier parte.

Todos los tests de la sección siguiente fueron ejecutados con una semilla fija con valor 123 en un generador *mersenne twister* con distribuciones uniformes que varían según el tipo de dato como se mencionó antes, cada test se ejecutó 5 veces por cada tamaño y tipo de dato en los tests aleatorios y en los tests en un conjunto delimitado por un círculo se ejecutaron 5 veces por cada combinación única de `(tamaño del conjunto, tipo de dato, porcentaje en cerradura, cantidad de colineales)`

[1]Determinar la no colinealidad de todos los tripletes de puntos posibles en un conjunto de 10.000 puntos con coordenadas enteras toma aproximadamente 8 horas en un conjunto que efectivamente cumple la no colinealidad de cada triplete en el computador utilizado.

Resultados

A continuación se muestran como gráficos los resultados de ejecutar ambos algoritmos de cerradura convexa, *GiftWrapping* y *Divide and Conquer* en conjuntos de puntos que varían desde 10^4 hasta 10^6.

Puntos aleatorios:

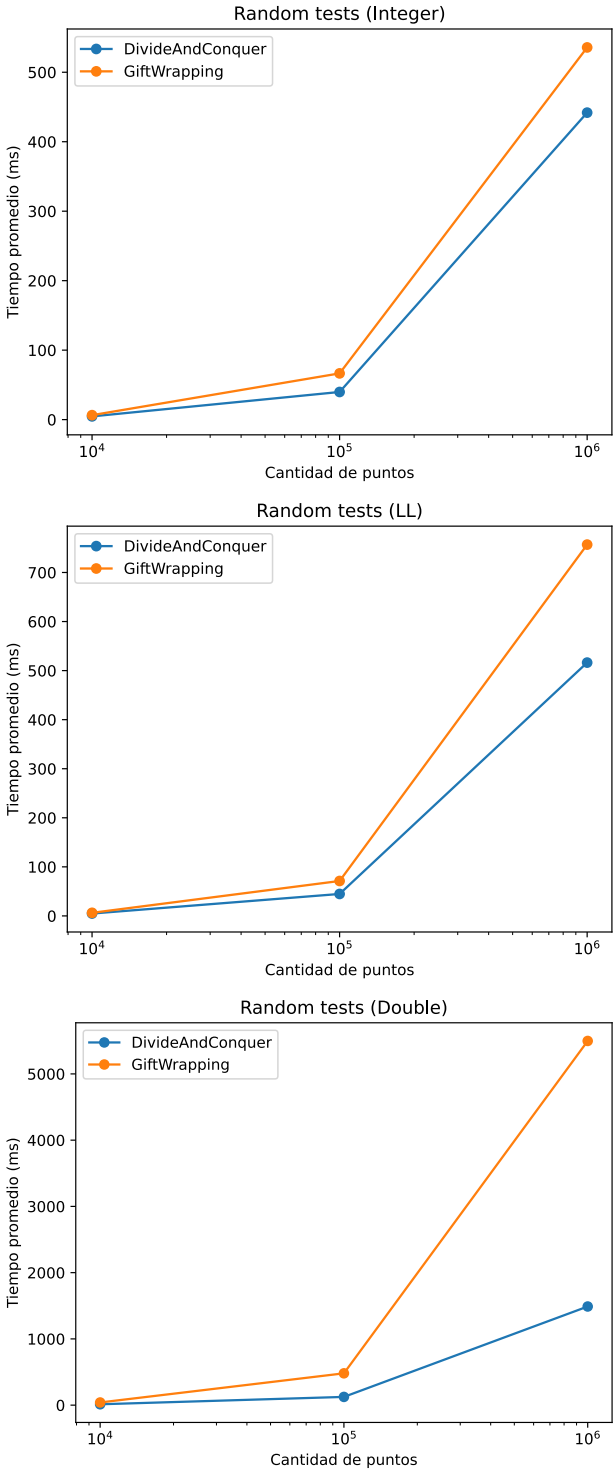


Figura 1: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos para puntos completamente aleatorios

Puntos con al menos 3 puntos colineales (en cualquier parte):

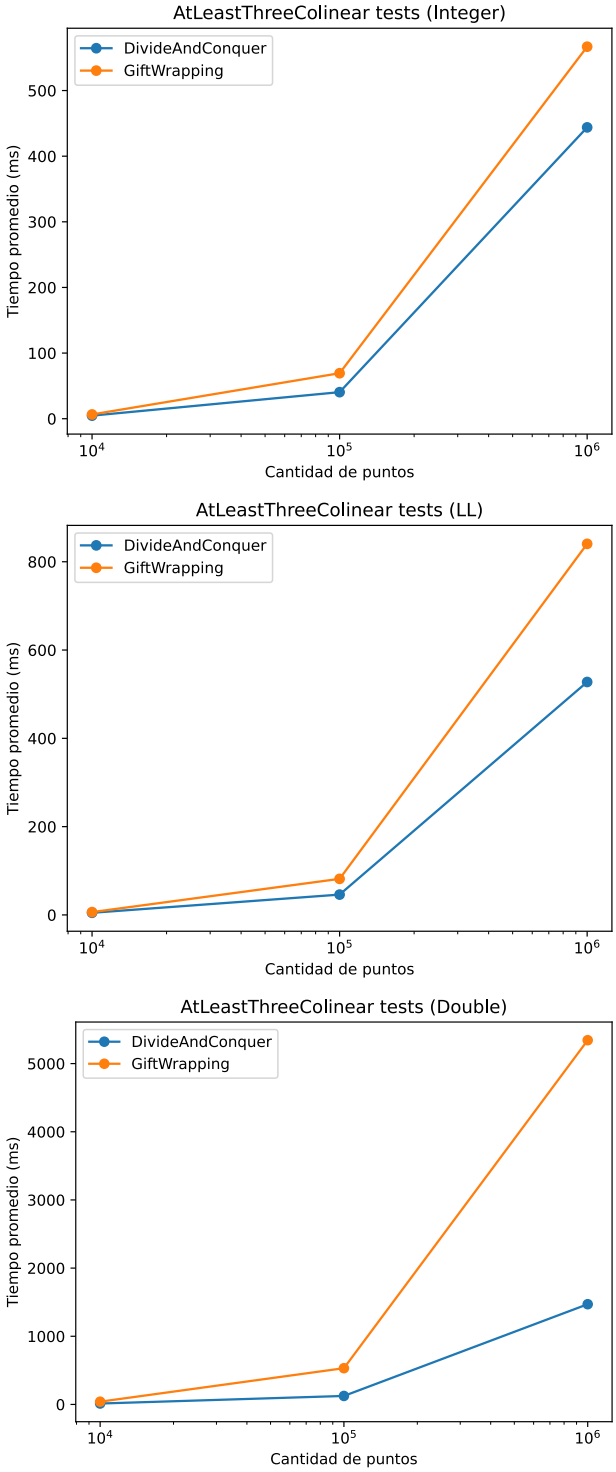


Figura 2: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos para puntos aleatorios con al menos 3 colineales

Puntos aleatorios en un círculo donde el 0.01% de los puntos forman parte de la cerradura convexa:

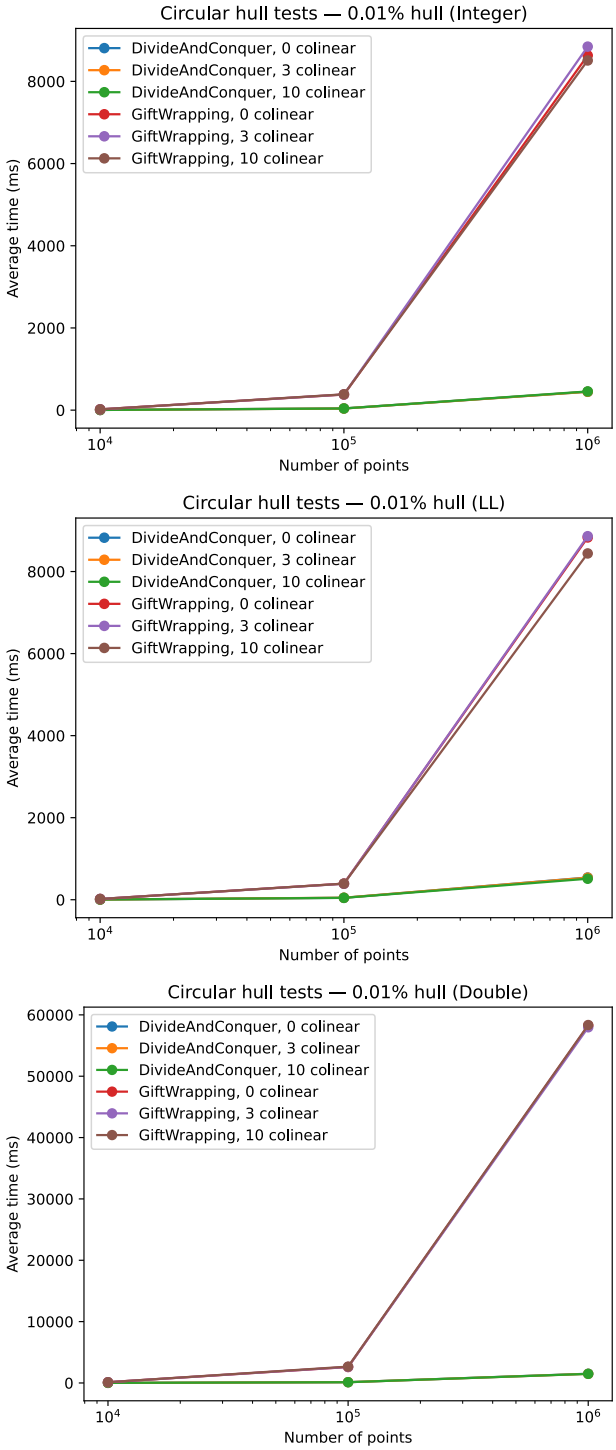


Figura 3: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 0.01% es la cerradura convexa

Mismos gráficos separados por algoritmo:

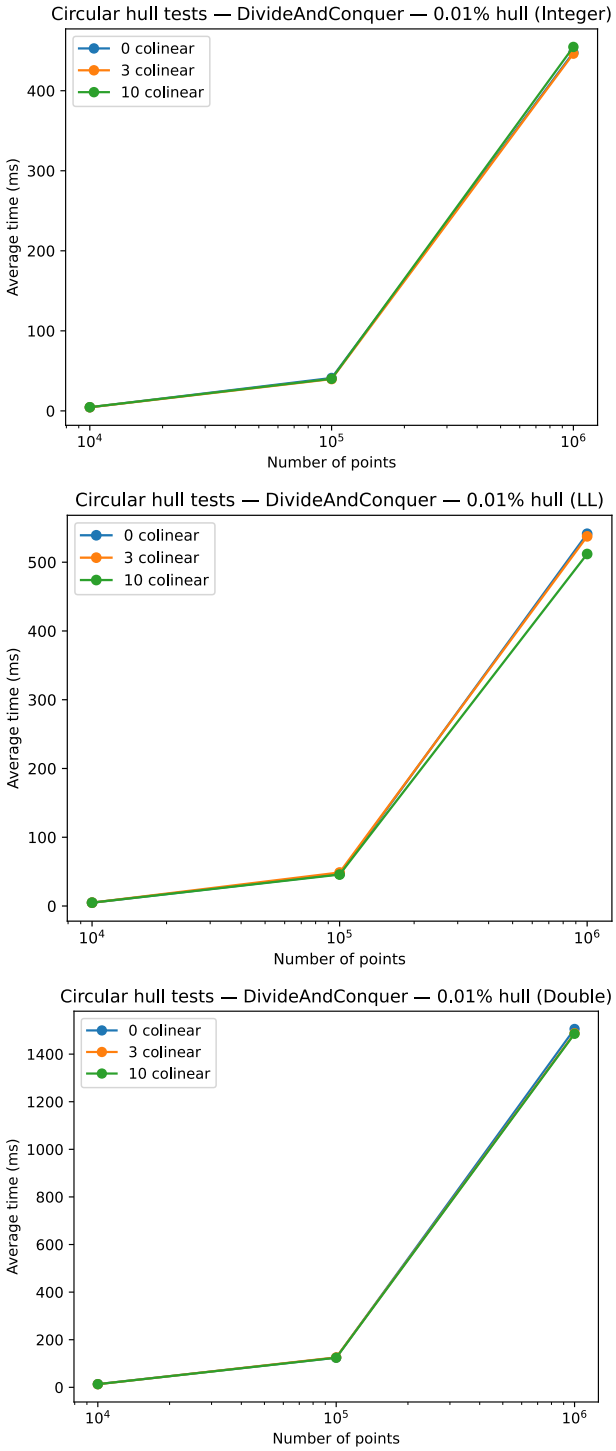


Figura 4: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 0.01% es la cerradura convexa

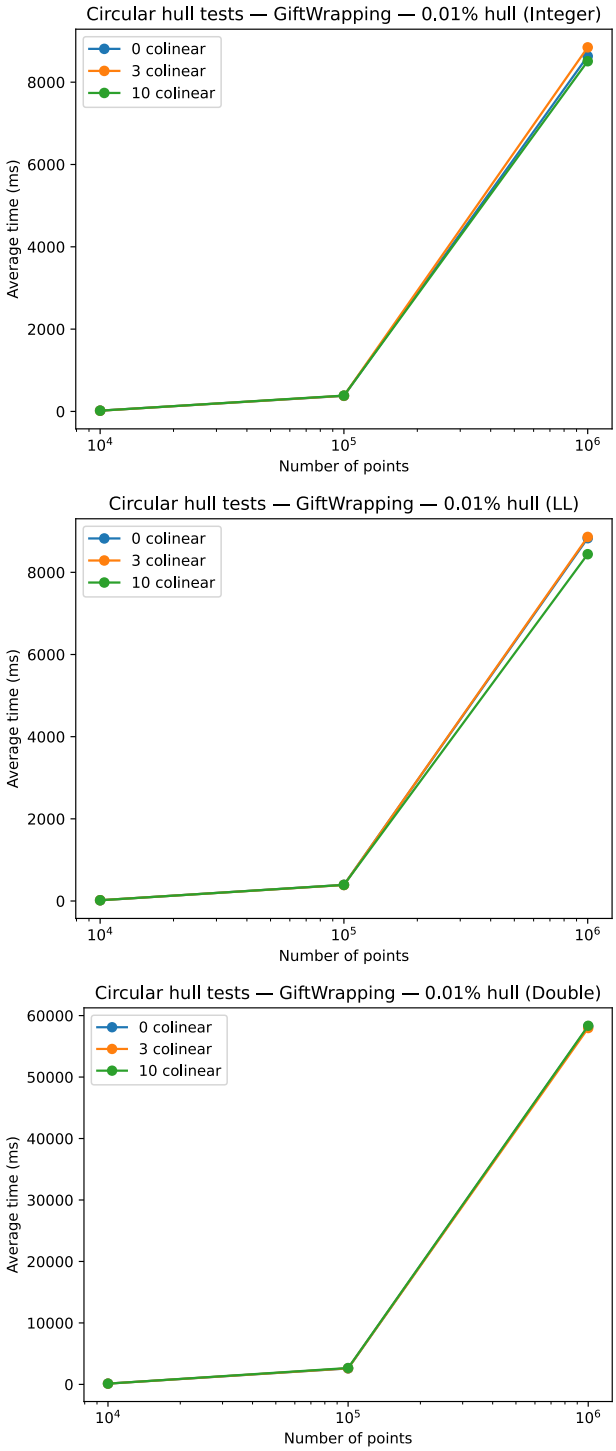


Figura 5: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 0.01% es la cerradura convexa

Puntos aleatorios en un círculo donde el 0.1% de los puntos forman parte de la cerradura convexa:

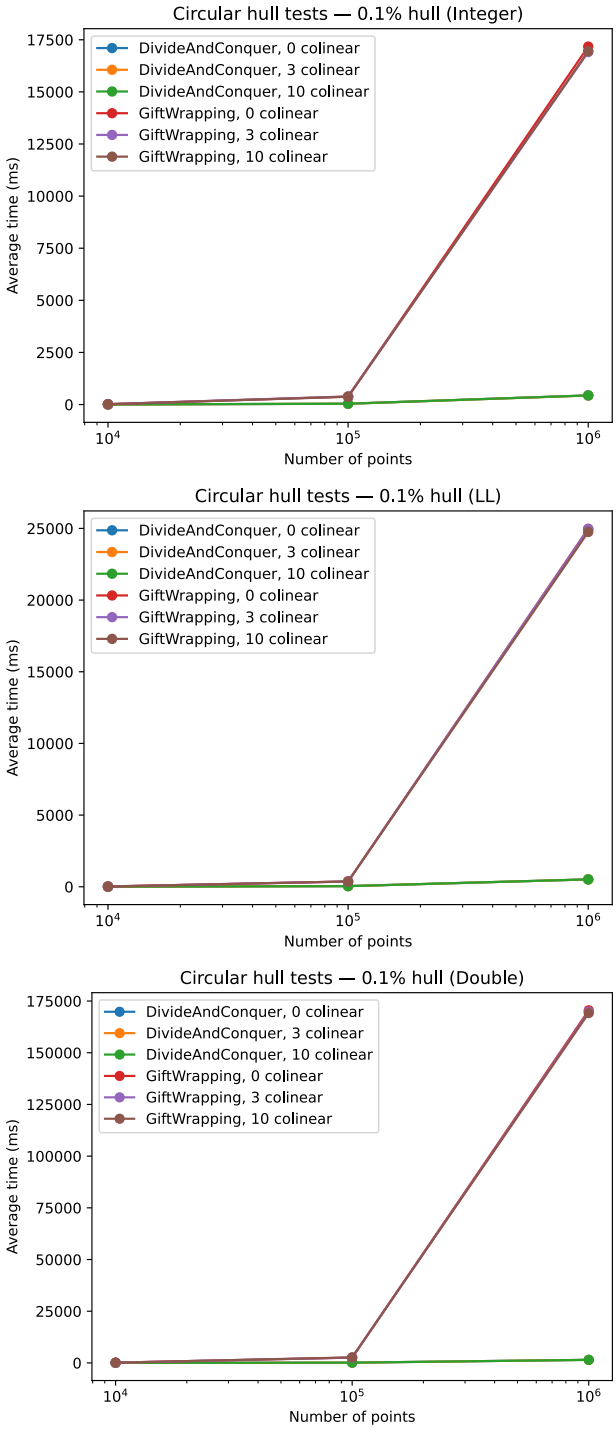


Figura 6: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 0.1% es la cerradura convexa

Mismos gráficos separados por algoritmo:

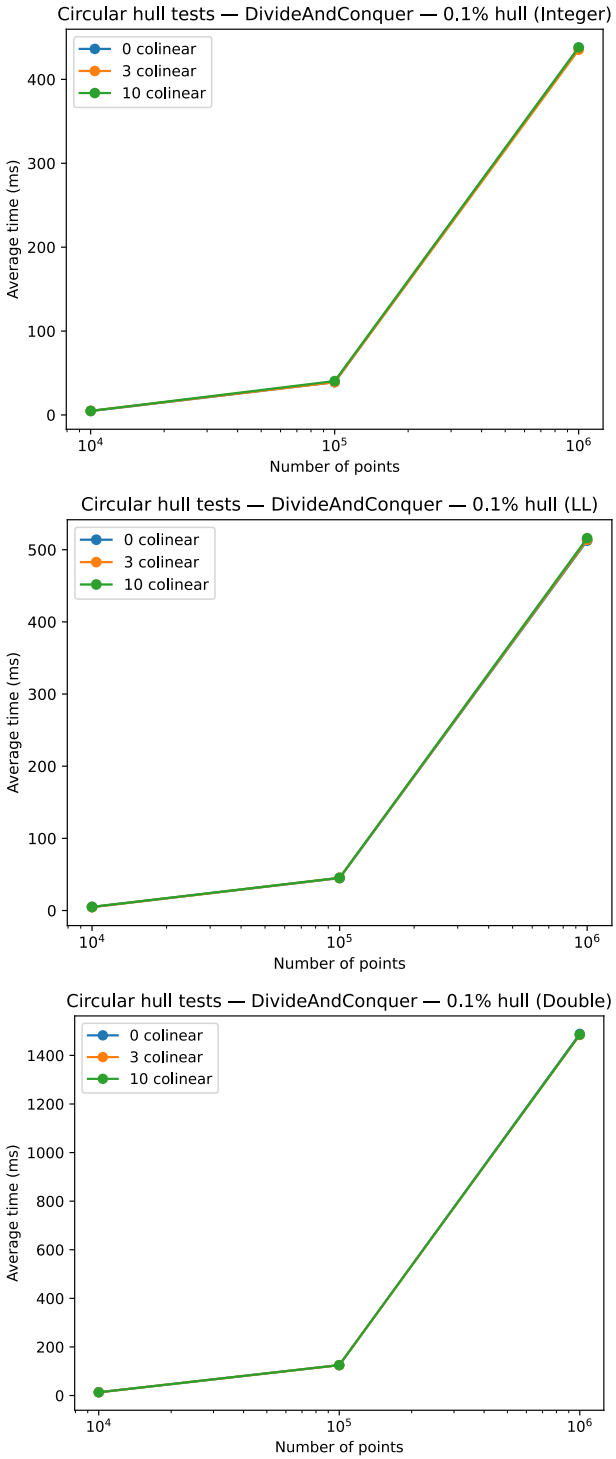


Figura 7: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 0.1% es la cerradura convexa

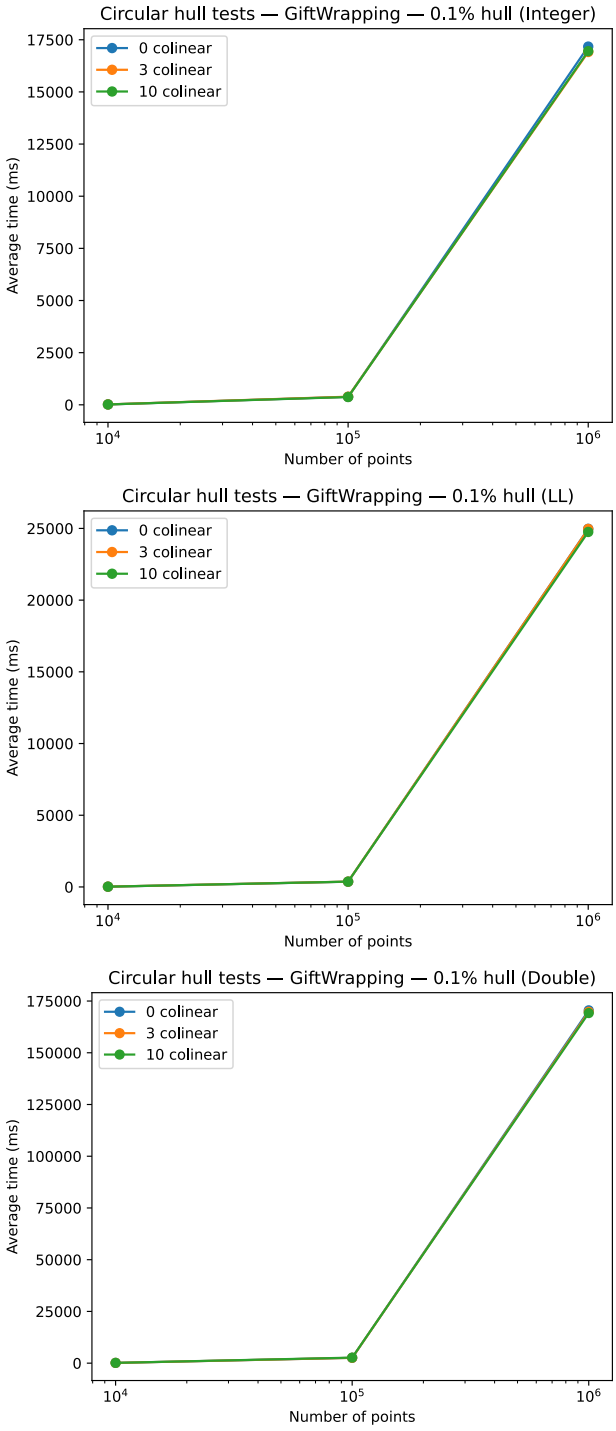


Figura 8: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 0.1% es la cerradura convexa

Puntos aleatorios en un círculo donde el 1.0% de los puntos forman parte de la cerradura convexa:

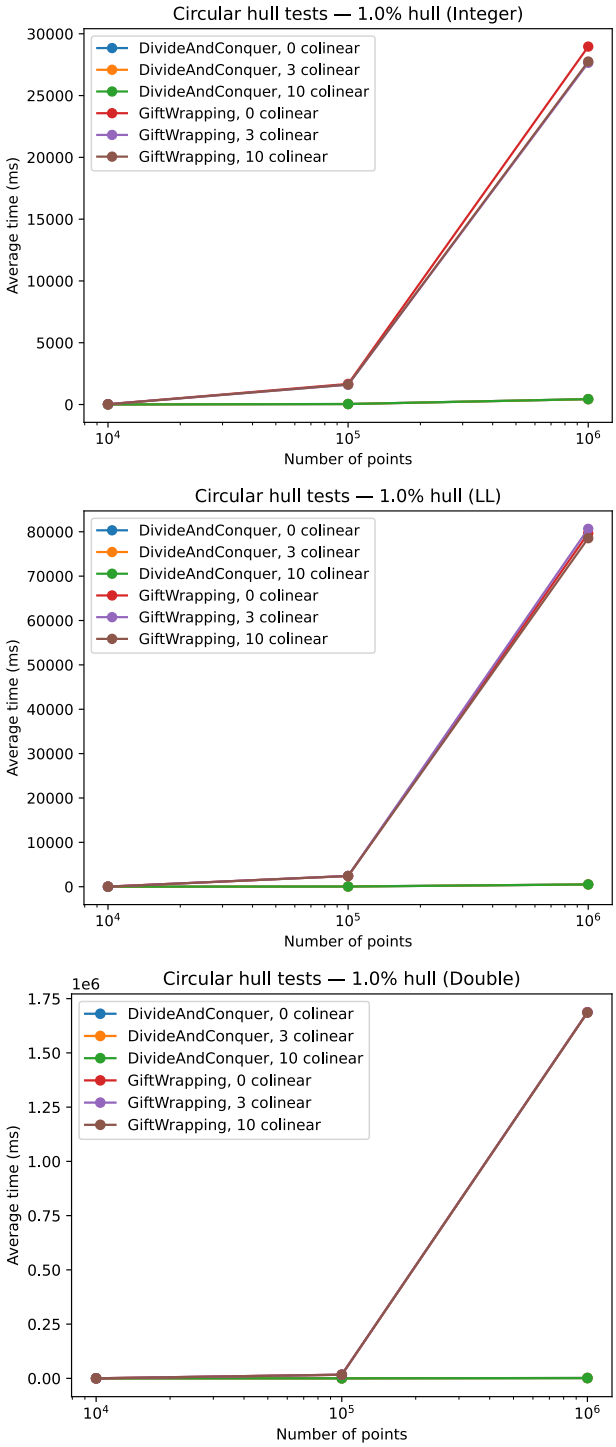


Figura 9: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 1.0% es la cerradura convexa

Mismos gráficos separados por algoritmo:

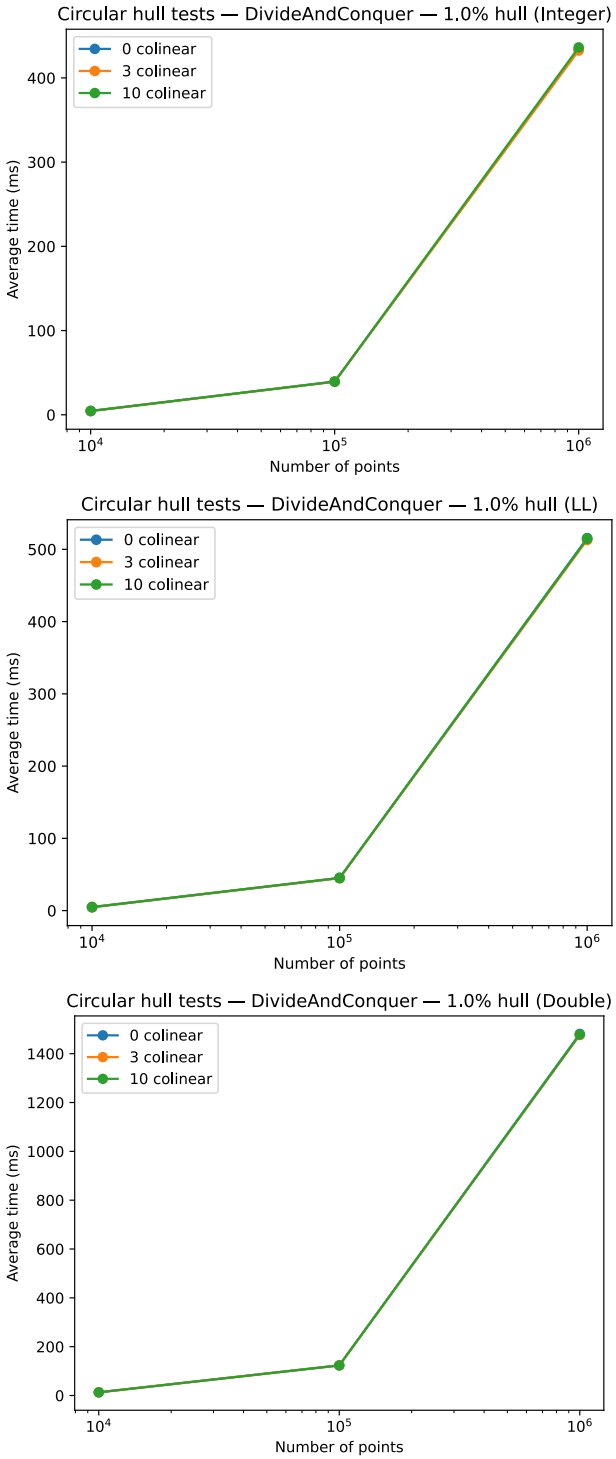


Figura 10: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 1.0% es la cerradura convexa

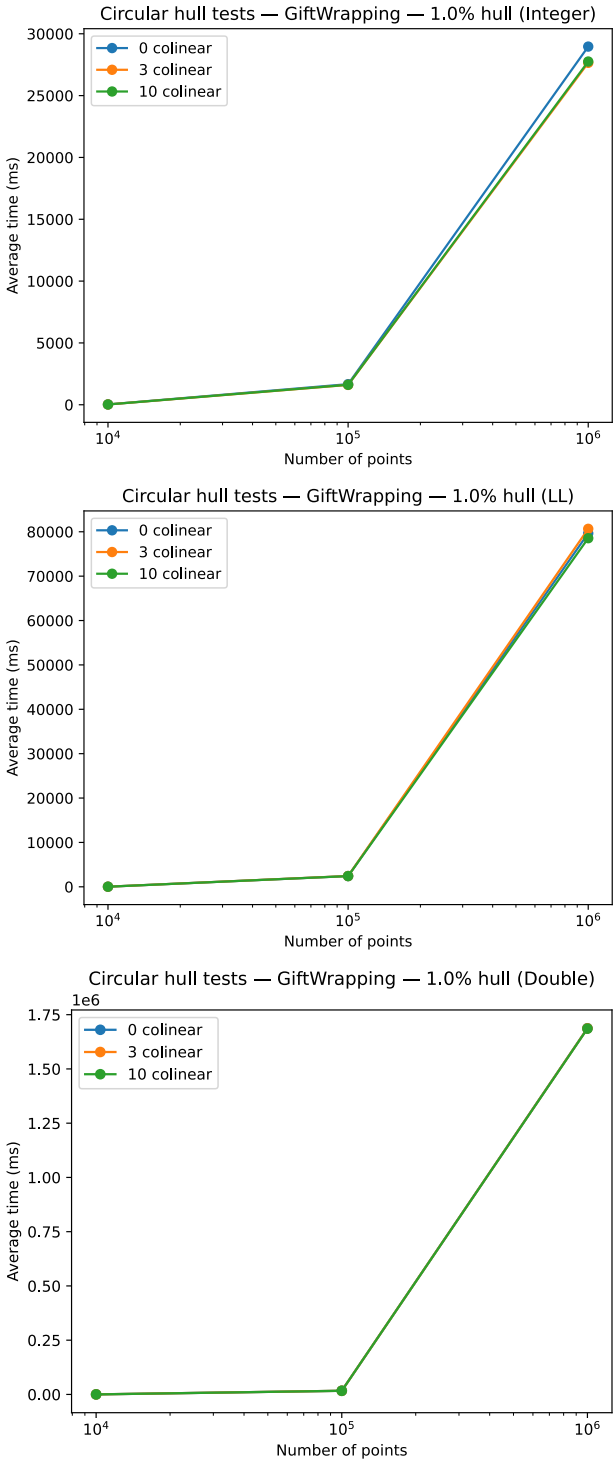
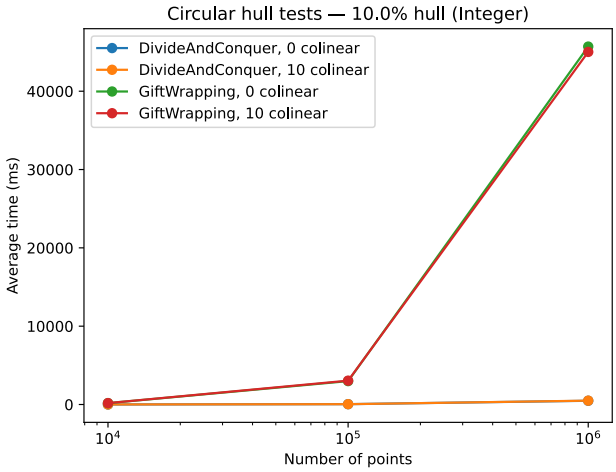
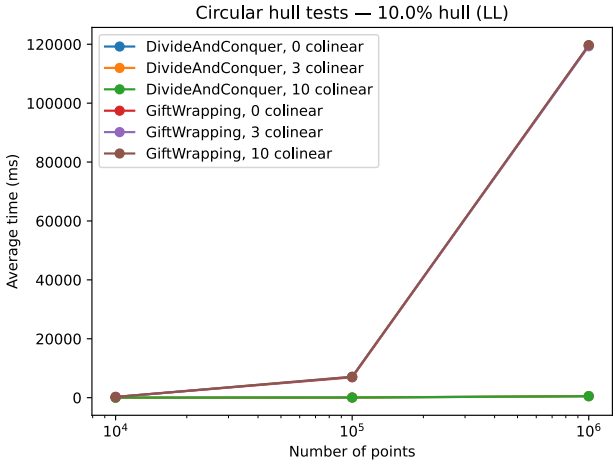


Figura 11: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 1.0% es la cerradura convexa

Puntos aleatorios en un círculo donde el 10.0% de los puntos forman parte de la cerradura convexa (A partir de aquí, los algoritmos dejan de funcionar del todo con enteros de 32 bits):





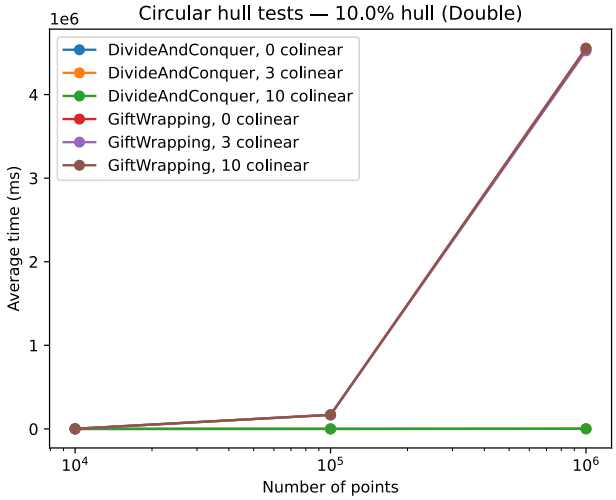


Figura 12: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 10.0% es la cerradura convexa

Mismos gráficos separados por algoritmo:

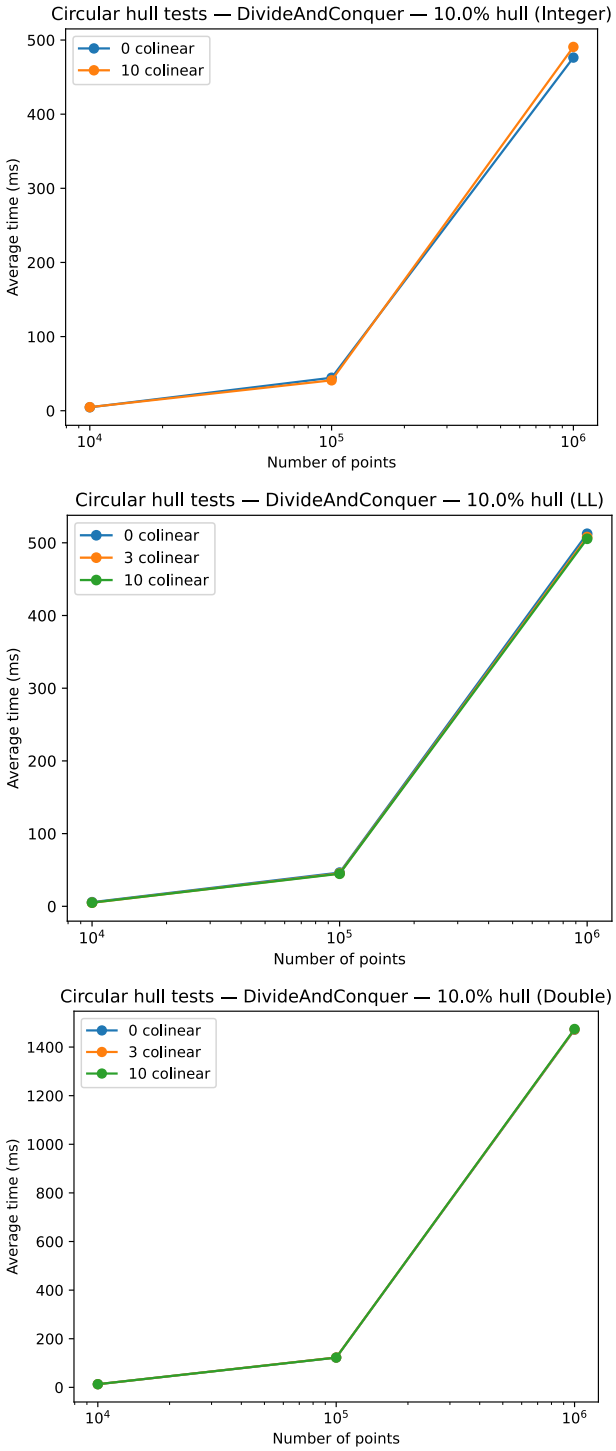


Figura 13: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 10.0% es la cerradura convexa

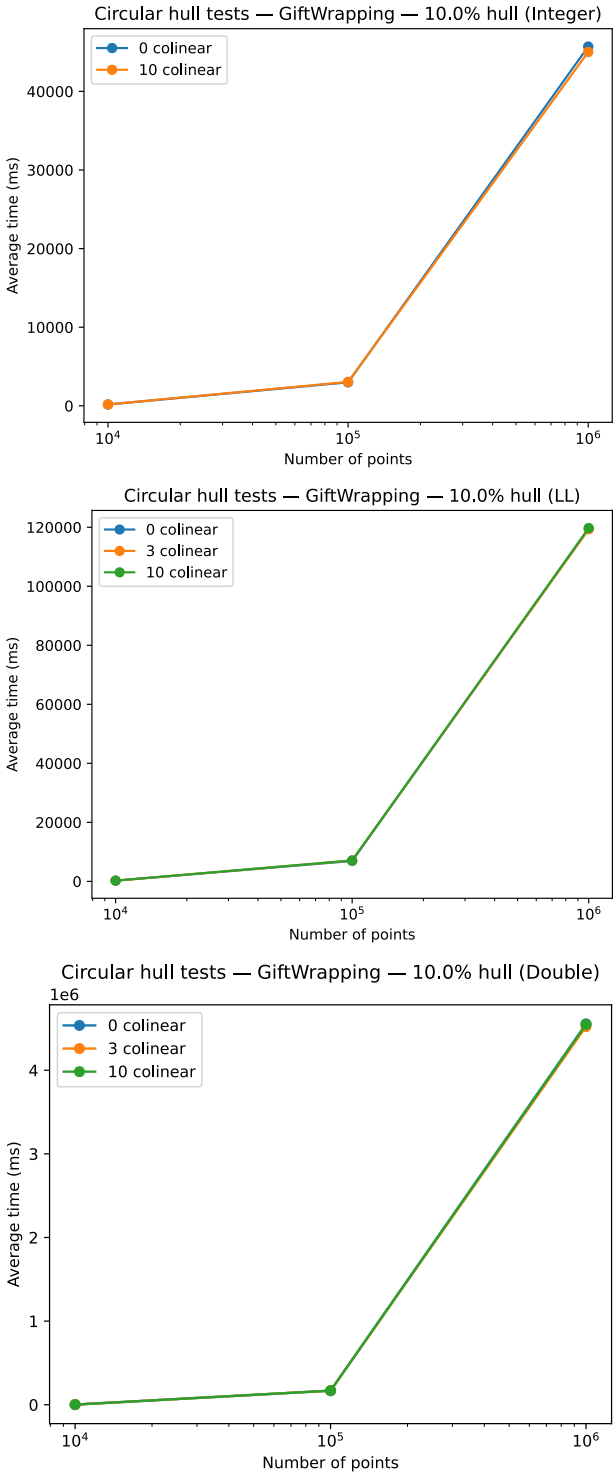


Figura 14: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 10.0% es la cerradura convexa

Puntos aleatorios en un círculo donde el 30.0% de los puntos forman parte de la cerradura convexa (Aquí ya no funcionan los algoritmos con enteros de 32 bits):

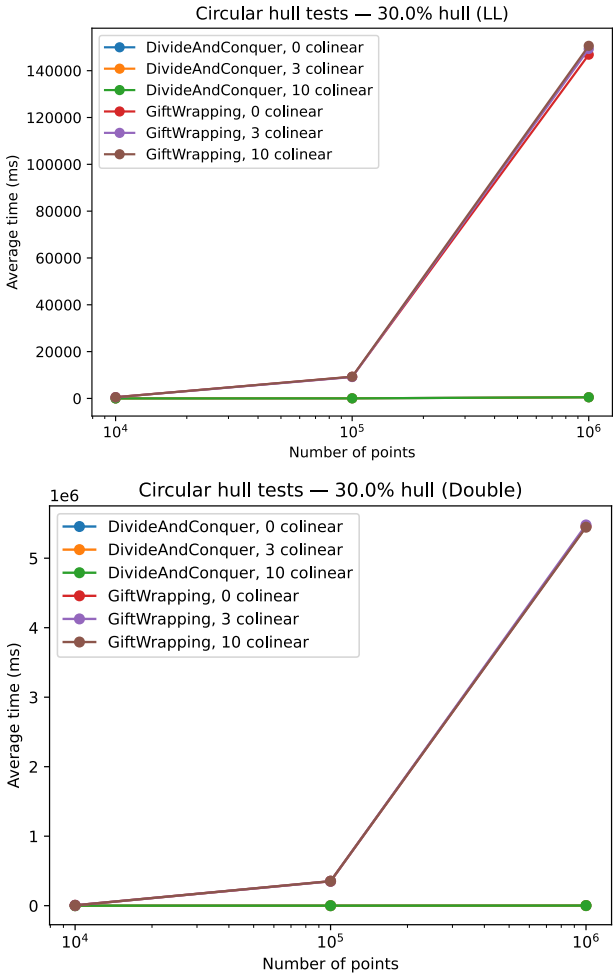


Figura 15: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 30.0% es la cerradura convexa

Mismos gráficos separados por algoritmo:

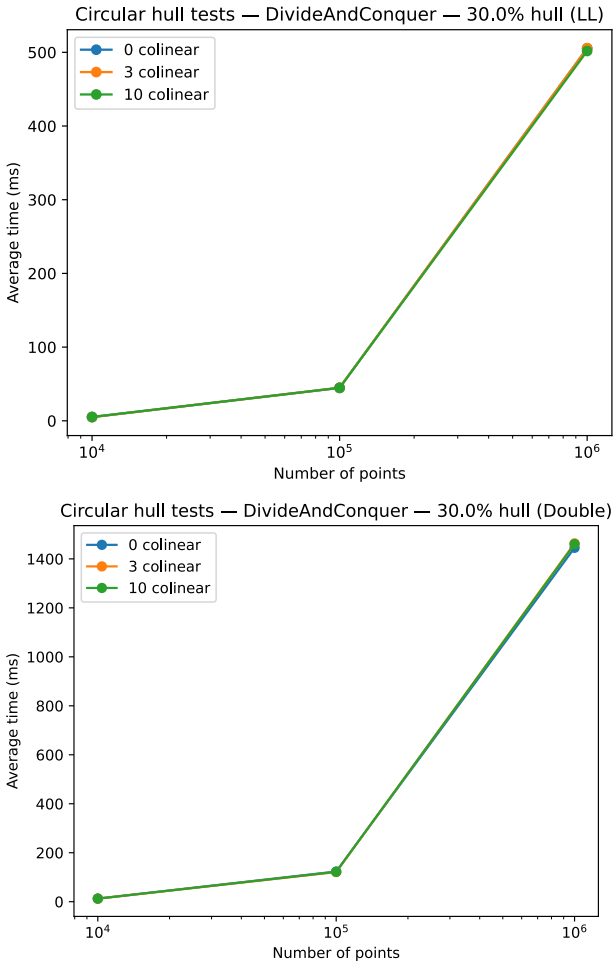


Figura 16: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 30.0% es la cerradura convexa

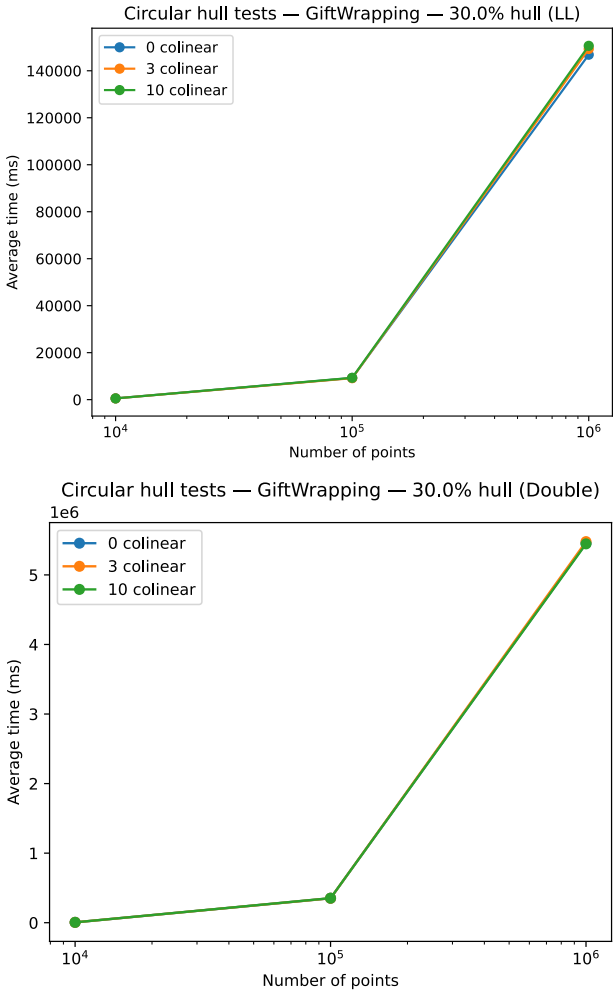


Figura 17: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 30.0% es la cerradura convexa

Puntos aleatorios en un círculo donde el 50.0% de los puntos forman parte de la cerradura convexa:

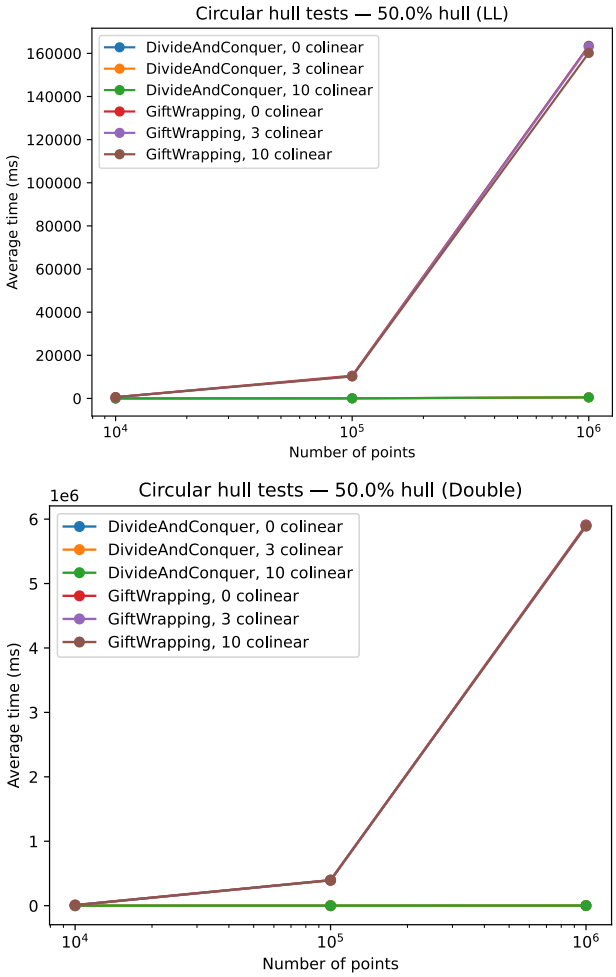


Figura 18: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 50.0% es la cerradura convexa

Mismos gráficos separados por algoritmo:

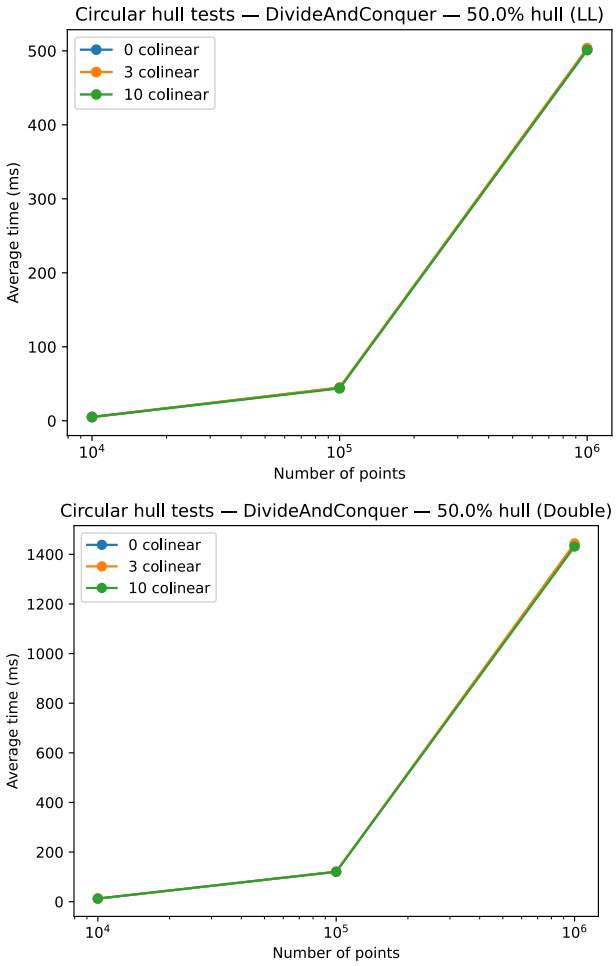


Figura 19: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 50.0% es la cerradura convexa

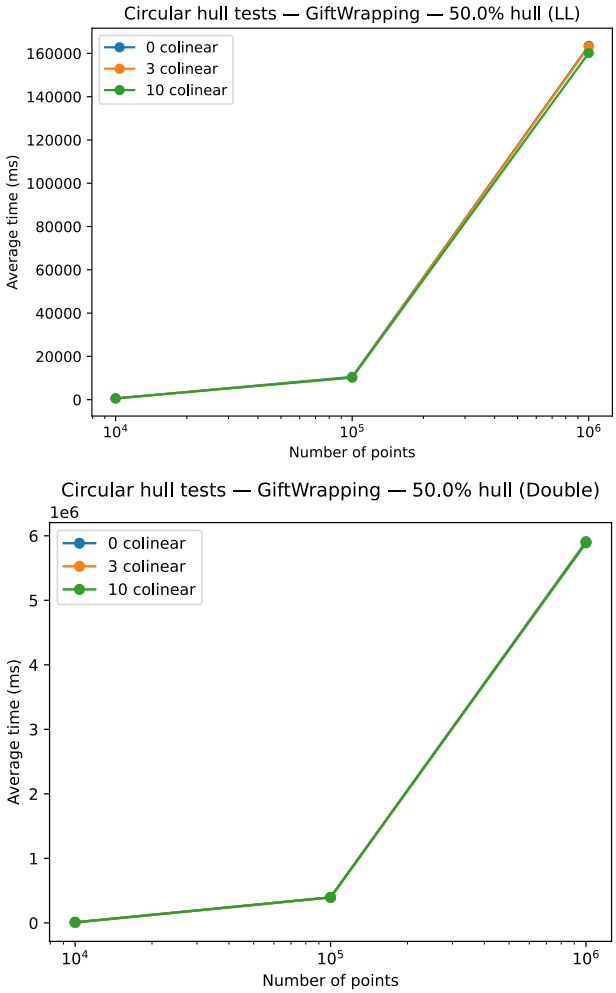


Figura 20: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 50.0% es la cerradura convexa

Puntos aleatorios en un círculo donde el 70.0% de los puntos forman parte de la cerradura convexa:

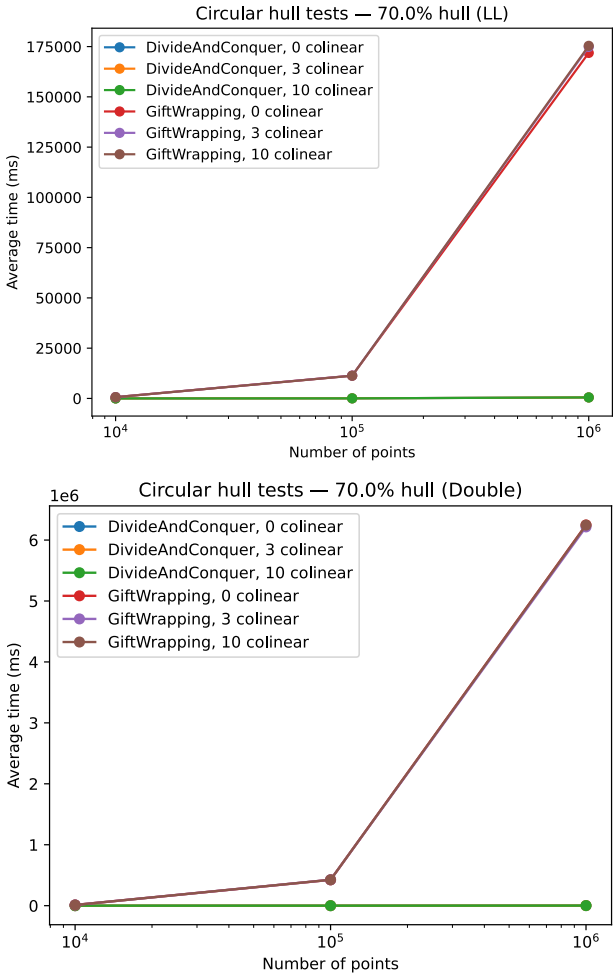


Figura 21: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 70.0% es la cerradura convexa

Mismos gráficos separados por algoritmo:

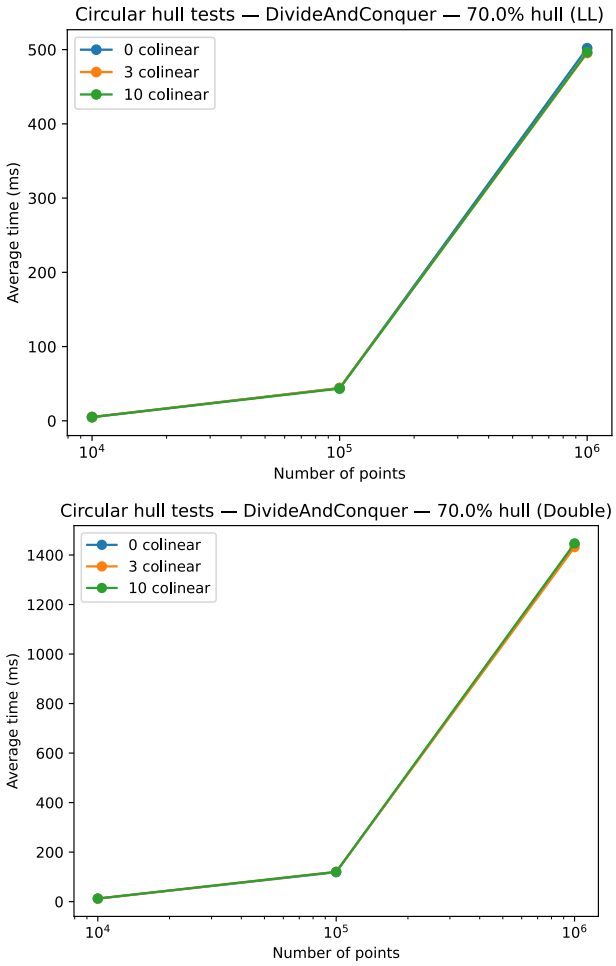


Figura 22: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 70.0% es la cerradura convexa

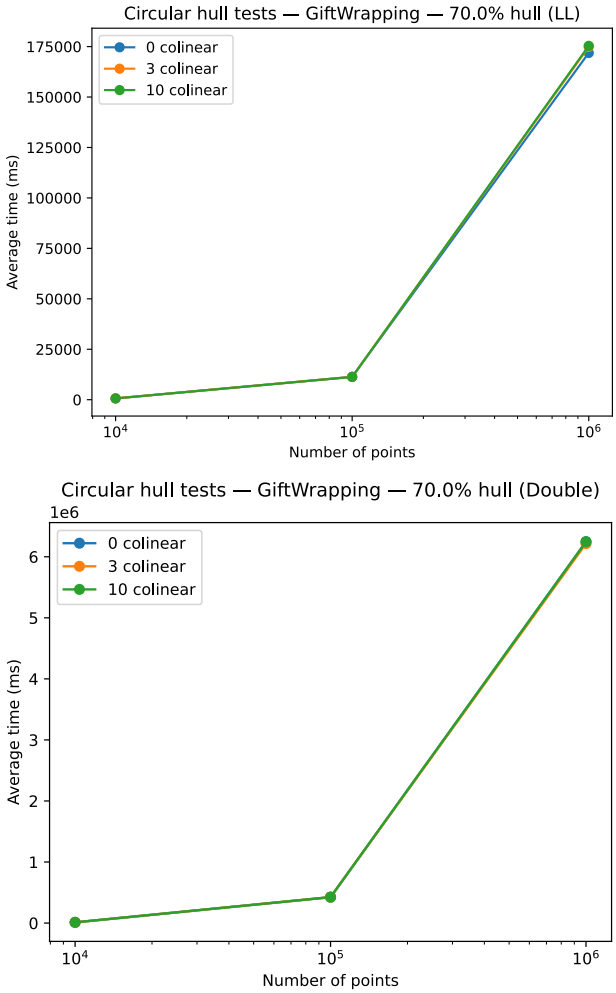


Figura 23: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 70.0% es la cerradura convexa

Puntos aleatorios en un círculo donde el 100.0% de los puntos forman parte de la cerradura convexa:

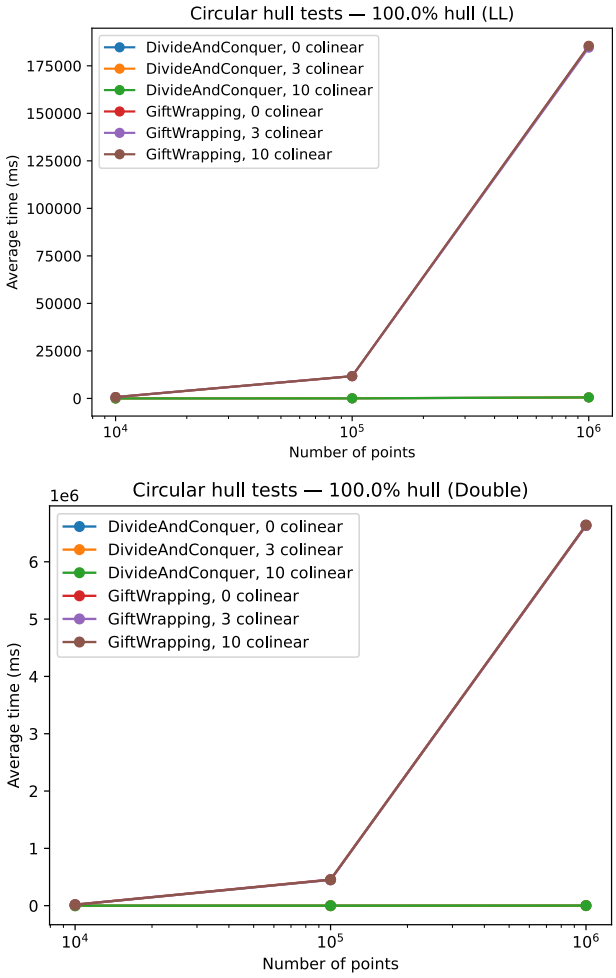


Figura 24: Gráficos de cantidad de puntos en relación a tiempo de ejecución de los algoritmos en milisegundos donde el 100.0% es la cerradura convexa

Mismos gráficos separados por algoritmo:

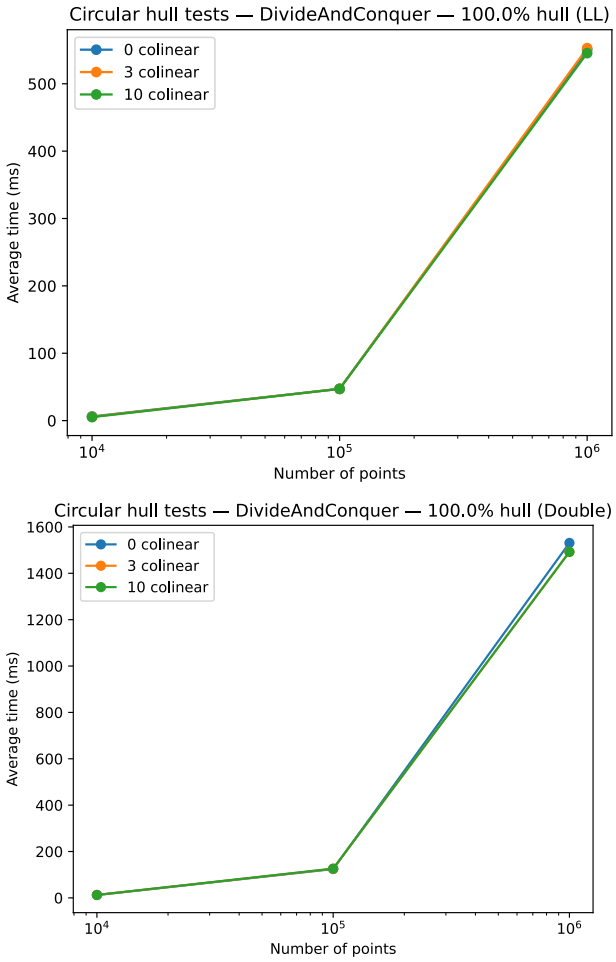


Figura 25: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *Divide and Conquer* en milisegundos donde el 100.0% es la cerradura convexa

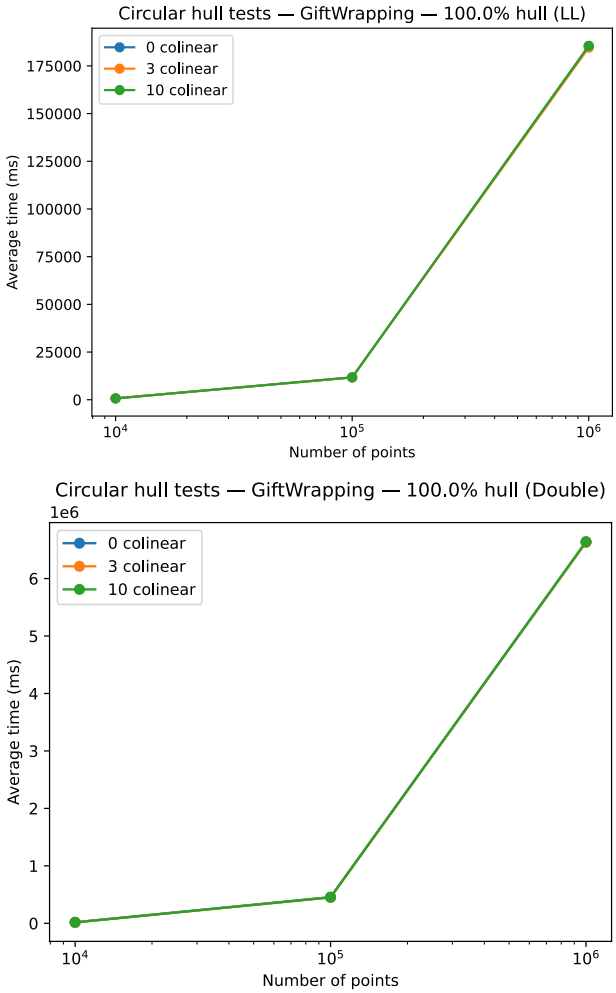


Figura 26: Gráficos de cantidad de puntos en relación a tiempo de ejecución de *GiftWrapping* en milisegundos donde el 100.0% es la cerradura convexa

Preguntas y análisis

1. ¿Cuál algoritmo es mejor para puntos generados aleatoriamente?

R: En promedio, ambos algoritmos tienen un desempeño similar para puntos generados aleatoriamente (con la semilla utilizada, la cual fue 123), sin embargo, el algoritmo *Divide and Conquer* tiene tiempos más estables. Véase Figura 1 y Figura 2

2. ¿Cuál algoritmo es mejor si un porcentaje pequeño de puntos pertenece a la cerradura convexa? ¿Se puede determinar ese porcentaje?

R: Depende del tamaño del conjunto de puntos, debido a la complejidad de ambos algoritmos, siendo estas $O(nh)$ para *GiftWrapping* y $O(n \log n)$ para *Divide and Conquer*, el primero será más eficiente si es que h es menor que $\log n$. Dado que se habla de porcentajes, si suponemos que el porcentaje de puntos que se desea que estén en la cerradura es del $k\%$, entonces:

$$
\begin{aligned}
h &= \frac{kn}{100} \\
\Rightarrow \frac{kn}{100} &< \log n \\
k &< \frac{100 \log n}{n}
\end{aligned} \tag{3}
$$

Luego para que k cumpla esta desigualdad su valor debe ser exponencialmente menor a medida que n crece. En cualquier otra circunstancia, el algoritmo *Divide and Conquer* tendrá mejor desempeño. Esto se puede evidenciar claramente al comparar los tiempos de la Figura 3 con los de la Figura 6, los cuales son muy distantes para *GiftWrapping* y luego a medida que el porcentaje crece, peor rendimiento tiene el algoritmo, como se ve en Figura 9, Figura 12, Figura 15, Figura 18, Figura 21 y Figura 24

El porcentaje de puntos en la cerradura no se puede determinar a priori, pues es precisamente el problema que se busca resolver. La única forma de saberlo de antemano es generar el conjunto de puntos con un patrón conocido que permita asegurar cual es la cerradura sin siquiera tener que correr el algoritmo de ante mano, como es el caso de un conjunto delimitado por un círculo, donde la circunferencia será la cerradura convexa y se puede determinar el porcentaje de puntos según cuantos puntos del total se inserten en dicha circunferencia.

3. ¿Cómo se desempeñan los algoritmos? ¿Existen diferencias en cuanto a tiempo de cálculo?

R: El algoritmo *Divide and Conquer* tiende a ser mejor casi siempre en tiempo, teniendo tiempos de cálculo drásticamente menores a menos que el porcentaje de puntos en la cerradura sea mucho menor que n, pero incluso con un porcentaje del 0.01%, *Divide and Conquer* sigue siendo menor para $n \geq 10000$. Con valores inferiores de n (o h) el algoritmo *GiftWrapping* tendrá mejor rendimiento. En la Figura 1 y Figura 2 se vé que el rendimiento es similar entre los algoritmos y casi igual para $n = 10000$

4. ¿Qué diferencias existen si los conjuntos de puntos son reales o si son enteros? Pruébalo en particular, para un conjunto de puntos en que la cerradura convexa tiene puntos colineales y otro que no tenga puntos colineales.

R: Cuando los puntos son enteros, los algoritmos toman considerablemente menos tiempo en calcular la cerradura convexa, habiendo un cambio tan drástico como pasar de un poco menos de 3 minutos (Figura 6) a casi 3 horas (Figura 24) para el caso del algoritmo *GiftWrapping* para un $n = 10^6$. La presencia o ausencia de puntos colineales en la cápsula no suele generar mucha diferencia en tiempo, pero si aumenta la propensión a que los algoritmos fallen y computen cápsulas distintas, siendo el algoritmo *GiftWrapping* el que suele entregar un resultado más acertado.