Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
CC4102 Diseño y Análisis de Algoritmos

Tarea 3
Closest pair of points

Integrantes: Gaspar Gumucio
Valentina Núñez
Gonzalo Sobarzo Abufón
Profesores: Benjamín Bustos
Gonzalo Navarro
Auxiliares: Asunción Gómez
Diego Salas

Fecha de entrega: 5 de diciembre de 2023
Santiago, Chile

1. Introducción

Los algoritmos aleatorizados son de gran interés debido a que ofrecen costos esperados independientes de la distribución de inputs que se le entregue. Aunque el adversario ingrese un input de peor caso, este algoritmo variará su costo en cada ejecución. Esto resulta conveniente cuando se desea resolver un problema donde no importa que algunas pocas ejecuciones demoren mucho sino que en la mayoría de los casos tengan bajo costo, no afectando que el input sea de peor caso.

En este estudio se pretende comparar el costo un algoritmo aleatorizado con uno determista. Como motivación se resolverá el problema *Closest Pair of Points* que consiste en encontrar la distancia del par de puntos más cercano de un conjunto de puntos. Se estudiará experimentalmente el tiempo que demora cada uno y se analizará si esto se adecua a los costos teóricos.

Especificamente se implementarán tres variantes para el algoritmo aleatorizado de Rabin [1]. La primera utilizando Hashing Universal, luego obteniendo el primo de Mersenne y por último Hashing con Funciones más rápidas.

Los experimentos consistiran en ejecutar los algoritmos sobre conjuntos de puntos aleatorios. En el primero se estudiará cómo varía el costo de los algoritmos versus el tamaño del conjunto de puntos. En el segundo se ejectuarán para un tamaño fijo pero variando el input en cada ejecución.

Se espera que como en la teoría el algoritmo determinista muestre costo $O(nlog(n))$ y los aleatorizados $O(n)$. También que utilizando primos de Mersenne dismuya el costo en comparación al Hashing Universal, y que el Hashing con funciones más rápidas sea el que menos demore.

Por último, al utilizar funciones de hash que tienen probabilidades de colisión $\frac{1}{m}$. Los histogramas de tiempo de las ejecuciones de los algoritmos aleatorizados deberían tener mayores casos de tiempos menores.

2. Desarrollo

Para esta tarea se tuvo que programar dos algoritmos que permitieran encontrar el par de puntos más cercanos en un conjunto de puntos, siendo uno determinístico y otro aleatorizado. Para el algoritmo determinístico se utilizó la técnica "dividir para reinarz para el algoritmo aleatorizado se implementaron tres métodos de hashing: hashing universal, hashing del apunte (en la última parte de la sección 6.4.1) y hashing que utiliza los primos de Mersenne.

Por último, se implementan test para comparar el comportamiento de estos dos algoritmos frente a diferentes inputs y también para comparar sus tiempos de ejecucción.

A continuación se detallan estas implementaciones.

2.1. Algoritmo Divide and Conquer.

Las principales funciones son:

- *divideAndConquer*: ordena un conjunto de puntos y aplica la función *recursion* para encontrar la distancia mínima entre pares del conjunto de puntos.
- *recursion*: utiliza técnica "dividir para reinar"para encontrar la distancia mínima entre pares de un conjunto de puntos.
- *x_dist*: obtiene el cuadrado de la distancia entre dos coordenadas.

2.1.1. Funcionamiento Divide and Conquer.

Dado un vector de puntos, la función *divideAndConquer* los ordena de forma ascendente según su coordenada x, para luego aplicar la función *recursion*.

La función *recursion* tiene como casos bases que el tamaño del vector de puntos sea 2 o menor a 2, en cuyo caso retorna la distancia entre los dos puntos o retorna infinito, respectivamente.
Para el caso recursivo, se calcula una recta divisioria según cierta coordenada x que divida el conjunto de puntos en dos mitades iguales. Si el tamaño del conjunto es par, se toma como coordenada divisoria el promedio de las coordenadas x de los puntos centrales, de lo contrario se toma la coordenada x del punto central.

Luego se aplica recursivamente *recursion* para ambas mitades. Se calcula el mínimo *dLRmin* del resultado obtenido de ambas soluciones. Según este mínimo obtenido, se recorren los puntos de la mitad izquierda (de forma descendente) y de la mitad derecha (de forma ascendente) guardandose aquellos que tengan una distancia menor a *dLRmin* con respecto a la recta divisoria. Con este conjunto obtenido, se prueban todas las combinaciones de pares de puntos, calculando su distancia y comparandola con la distancia mínima actual, actualizando así *dLRmin*.

Por último, se retorna *dLRmin*, que representa la distancia mínima entre dos puntos de todo el conjunto de puntos original.

2.2. Algoritmo Aleatorizado.

2.2.1. Descripción General:

El algoritmo aleatorizado fue implementado en el archivo *aleatorizado*5.*cpp*, y la función que lo implementa se nombró como *aleatorizado* la cual entrega un par con la distancia mínima cuadrada y la mínima cuadrada obtenida en el primer paso del agloritmo. Esta última se entrega solo por razones de debuggin debido a que el código daba problemas en esta parte. Se utiliza la distancia cuadrática por razones de eficiencia ahorrandose así el calculo de una raíz cuadrada.

La función *aleatorizado* recibe como parámetro un arreglo de puntos. Los puntos son representados en la estructura *Point* y para crear arreglos se utiliza la librería *vector* de C++. En resumen el algoritmo tiene 3 etapas:

- Se encuentra una distancia mínima inicial llamada *d_square* comparando n pares al azar, esto se implementa en la función *comparacion_pares*.
- Asignar cada punto a un cuadrado de una grilla: esta es una grilla de cuadrados de lado $d = \sqrt{d\_square}$, notando que cada punto se puede asignar al cuadrado que lo contiene sin ambiguedad al dividir sus coordenadas por d. Luego ese cuadrado es representado por una llave al poner su abscisa en los primeros 18 bits de un unsigned long long y su ordenada en los 18 bits menos significativos.
- Recorrer puntos encontrando su distancia mínima a otros puntos dentro de su vecindario de 1 cuadrado a la redonda. Quedandose con el mínimo de todas las comparaciones.

2.2.2. Hashing Universal:

Para guardar los puntos eficientemente segun el cuadrado de la grilla que los contenía se utilza un tabla de hash. Para el hashing universal, la función de hash que entrega la posición en la tabla donde se debe guardar la llave es la siguiente:

$$h(x) = ((ax + b) \bmod p) \bmod m$$

Donde p es un número primo. En primera instancia se utilizó el algoritmo de Miller-Rabin para encontrar el primo, pero debido a que demoraba más del doble que el resto del algoritmo se decide elegir una lista de primos que abarcaran cada orden de magnitud y elegir de entre ellos el que fuera mayor al universo.

Luego con a y b elegidos aleatoriamente en el rango $[1, p-1]$ y $[0, p-1]$ respectivamente, se calcula el modulo de m donde $m = 2n$, el tamaño de la tabla de hash eligiendose como el doble de elementos a insertar (número de puntos).

2.2.3. Primos de Mersenne:

Es lo mismo que en el Hashing Univeral, solo que ahora la elección del primo se hace de la siguiente lista de primos conocidos como primos de Mersene:

$$M_n = [3, 7, 31, 127, 8191, 131071, 524287, 2147483647]$$

Como un primo de Mersenne es de la forma $2^k - 1$ la operación modulo se puede optimizar utilizando operadores de bits.

2.2.4. Funciones más Rápidas:

Se elige el tamaño de la tabla de hash como una potencia de 2. Esto se hace aplicando $l = log(m)$, así el nuevo m es de la forma $m = 2^l$. Luego calcular modulo de m es sencillo ya que es obtener los l bits menos significativos.

2.3. Pruebas

En esta sección, se explicarán los dos experimentos realizados por el grupo, en los cuales se utilizó una función para generar puntos aleatorios ubicada en el archivo `structs.cpp` con el fin de generar n puntos pertenecientes al rango $[0, 1) \times [0, 1)$ de forma aleatoria. Es importante señalar que se redujo la dimensión de los arreglos sobre los cuales se ejecutó el primer experimento debido a que se estimaba una ejecución superior a las 24 horas para la distribución original. Por lo tanto, se decidió reducir la dimensión para obtener los resultados lo más rápido posible.

2.3.1. Conjunto de puntos creciente

Este experimento, ubicado en el archivo `experimento_1.cpp`, consiste en la creación de n puntos aleatorios en el rango mencionado anteriormente, con n en $[500,000; 5,000,000]$ y pasos de $500,000$. Para cada arreglo, se ejecutan los algoritmos creados 100 veces cada uno. Del algoritmo determinístico se obtiene la distancia mínima y la duración de la búsqueda en cada ejecución. En cambio, para los algoritmos, se obtienen estas dos variables además del d obtenido con la comparación de pares aleatorios en el primer paso del algoritmo, para su posterior evaluación.

2.3.2. Conjunto de entradas aleatorias

Para este experimento, ubicado en el archivo `experimento_2.cpp`, se toman inputs aleatorios generados en cada una de las 50 iteraciones realizadas con n en $[100,000; 1,000,000]$ y pasos de $100,000$ para poder coomparar si el comportamiento se adecua al presentado por la teoría, se guarda la misma información del experimento anterior para su posterior análisis.

3. Resultados

Para poder obtener los resultados los programas se ejecutaron en un pc con las siguientes especificaciones:

- Procesador:
  - Modelo: AMD Ryzen 7 4800hs
  - Cores: 8
  - Threads: 16
  - Frecuencia: 2.7 GHz hasta 4.0 GHz
  - Cache N2: 4 MB
  - Cache N3: 8 MB
  - Memoria máxima: 64 GiB
- Memoria principal (RAM): 16 GB a 3200Mhz
- Memoria secundaria:
  - Modelo: Samsung MZVLQ512HBLU
  - Capacidad: 512 GB
  - Velocidad de escritura: 1800 $\frac{MB}{s}$
  - Velocidad de lectura: 3100 $\frac{MB}{s}$
  - Tamaño de bloque: 4096 bytes
- Sistema Operativo: Ubuntu 22.04.3.
- Versión Compilador: GCC 13.2

A continuación presentaremos los gráficos de los resultados que se obtuvieron al realizar los experimentos.
Dado n puntos en el rango rango [0, 1) x [0, 1), el primer experimento consistió en probar cada algoritmo, variando n de 5 a 50 millones (cada 5 millones), utilizando el mismo arreglo de puntos y ejecutando 100 veces por cada n. Esto con el objetivo de analizar los timepos de ejecucción y el comportamiento de las medias en función de n, para cada algoritmo.

Para el segundo experimento, se probó cada algoritmo con diferentes inputs aleatorios, esto con el objetivo de analizar el tiempo promedio.

3.1. Experimento 1

A continuación se muestran los resultados obtenidos para cada algoritmo, luego de realizar el primer experimento.

En las figuras 1, 2, 3 y 4 se muestra el histograma de los tiempos de ejecución para cada n obtenidos por el algoritmo determinístico, algoritmo aleatorio, algoritmo aleatorio con mersenne y algoritmo aleatorio con función hash rápida, respectivamente.

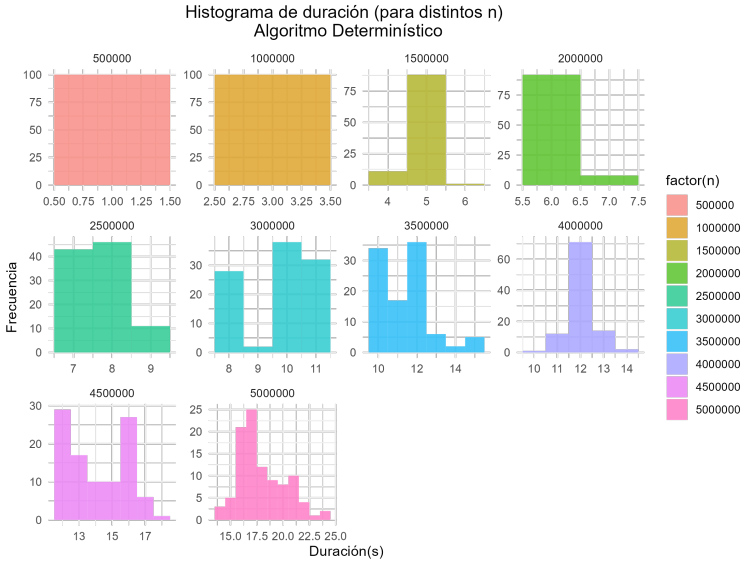

Figura 1: Histograma tiempos de ejecución-
Algoritmo Divide and Conquer

Se puede observar que, para n en el rango de 0.5 a 1 millones, el algoritmo determinístico presenta una distribución simétrica, su media en el centro y poca dispersión. Se tiene una frecuencia de 100 para el mismo rango de tiempo, siendo estos rangos (0.5, 1.5) y (1.5, 2.5) segundos, para n igual a 0.5 y 1 millones, respectivamente.

Para n mayor a 1 millón, ya no se tiene una distribución simétrica, sin embargo la dispersión es mínima hasta n 3 millones, los rangos de tiempo se mantienen cercanos. A partir de los 4.5 millones existe mayor asimetría y dispersión.

En resumen, se observa que para n no muy grande, los tiempos de ejecucción del algoritmo deterministico no varían significativamente, sino que se mantienen relativamente dentro del mismo rango de tiempo.

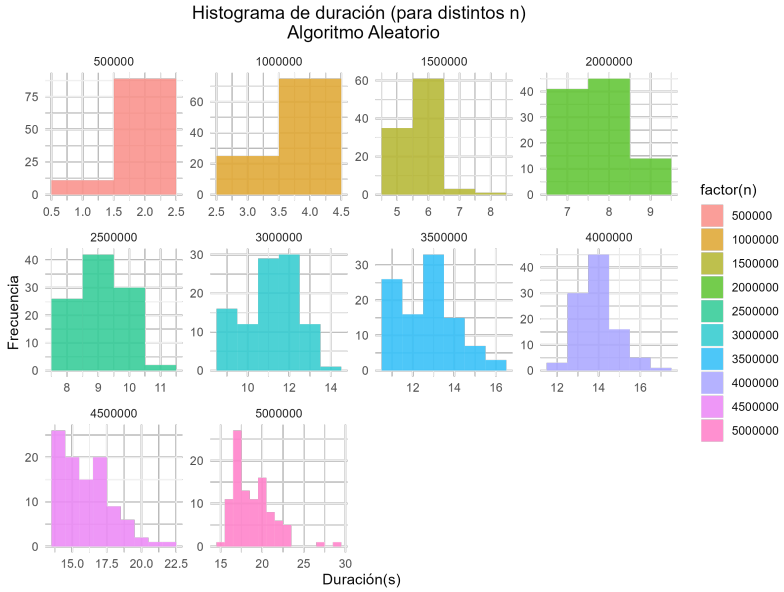

Figura 2: Histograma tiempos de ejecución - Algoritmo Aleatorizado

Se puede observar que, todos los n tienen una distribución asimétrica. Para n menor a 2.5 millones, las frecuencias se concentran en pocos intervalos. Y para n mayor a 2.5 millones se presenta mayor dispersión.

En resumen, los tiempos de ejecucción del algoritmo aleatorio varían significativamente a partir de n mayor a 2.5 millones.

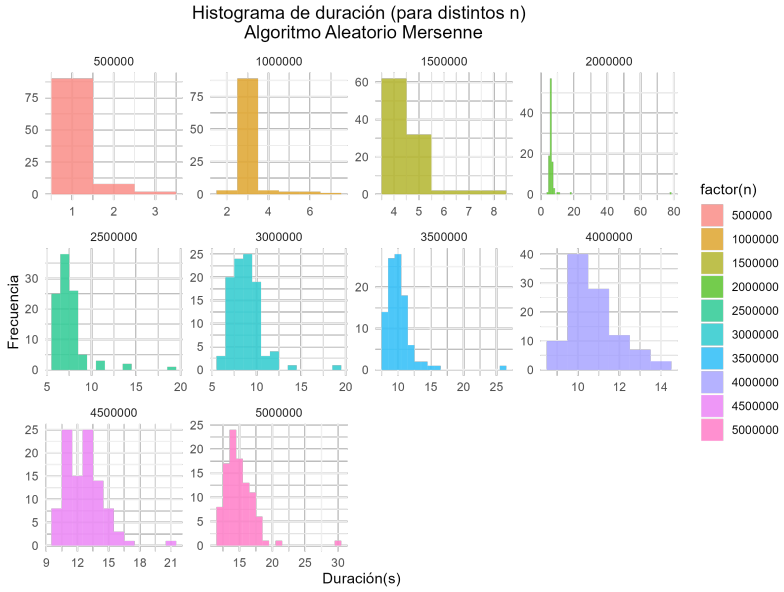

Figura 3: Histograma tiempos de ejecución - Algoritmo Aleatorizado Mersenne

Se observa que se tiene asimetría para la distribución de los tiempos de ejecucción. Para n menor a 1.5 millones no se presenta mucha dispersión. Para n igual a 2 millones hay presencia de outliers. Luego para n mayores a 2 millones también existen outliers, pero no tan significtavos y se puede apreciar que la distribución tiende a una normal.

En resumen, los tiempos de ejecucción del algoritmo aleatorio con mersenne no varían significativamente para n mayor a 1.5 millones, a pesar de que se tienden a concentrar en cierto intervalo, existen ejecuciones que generan tiempos muy lejanos al promedio (outliers).

El siguente gráfico muestra el comportamiento de la medias de los tiempos de ejecucción en función del tamaño del input n, para cada algoritmo.

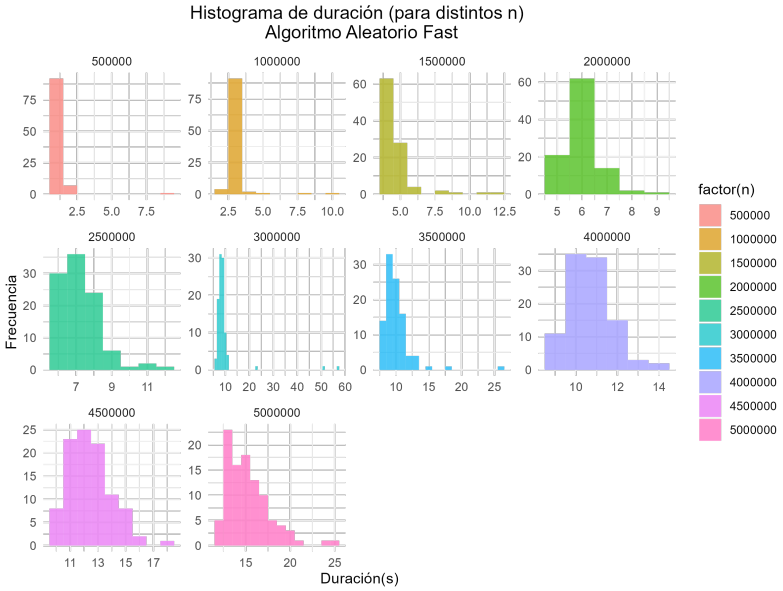

Figura 4: Histograma tiempos de ejecución
Algoritmo Aleatorizado Fast

Se puede observar que la distribución de los tiempos de ejecucción no es simétrica. Para n igual a 0.5 y 1 millones los tiempos están mayormente concentrados en un intervalo, empezándose a disperar a medida que crece n. Ya para n mayor a 1 millón se presentan outliers. La distribución tiende a concentrarse a la izquierda.

En resumen, los tiempos de ejecucción del algoritmo aleatorio con hash rápido varían significativamente a partir de n mayor a 1.5 millones, a pesar de que se tienden a concentrar en cierto intervalo, existen ejecuciones que generan tiempos muy lejanos al promedio (outliers).

El siguente gráfico muestra el comportamiento de la medias de los tiempos de ejecucción en función del tamaño del input n, para cada algoritmo.

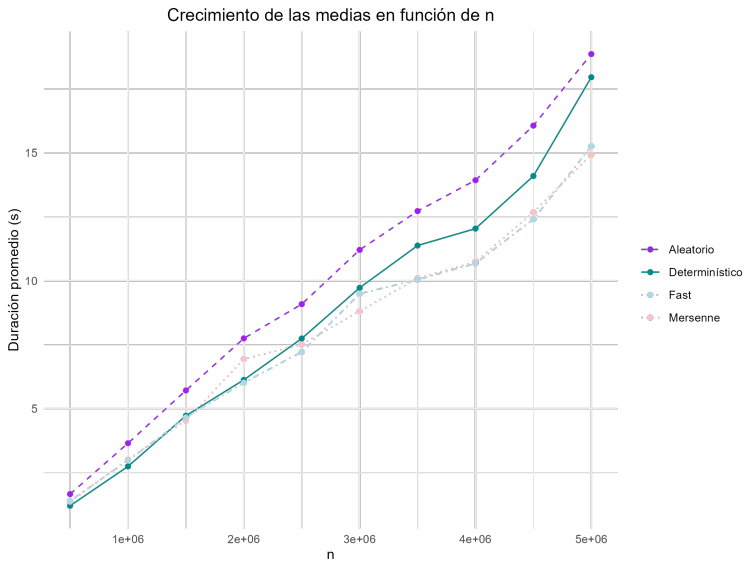

Figura 5: Tiempos promedios

3.2. Experimento 2

Con respecto a los inputs aleatorios se observa las curvas de tiempo en la figura 6 se ve que la curva con el peor tiempo promedio es la correspondiente al algoritmo aleatorizado usando hashing universal, posteriormente esta el algoritmo determinístico con el segundo peor rendimiento y por último se encuentran muy igualadas las curvas del algoritmo aleatorio usando primos de Mersenne y hashing fast.

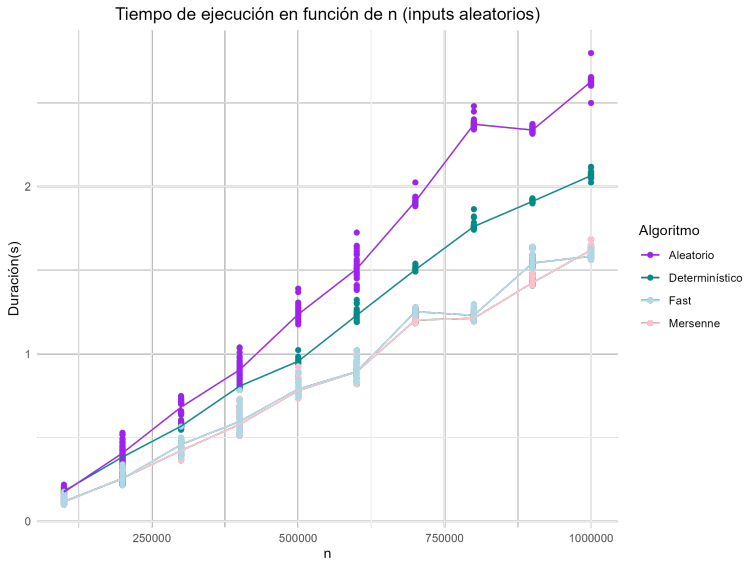

Figura 6: Tiempos promedios (inputs aleatorios)

4. Análisis

4.1. Divide & Conquer vs Aleatorizado

Se puede observar que la primera componente aleatoria de los algoritmos aleatorizados es la elección del d. Dependiendo si este es más parecido al mínimo original el algoritmo tiene que comparar menos puntos ya que los cuadrados de la grilla son más pequeños. Esta componente aleatoria se ve reflejada en los histogramas obtenidos, que muestran más varianza que el determinista.

Observando las figuras 5 y 6 se concluye que el algoritmo determinista demora menos que el aleatorizado. Esto contradice la teoría, donde el aleatorizado debería ser de orden menor. Esto pudo deberse a la implementación o a la elección de estructuras de datos menos eficientes. Mientras se implementaba se observó que pequeños cambios en el código afectaban significativamente en los tiempos, por ejemplo al no encapsular código en funciones o no crear muchas copias de variables auxiliares. Todo eso pudo haber afectado en que terminara resultando más costoso que el determinista, que tenía la ventaja de ser más simple de implementar induciendo menos fuentes de ineficiencia.

4.2. Experimento 1: Ejecuciones para un mismo arreglo

Como en este experimento el arreglo de puntos es el mismo para las 100 iteraciones, lo que se está reflejando al calcular el tiempo medio es el costo esperado. En ese sentido podemos ver en la figura 1 que el algoritmo determinista varía menos su tiempo de ejecución que los aleatorizado. El determinista obtiene tiempos en un rango de 2 segundos mientras que los aleatorizados varian sus tiempos en un rango mayor a 3 segundos para n mayores (figuras 2 a 4), esto corresponde con la teoría dado que el determinista no tiene un factor aleatorio en cada ejecución y lo único que puede hacer variar su tiempo son factores físicos del procesamiento de la máquina.

También se puede observar que en los algoritmos aleatorizados los histogramas concentran una mayor cantidad de tiempos menores que mayores dentro del rango de tiempos. Esto acierta con la hipótesis planteada en la introducción, ya que la función al ser aleatoria puede tener colisones con probabilidad $\frac{1}{m}$ esta probabilidad hace esperar que hayan pocos casos donde el algoritmo demore más, o sea donde tocó una función de hash que producía muchas colisiones y no distribuía los elementos de manera uniforme.

4.3. Experimento 2: Ejecuciones para arreglos aleatorios

Al igual que en el experimento 1, la figura 6 muestra que los costos de los algoritmos aleatorios crecen linealmente con el tamaño del input n. Esto acierta con la hipótesis y la teoría donde el algoritmo aleatorizado es de $O(n)$. Sin embargo en el determinista no es posible determinar con certeza el comportamiento $O(nlog(n))$, ya que también tiende a crecer linealmente, puede que para valores de inputs más grandes se vea reflejado este efecto.

4.4. Comparación Hashing

Se puede observar en la figura 5 y 6. Que al utilizar primos de Mersenne y Funciones más rápidas los tiempos medios disminuyen en comparación al aleatorizado, lo que concuerda con la hipótesis. Sin embargo no hay grandes diferencias antre Mersenne y Funciones más rápidas. Esto pudo deberse a que aplicar modulo m no es costoso cuando el tamaño de la tabla no tiene valores muy grandes. Como el tamaño de la tabla era función del numero de puntos quizás probar con un n más grande hubiera evidenciado el comportamiento esperado.

4.5. Comparando experimentos

Al comparar los resultados obtenidos en ambos experimentos se ve aroximadamente que los algoritmos se comportan de la misma manera en términos de la forma de la curva independiente si se itera 100 veces sobre el mismo input o si se ejecuta 50 veces sobre entradas distintas generadas aleatoriamente, lo que safisface el marco teórico de los algoritmos aleatorizados mostrando que el tiempo esperado logrado es independiente del arreglo de puntos recibido.

5. Conclusiones

Se logran validar la mayoría de las hipotesis presentadas en la introducción:

- Los algortimos aleatorizados presentan el comportamiento $O(n)$
- Aplicar primos de Mersenne o Funciones más rápidas si produce costos menores.
- Los histogramas evidencian las probabilidades asociadas a la función de hash (los tiempos se concentran en tiempos más bajos, teniendo pocos casos de tiempos mayores)

Sin embargo otras no resultaron como lo esperado:

- El algoritmo determinista no muestra un comportamiento $O(nlog(n))$
- El algoritmo determinista presenta costos menores al aleatorizado
- Se esperaba que Funciones más rápidas tuviera tiempos menores a Mersenne, pero se comportaron similarmente

Referencias

[1] M. Rabin. «Probabilistic algorithms». En: *Algorithms and Complexity: Recent Results and New Directions*. As cited by Khuller & Matias (1995). Academic Press, 1976, págs. 21-39.