Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

8 de Abril de 2016

Trabajo Práctico Número 1

Integrante	LU	Correo electrónico
Ciruelos Rodríguez, Gonzalo	063/14	gonzalo.ciruelos@gmail.com
Costa, Manuel José Joaquín	035/14	manucos94@gmail.com
Gatti, Mathias Nicolás	477/14	mathigatti@gmail.com
Maddonni, Axel	200/14	axel.maddonni@gmail.com

Índice

1. Kaio Ken

1.1. Explicación formal del problema

1.2. Explicación de la solución

1.3. Complejidad del algoritmo

El análisis de complejidad es simple, es un algoritmo de Divide & Conquer clásico, que divide siempre el trabajo en 2 y luego fusiona los resultados de los subproblemas en tiempo $O(n)$. Haciendo una analogía, por ejemplo, con el algoritmo de MergeSort, se puede predecir fácilmente que la complejidad será de $O(nlogn)$.

1.3.1. Esbozo del algoritmo

El algoritmo fue analizado en profundidad anteriormente. A grandes rasgos, puede describirse de la siguiente manera:

Algorithm 1 Esbozo del algoritmo de KaioKen

```
procedure GENERARPELEAS(int n, int pactual, int inicio)
    if n = 1 then
        matrizpeleas[pactual][inicio] ← 1
    if n = 2 then
        matrizpeleas[pactual][inicio] ← 1
        matrizpeleas[pactual][inicio + 1] ← 2
    else
        for j ∈ [0, ..., n) do
            if j < n/2 then
                matrizpeleas[pactual][inicio + j] ← 1
            else
                matrizpeleas[pactual][inicio + j] ← 2
        generarpeleas(n/2, pactual + 1, inicio)
        generarpeleas((n+1)/2, pactual + 1, n/2 + inicio)
```

Como puede verse claramente, tenemos dos casos base que toman tiempo constante en ser resueltos.

Por otro lado, el tercer caso realiza un trabajo de costo lineal, escribiendo n entradas de la matriz, y luego hace 2 llamadas recursivas, dividiendo el trabajo en 2 mitades iguales (en caso de que n sea impar, la segunda mitad va a tener un elemento más).

1.3.2. Análisis temporal

Si quisieramos expresar la cantidad de operaciones que realiza el algoritmo para un input de tamaño n, podriamos escribirlo fácilmente de la siguiente manera:

$$T(1) = 1$$

$$T(2) = 2$$
$$T(n) = n + 2T\left(\frac{n}{2}\right)$$

Ahora podemos usar el teorema maestro. El teorema maestro se referia a relaciones de recurrencia de la pinta:

$$T(n) = f(n) + aT\left(\frac{n}{b}\right)$$

Y afirmaba, entre otras cosas, que si $f(n) \in O(n^c \log^k n)$ donde $c = \log_b a$, entonces $T(n) \in \Theta(n^c \log^{k+1} n)$. En este caso, se ve claramente que $f(n) = n \in O(n^1 \log^0 n)$, y además $1 = \log_2 2$, por lo que el teorema maestro se puede aplicar, y nos dice que

$$T(n) \in \Theta(n \log n)$$

La complejidad de este algoritmo es siempre $\Theta(n \log n)$, sin distinción entre casos, es decir, este algoritmo no tiene mejor o peor caso. La forma más clara de verlo es que el único input del problema es n, y no hay otro parámetro que pueda modificar su complejidad.

1.4. Performance del algoritmo

Como dijimos antes, la complejidad del algoritmo es siempre $\Theta(n \log n)$, sin distinción entre casos, por lo que el análisis de performance es simple.

Primero veamos que, en la práctica, la complejidad del algoritmo es efectivamente $\Theta(n \log n)$.

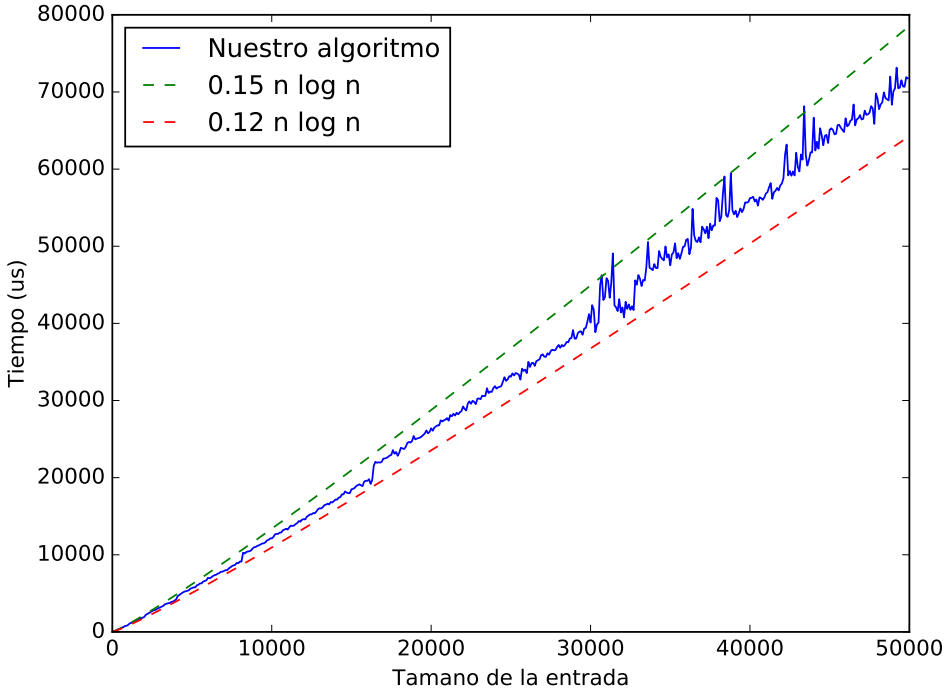

Figura 1: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Se ve claramente en la figura 1 que el tiempo que toma el algoritmo esta acotado por arriba y por debajo por $kn \log n$ para algun k, es decir, el algoritmo es $\Theta(n \log n)$.

Para hacer un análisis más fino e interesante, es necesario hacer un *close-up* y ver las complejidades de mas cerca.

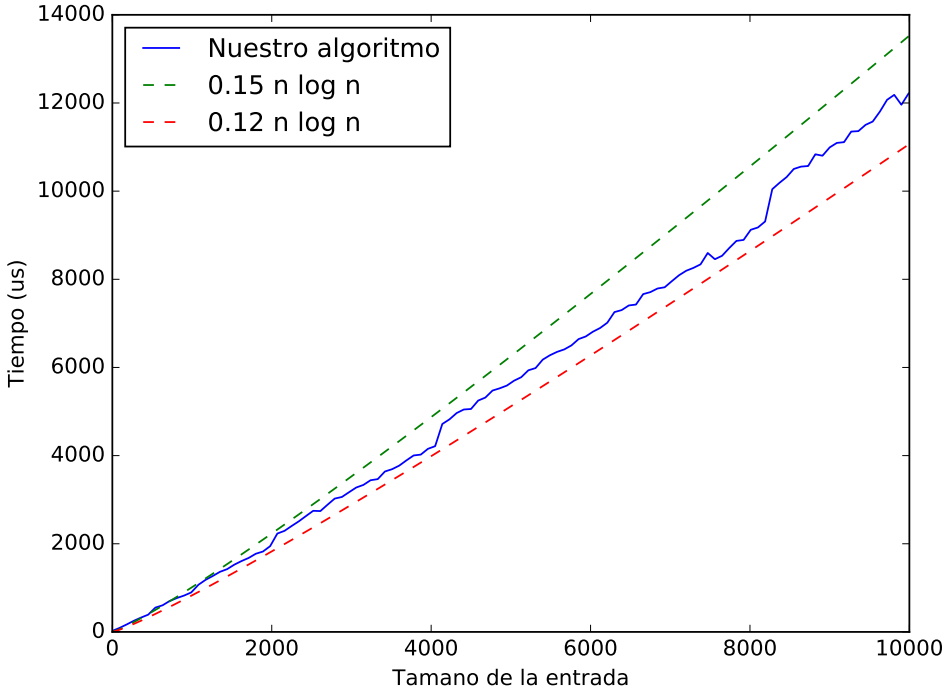

Figura 2: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Como puede verse en la figura 2, el algoritmo claramente es $\Theta(n \log n)$, pero también puede observarse que hay *saltos* de tiempo en algunos lugares. Se puede inferir fácilmente que estos saltos suceden en las entradas donde $n = 2^k + 1$ para algun k, es decir, cuando n es una potencia de 2 más 1. Esto se debe a que, como fue explicado antes, la cantidad de peleas necesarias es $\lceil \log_2(n) \rceil$, entonces para todos los numeros entre dos potencias de 2, la cantidad de peleas requerida es la misma, pero luego de una potencia de 2 esta cantidad de peleas aumenta en 1. A esto se debe los saltos luego de las potencias de 2.

Para visualizar más claramente este hecho, realizamos el siguiente gráfico, en el que están marcadas las potencias de 2.

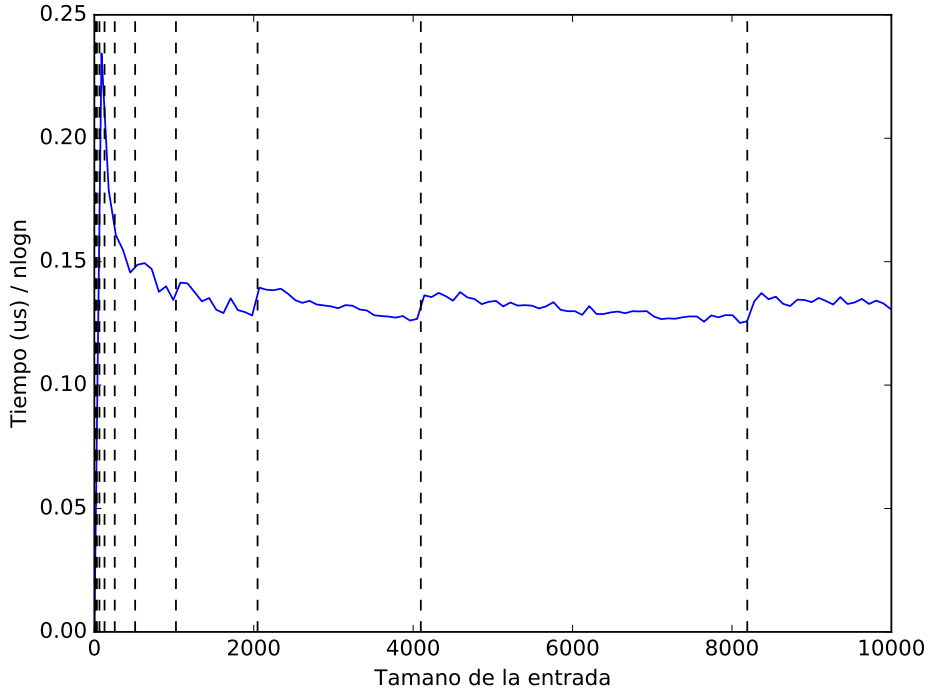

Figura 3: Tiempo que toma el algoritmo en μs dividido $n \log n$para una entrada de tamaño n.

Con la figura 3 se confirma lo que dijimos anteriormente. Luego de cada potencia de 2, el tiempo aumenta, y luego baja lentamente, dado que la relacion tiempo - $n \log n$ se mantiene constante, pero n aumenta, con lo cual la división se achica.

1.4.1. Método de experimentación

Para este problema es fue muy fácil generar casos de prueba, dado que el único input para el programa es n, por lo que simplemente corrimos el programa con diferentes n de entrada. Ejecutamos el programa 50 veces para cada n y el resultado que graficamos fue el mínimo de todas las ejecuciones (para cada n). Elegimos el mínimo porque es el que nos permite eliminar outliers que tienen que ver con context switches o con factores de la ejecucion sobre los que no tenemos control.

2. Genkidama

2.1. Explicación formal del problema

2.2. Explicación de la solución

2.3. Complejidad del algoritmo

2.4. Performance del algoritmo

Como vimos anteriormente, el algoritmo tiene una complejidad de $\Theta(n)$. El algoritmo no tiene peor o mejor caso propiamente dichos (dado que es $\Theta(n)$ para todos los casos), pero veremos que las constantes pueden cambiar para algunas entradas por cuestiones de implementación.

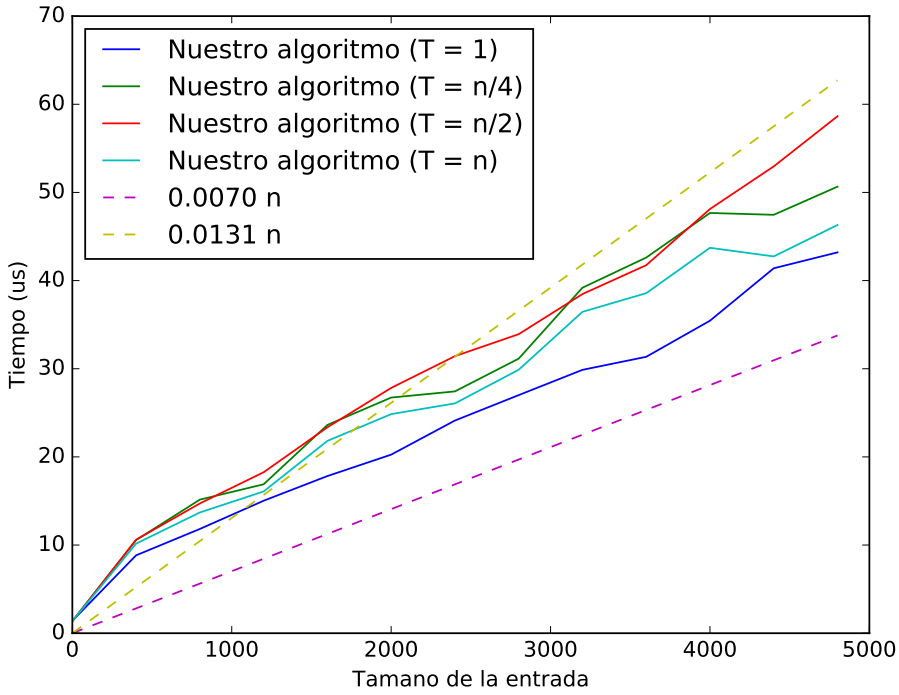

Figura 4: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Como se observa, la implementación tiene complejidad lineal, como era esperado. Sin embargo, se observan diferencias según el T que se elija. Recordemos que el T era el radio de impacto del genkidama. Esto se debe a una cuestión de implementacion. Para los casos donde $T = 1$ o $T = n$, nuestra implementación se ahorra entrar a un while, reduciendo la cantidad de instrucciones que ejecuta cada vez.

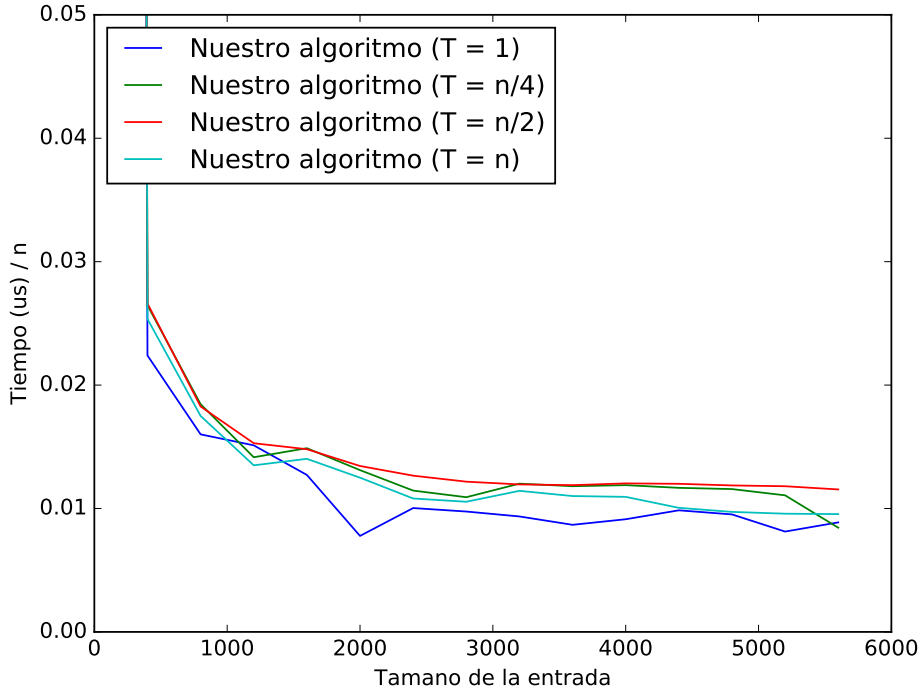

Figura 5: Tiempo que toma el algoritmo en μs dividido n para una entrada de tamaño n.

Este gráfico nos ayuda aún más a confirmar la complejidad lineal de este algoritmo, ayudandonos a ver las constantes para cada caso. Nuevamente, observamos que el caso $T = 1$ tiene una constante un poco menor, debido a lo explicado anteriormente.

Finalmente, veamos que, para instancias de igual tamaño, el tiempo que se tarda en resolverlas no varía demasiado (confirmando el hecho de que para T fijo, la elección de los puntos no cambia el tiempo de ejecución de manera significativa).

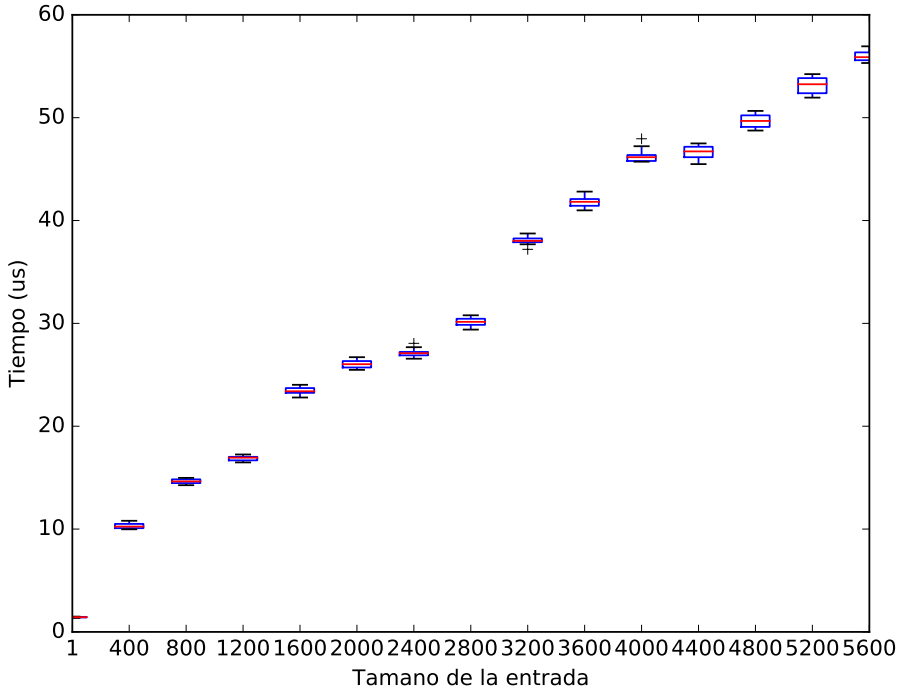

Figura 6: Tiempo que toma el algoritmo en μs para una entrada de tamaño n. $T = n/4$. Se indican los valores del primer al tercer cuatril con un rectángulo azul y la mediana con una linea roja. El máximo y minimo se indican con lineas negras arriba y abajo del rectángulo.

2.4.1. Método de experimentación

Para los primeros dos gráficos (es decir, para la figura y la figura), generamos 50 instancias al azar de para cada n. Las instancias fueron generadas de la siguiente manera: para cada n, elegimos n puntos al azar de la grilla $\{1, ..., n^2\} \times \{1, ..., n^2\}$. Estos puntos eran elegidos al azar usando el mismo *seed* cada vez (para la instancia 1 usabamos 1 como seed, para la instancia 2, usabamos 2, etc.), de tal manera que los experimentos fueran reproducibles de una manera válida.

Cada instancia fue ejecutada 20 veces, y el resultado final era el mínimo de todas las corridas. Luego, tomabamos la mediana de todas las instancias. Es decir, el restultado final es la mediana de los mínimos. Como puede observarse en la figura 6, la mediana es muy representativa de lo que sucede.

3. Kamehameha

3.1. Explicación formal del problema

Sea el conjunto de puntos del primer cuadrante $C = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. Se desea hallar alguna partición, P, del mismo que tenga cardinalidad mínima y cumpla la siguiente condición: $(\forall X \in P)(\exists r : \text{semirrecta})(\forall \textit{punto} \in X)\ \textit{punto} \in r$. Esto es, que cada conjunto de la partición solo puede tener puntos que pertenecen a una misma semirrecta (sea cual sea esta). Notar que la segunda condición no restringe la posibilidad de que r atraviese puntos de otro conjunto de la partición.

Por ejemplo, consideremos $C = \{(3,2), (3,5), (3,7), (5,6), (7,4)\}$. Tomemos la partición $P' = \{\{(3,2), (3,5)\}, \{(3,7)\}\}, \{(5,6), (7,4)\}\}$. Para ver si P' es solución de nuestro problema grafiquemos los puntos y trazemos una posible configuración de semirrectas correspondiente a la partición, como en la figura (8)[1]. Viendo esto es claro que podría atravesarse todos los puntos con solo dos semirrectas en lugar de 3. Inspirados en la figura (7) deducimos que una solución es $P = \{\{(3,2), (3,5), (3,7)\}, \{(5,6), (7,4)\}\}$. En este caso puntual además resulta que es la única solución posible (teniendo en cuenta que no nos importa el orden de los puntos por ser conjuntos). Aunque esto en general no vale, como se ve por ejemplo en las figuras (9) y (10), donde las elecciones de semirrectas inducen particiones claramente distintas.

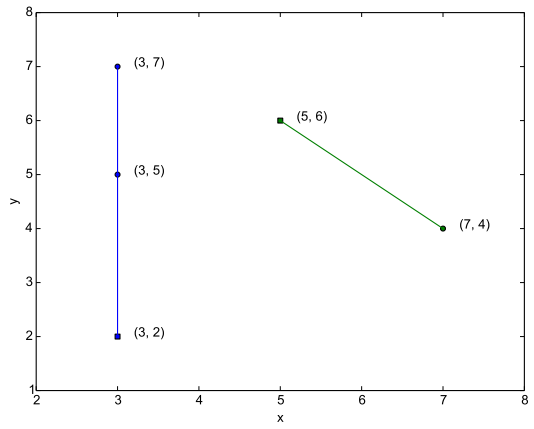

Figura 7: Posible elección de semirrectas para la partición P.

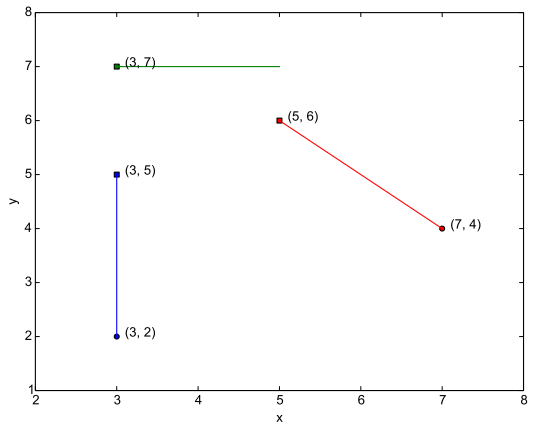

Figura 8: Posible elección de semirrectas para la partición P'.

[1] En esta sección los gráficos usarán la siguiente convención: los puntos considerados están marcados con círculos y cuadrados, representando estos últimos el origen de las semirrectas.

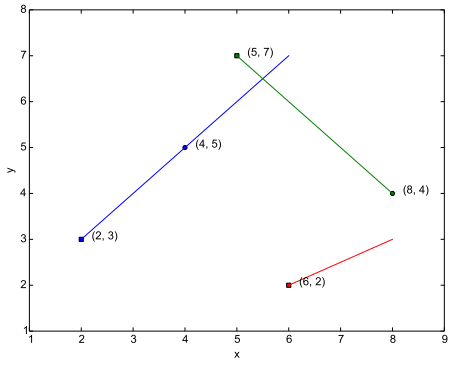

Figura 9: Solución óptima para un conjunto dado de 5 puntos, donde no hay 3 puntos alineados.

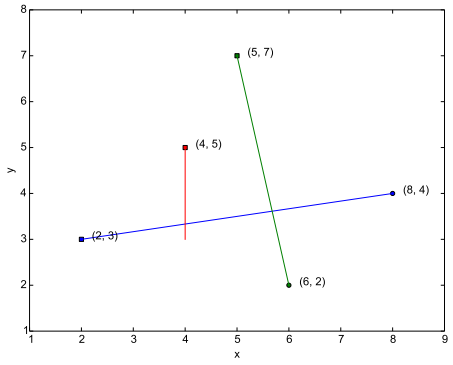

Figura 10: Solución óptima para un conjunto dado de 5 puntos, donde no hay 3 puntos alineados.

Debido a que entre dos puntos siempre se puede establecer un semirrecta que los una, ningún conjunto de la partición solución, P, puede tener menos de dos elementos, salvo quizás uno (como ocurre en la figura 9)). Si tuviera al menos dos conjuntos en P con un solo elemento, XyX', entonces podría considerar P', igual a P salvo que tiene a $X'' = X \cup X'$ (que es válido porque seguro hay una semirrecta que una los dos puntos) y por lo tanto no posee a XyX'. Claramente el cardinal de P' es estrictamente menor al de P, lo que es un absurdo pues P era solución. Luego, P puede tener a lo sumo un conjunto con un elemento.

Por lo dicho en el párrafo anterior, se puede establecer una cota superior al cardinal de la solución cuando tengo n puntos: $\lceil \frac{n}{2} \rceil$ ¿Cuándo se alcanza dicha cota? Cuando no existe ninguna semirrecta que pueda atravesar más de dos puntos. Una disposición fácil de generar este tipo de casos es considerar n puntos esparcidos sobre una circunferencia (una recta solo puede ser tangente o secante respecto a una circunferencia). En la figura (11) puede verse un ejemplo de esta situación para 16 puntos.

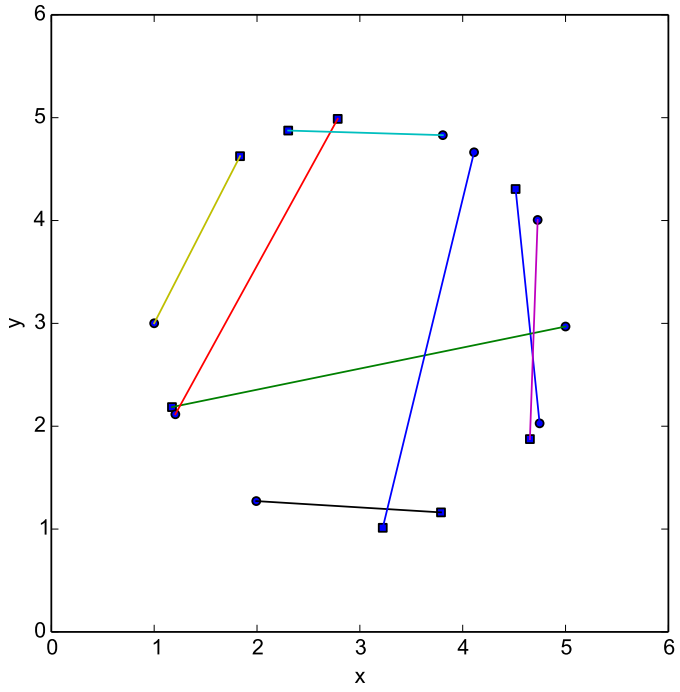

Figura 11: Solución óptima para un conjunto de 16 puntos dispuestos sobre una circunferencia de radio 2 centrada en $(3, 3)$.

3.2. Explicación de la solución

En esta sección daremos una breve explicación de porque el método de `backtracking` resulta correcto para resolver el problema presentado, luego contaremos las podas y estrategias utilizadas, y finalmente presentaremos una explicación de las partes fundamentales de la implementación.

3.2.1. Árbol de posibilidades

Los algoritmos de `backtracking` son un refinamiento de los algoritmos de fuerza bruta por lo que pensemos primero la solución usando este método. Este último tipo de algoritmos consisten en enumerar todos los posibles candidatos a solución del problema y ver si satisfacen las condiciones pedidas o no. En nuestro caso particular, todas los posibles candidatos van a ser las particiones que cumplan la segunda condición del enunciado, y de estas nos quedamos con alguna que tenga cardinal mínimo.

El proceso de construir las particiones se puede pensar como un árbol en el que en cada nivel se agrega un nuevo conjunto de los posibles a la partición que estoy armando. Así, en el nivel 0 (la raíz) tengo un conjunto vacío. En el nivel 1 tengo todos los subconjuntos de C que verifican que existe alguna semirrecta que los une. Al escoger alguno de estos conjuntos lo agregamos en la potencial partición. Debido a esto, en el nivel 2 tenemos un subconjunto de las posibilidades del nivel 1 estrictamente más pequeño pues debemos eliminar todas aquellas opciones que involucraban alguno de los puntos presentes en el conjunto previamente agregado

(los conjuntos de la partición deben ser disjuntos). Se repite un procedimiento análogo hasta completar la partición, momento en el que estaremos parados en una hoja del árbol.

Haciendo DFS (Depth-first search) recorremos todo el árbol de posibilidades, de forma que finalmente encontramos todas las particiones que cumplen la condición dos. Finalmente nos quedamos con alguna de cardinal mínimo (si hubiera más de una).

La correctitud de tal algoritmo es clara pues encuentra todo el universo de posibles soluciones, por lo que si efectivamente existe alguna eventualmente la obtiene. En particular, para todo conjunto finito es posible establecer particiones, así que siempre hay solución.

La idea de `backtracking` es preservar esta garantía de correctitud pero agilizando la búsqueda mediante la utilización de podas en el árbol de posibilidades. Dicho de otra forma, un algoritmo de este tipo descarta `candidatos a soluciones parciales` que a partir de algún criterio podemos predecir que no nos van a conducir a la solución. En nuestro problema, un candidato a solución parcial es una partición en construcción. En la siguiente sección mostramos con que criterios los descartamos.

3.2.2. Podas realizadas

Una estrategia básica, basada en lo observado en la sección 3.1, es no considerar agregar conjuntos que tienen un solo elemento, salvo potencialmente en el último paso. Además, planteamos dos podas:

1. Si actualmente estamos parados en un nodo de nivel s (es decir que el cardinal de nuestro candidato a solución parcial es s) y anteriormente ya habíamos encontrado una solución en el nivel s, entonces no tiene sentido continuar por ese camino pues sabemos que todas los candidatos a soluciones a los que se arrive van a ser peores que un candidato previamente encontrado. De esta forma estamos ahorrándonos una búsqueda innecesaria y potencialmente muy costosa.

2. Si actualmente estamos parados en un nodo de nivel s, y nos movemos a un sucesor en el nivel $s+1$ que resulta ser candidato a solución (y que debe ser mejor que el mejor que pudiéramos haber encontrado antes debido a la poda anterior), al volver al predecesor del nivel s (DFS) no tiene sentido seguir probando con el resto de los sucesores pues no pueden dar candidatos a soluciones mejores que de tamaño $s+1$.

3.2.3. Pseudocódigo

Variables globales: Para tratar el problema definimos cuatro variables globales:

- `Vector(<int, int>)` *puntos*: almacena el conjunto de puntos. Dado que lo guardamos en un vector, se está imponiendo un orden sobre los puntos. Y como *puntos* no se modifica nunca en el transcurso del programa, aprovecharemos dicho orden para referirnos a cada punto por el entero que representa su posición en *puntos*.
- `int` n: cantidad de puntos.
- `int` *mejor*: guarda el tamaño de la mejor partición hallada hasta el momento.

- `Vector(Vector(int))` *mejor_sol*: guarda la mejor partición encontrada hasta el momento.

Clase Tablero: La estructura `Tablero` nos permite condensar bien la información de la solución parcial que estamos desarrollando en un determinado momento junto con lo que necesitamos saber para hallar a sus sucesores. Las tres variables internas que tiene son:

- `Vector(Vector(int))` *solucion*: almacena una candidata solución parcial hallada hasta el momento. Es decir que eventualmente *solucion* es una partición de *puntos*, que puede ser efectivamente o no solución del problema planteado.
- `Vector(bool)` *vivos*: tiene tamaño n. $vivos[i] = true \Leftrightarrow$ el punto i todavía no fue agregado en algún conjunto de *solucion*.
- `int` *cant_vivos*: cantidad de puntos que todavía no fueron añadidos a *solucion*.

En cuanto a las funciones miembro de la clase, consideramos que el código es lo suficientemente simple y los nombres lo suficientemente descriptivos como para que sea necesaria una explicación detallada de cada uno.

Algorithm 2 Pseudocódigo del procedimiento de `solve` en Kamehameha

```
procedure SOLVE(Vector(<int, int>) ptos)→ Vector(Vector(int))
    Puntos ← ptos                                   ▷ O(n)
    Tablero inicial ← Tablero()                     ▷ O(n)
    mejor ← puntos.size()                           ▷ O(1)
    backtracking(inicial, 0)                        ▷ O(T(n))
    return mejor_sol
```

Función `solve`

Algorithm 3 Pseudocódigo del procedimiento de `backtracking` en Kamehameha

```
procedure BACKTRACKING(Tablero t, int step)
    if step ≥ mejor then                                                    ▷ O(1)
        return
    if t.Solucionado() then                                                 ▷ O(n)
        mejor ← step                                                        ▷ O(1)
        mejor_sol ← t.Solucion()                                            ▷ O(n)
    else
        for i ∈ [0, ..., n), t.EstaVivo?(i) do                              ▷ n veces
            if t.SoloQuedaUno?() then                                       ▷ O(1)
                t.Matar(i)                                                  ▷ O(n)
                backtracking(t, step + 1)
                return
            for j ∈ [0, ..., n), j ≠ i ∧ t.EstaVivo?(j) do                  ▷ n veces
                Tablero sucesor ← Copiar(t)                                 ▷ O(2n + 1)
                Vector(int) derrotados ← [i, j]                             ▷ O(1)
                for k ∈ [0, ..., n), k ≠ i ∧ k ≠ j ∧ t.EstaVivo?(k) do      ▷ n veces
                    if x_i ≠ x_j ∧ y_i ≠ y_j then                           ▷ O(1)
                        if (x_k − x_i)/(x_j − x_i) = (y_k − y_i)/(y_j − y_i) ∧ MismoCuadrante?(i, j, k) then   ▷ O(1)
                            derrotados.AgregarAtras(k)                      ▷ O(n)
                    else
                        if (AlinHor?(i, j, k) ∨ AlinVer?(i, j, k)) ∧ MismoCuadrante?(i, j, k) then   ▷ O(1)
                            derrotados.AgregarAtras(k)                      ▷ O(n)
                sucesor.Matar(derrotados)                                   ▷ O(n)
                backtracking(sucesor, step + 1)
                if mejor = s + 1 then                                       ▷ O(1)
                    return
```

Función de `backtracking`

Función `mismoCuadrante`

3.3. Complejidad del algoritmo

3.3.1. Complejidad en peor caso

El peor caso como vimos es cuando no hay tres o más puntos alineados. En tal situación la partición solo estará compuesta por conjuntos de dos puntos (y uno de un elemento si n es impar), por lo que tendrá cardinalidad igual a $\frac{n}{2}$. Por lo tanto, todas las hojas de nuestro árbol estarán a distancia $\frac{n}{2}$ de la raíz. Entonces nuestro algoritmo recorrerá todas las ramas hasta el fondo. Bueno, en rigor esto no es exactamente así debido a las podas. No obstante omitiremos ese detalle en el cálculo de la complejidad, y asumiremos la primer situación pues es al menos tan mala y por lo tanto permite acotar la complejidad de nuestro algoritmo.

La complejidad total del algoritmo es la complejidad de `solve`, $O(n + T(n)) = O(maxn, T(n))$. Veamos entonces cuál es la cota superior de complejidad en peor caso de

`backtracking`.

3.3.2. Ecuación de recurrencia

Primero debemos dar la ecuación de recurrencia, T, para la cantidad de operaciones que realiza *backtracking* con un input de tamño n en el peor caso. Tal ecuación, basados en las complejidades anotadas en el algoritmo 0, es la siguiente

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 9 & \text{si } n = 1 \\ n^3 + n^2 \times T(n-2) & \text{si } n \geq 2 \end{cases} \tag{1}$$

A continuación damos una fórmula cerrada para esta ecuación de recurrecia cuando n es par[2]:

Lema 3.1:

$$\begin{aligned} F(n) &= \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i) \prod_{j=0}^{i} (n-2j)^2 \right) + \left(\prod_{i=0}^{\frac{n}{2}-1} (n-2i)^2 \right) T(0) \\ &= \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i) \prod_{j=0}^{i} (n-2j)^2 \right) + \prod_{i=0}^{\frac{n}{2}-1} (n-2i)^2 \end{aligned} \tag{2}$$

Una demostración de este lema puede encontrarse en el apéndice.

Ahora acotemos $F(n)$:

$$\begin{aligned} F(n) &= \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i) \prod_{j=0}^{i} (n-2j)^2 \right) + \prod_{i=0}^{\frac{n}{2}-1} (n-2i)^2 \\ &\leq \sum_{i=0}^{\frac{n}{2}-1} \left(n \prod_{j=0}^{i} n^2 \right) + \prod_{i=0}^{\frac{n}{2}-1} n^2 \\ &= \sum_{i=0}^{\frac{n}{2}-1} (n^{2i+1}) + n^{2\times(\frac{n}{2}-1)} \\ &= \sum_{i=0}^{\frac{n}{2}-2} (n^{2i+1}) + n^{n-1} + n^{n-2} \\ &\leq \frac{n}{2} n^{n-1} + n^{n-1} \\ &\leq \frac{3}{2} n^n \end{aligned} \tag{3}$$

Finalmente $F(n) = T(n)$ es $O(n^n)$ por lo que también es $O(n^{n+2})$, que era la complejidad pedida.

[2]Para n impar puede obtenerse una fórmula muy parecida, reemplazando el $T(0)$ por $T(1)$ y cambiando $\frac{n}{2}$ por $\frac{n-1}{2}$. Para evitar sobrecargar la demostración nos limitamos al caso par pero el impar es análogo.

3.4. Performance del algoritmo

Como vimos anteriormente, el algoritmo tiene una complejidad de $O(n^{n+2})$. Sin embargo, esta cota no era ajustada. Veamos si esto se refleja en la realidad. El primer caso que analizaremos es el peor caso, es decir, aquel en el que siempre se necesitan $\frac{n}{2}$ rectas, el máximo posible.

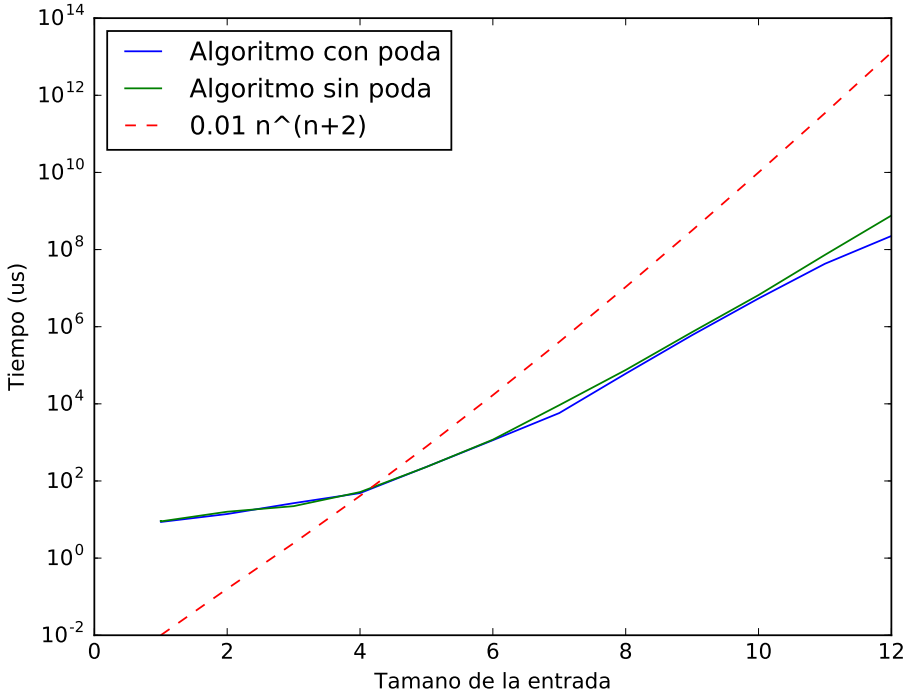

Figura 12: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Como puede verse, el tiempo requerido por el algoritmo crece muy aceleradamente, razón por la cual debimos analizarlo hasta $n = 12$, para el cual cada corrida tardaba alrededor de 15 minutos (Nótese que si $n = 12$, entonces $n^{n+2} \equiv 10^15$).

Como este es el peor caso, la poda no contribuye en nada, dado que hay que recorrer cada posibilidad de entre todas las combinaciones. Esto se refleja claramente en el gráfico.

Analicemos ahora el caso promedio.

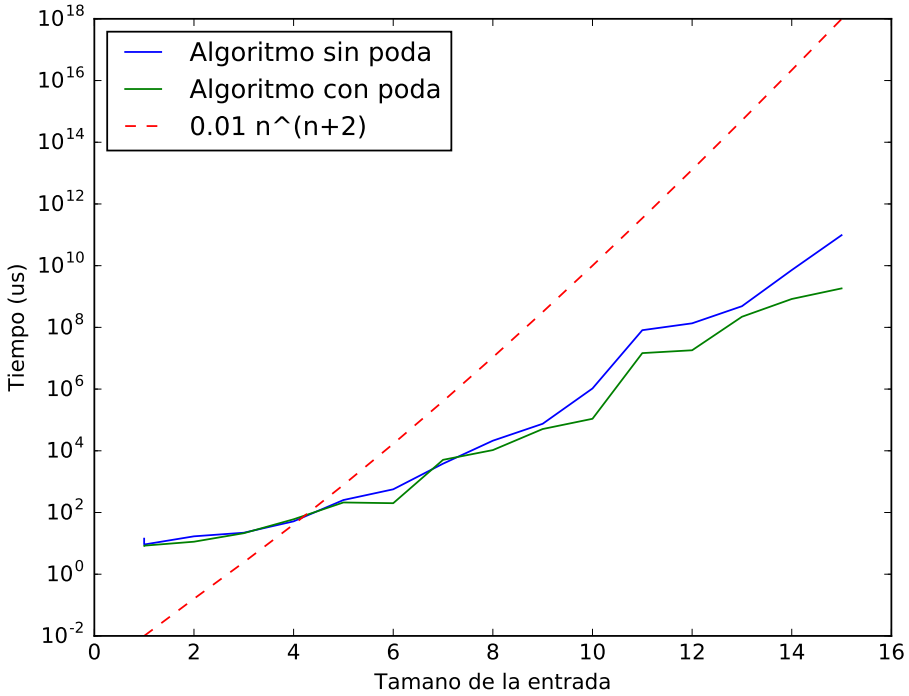

Figura 13: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Como puede verse en la figura 13, el caso promedio es muy similar al peor caso. Esto se debe, a que lo mas probable es que se necesiten $\frac{n}{2}$ rectas para cubrir a todos los puntos. Esto se prueba en el apéndice.

Por último, analicemos el mejor caso. El mejor caso es, obviamente, en el que se necesita una sola recta para cubrir a todos los puntos, y esta recta se encuentra en el primer intento. Como vimos antes, en este caso la complejidad del algoritmo es de $O(n)$. Veamos si esto se confirma experimentalmente.

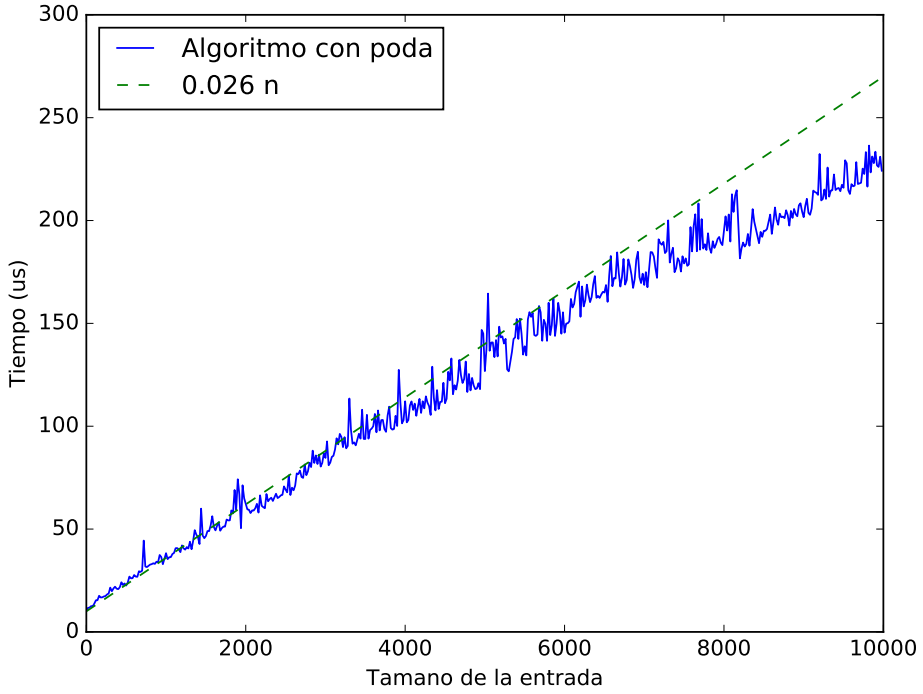

Figura 14: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

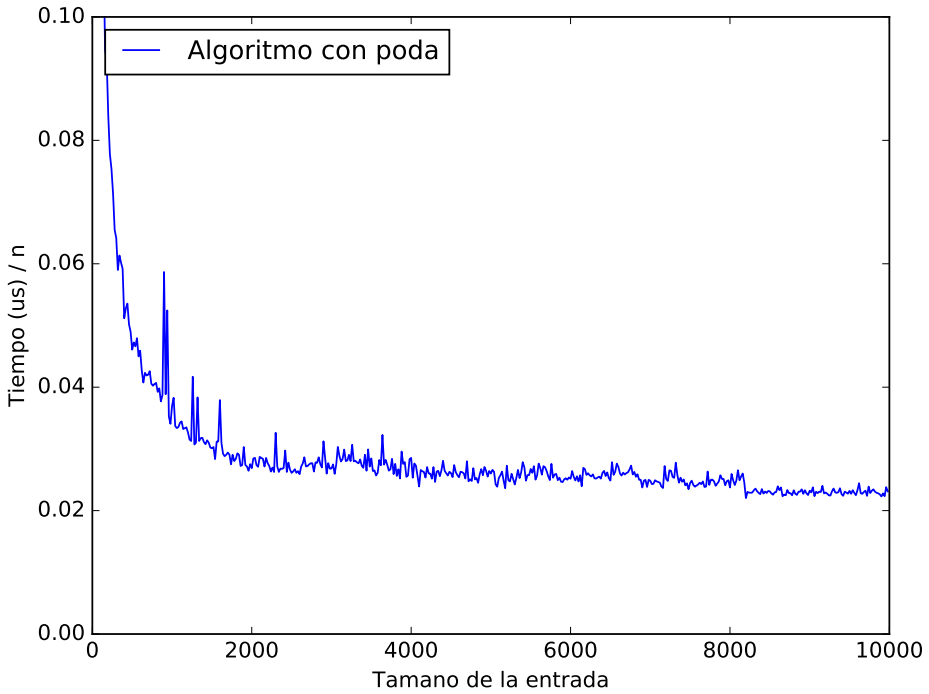

Figura 15: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Efectivamente, se ve que la complejidad del algoritmo es de $O(n)$. La disminución del tiempo de ejecución alrededor de $n = 8200$ no pudo ser explicada por nosotros. Dado que 8192

es una potencia de 2, podemos inferir que tiene que ver con un tema de *cache* al almacenar los datos en memoria.

3.4.1. Método de experimentación

Para el peor caso, para asegurarnos que siempre se requirieran exactamente $\frac{n}{2}$ rectas para cubrir todos los puntos. Para esto lo que hicimos fue tomar n puntos sobre una circunferencia (específicamente, sobre la circunferencia de radio $\frac{n}{2}$ con centro en el (n, n)). Nótese que (por la definición de circunferencia) no hay 3 puntos en la misma recta, por lo tanto necesitaremos $\frac{n}{2}$ rectas para cubrir todos los puntos.

El caso promedio se genera de manera similar al problema anterior. Para cada n, elegimos n puntos al azar de la grilla $\{1, ..., n^2\} \times \{1, ..., n^2\}$.

Estos puntos eran elegidos al azar usando el mismo *seed* cada vez (para la instancia 1 usabamos 1 como seed, para la instancia 2, usabamos 2, etc.), de tal manera que los experimentos fueran reproducibles de una manera válida. La cantidad de instancias que creamos para cada n fue 40. Cada instancia fue ejecutada 20 veces, y el resultado final era el mínimo de todas las corridas. Luego, tomabamos la mediana de todas las instancias. Es decir, el restultado final es la mediana de los mínimos.

Finalmente, para el mejor caso, el caso de tamaño n estará formado por los puntos $\{(1, 1), (2, 2), \cdots, (n, n)\}$, en ese orden, de tal manera que el backtracking tome a los puntos $(1, 1)$ y $(2, 2)$ en su primer intento, y ese intento sea el exitoso.

4. Apéndice

4.1. Demostración del Lema 3.1

Vamos a demostrar que $T(n) = F(n)$, si $n = 2k$ con $k \in \mathbb{N}$, por inducción en k.

Caso base: $k = 0 \implies n = 0$

$T(0) = 1$

Dado que las sumatorias y las productorias en las que el límite inferior es más grande que el superior se devuelve el elemento neutro para la suma y el producto, 0 y 1 respectivamente, tenemos que $F(0) = 1$. Entonces demostramos el caso base.

Paso inductivo: Supongamos que $T(n) = F(n)$. Veamos que también vale $T(n+2) = F(n+2)$

$$\begin{aligned} T(n+2) &= (n+2)^3 + (n+2)^2 T(n) \\ &= (n+2)^3 + (n+2)^2 F(n) \\ &= (n+2)^3 + (n+2)^2 \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i) \prod_{j=0}^{i} (n-2j)^2 \right) + (n+2)^2 \prod_{i=0}^{\frac{n}{2}-1} (n-2i)^2 \\ &= (n+2)^3 + \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i)(n+2)^2 \prod_{j=0}^{i} (n-2j)^2 \right) + (n+2)^2 \prod_{i=0}^{\frac{n}{2}-1} (n-2i)^2 \end{aligned} \tag{4}$$

Por otro lado tenemos

$F(n+2) = \sum_{i=0}^{\frac{n}{2}} \left((n+2-2i) \prod_{j=0}^{i} (n+2-2j)^2 \right) + \prod_{i=0}^{\frac{n}{2}} (n+2-2i)^2$

Ahora hagamos un par de observaciones:

$$\begin{aligned} (n+2)^2 \prod_{j=0}^{i} (n-2j)^2 &= (n+2)^2 \prod_{t=1}^{i+1} (n-2(t-1))^2 \\ &= (n+2)^2 \prod_{t=1}^{i+1} (n+2-2t))^2 \\ &= \prod_{t=0}^{i+1} (n+2-2t)^2 \end{aligned} \tag{5}$$

$$\begin{aligned} (n+2)^3 + \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i) \prod_{t=0}^{i+1} (\ldots) \right) &= (n+2)^3 + \sum_{r=1}^{\frac{n}{2}} \left((n-2(r-1)) \prod_{t=0}^{r} (\ldots) \right) \\ &= (n+2)^3 + \sum_{r=1}^{\frac{n}{2}} \left((n+2-2r)) \prod_{t=0}^{r} (\ldots) \right) \\ &= \sum_{r=0}^{\frac{n}{2}} \left((n+2-2r)) \prod_{t=0}^{r} (\ldots) \right) \end{aligned} \tag{6}$$

Notar que la igualdad 5 puede instanciarse para $i = \frac{n}{2} - 1$. Aplicando las igualdades 5 (dos veces) y 6 sobre la ecuación 4, llegamos a que $T(n+2) = F(n+2)$. $\square$