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 Ezequiel	200/14	axel.maddonni@gmail.com

Índice

1. Kaio Ken **3**
1.1. Explicación formal del problema . . . 3
1.2. Explicación de la solución . . . 3
1.2.1. Correctitud . . . 5
1.2.2. Optimalidad . . . 5
1.3. Complejidad del algoritmo . . . 6
1.3.1. Esbozo del algoritmo . . . 6
1.3.2. Análisis temporal . . . 7
1.4. Performance del algoritmo . . . 7
1.4.1. Método de experimentación . . . 10

2. Genkidama **11**
2.1. Explicación formal del problema . . . 11
2.2. Explicación de la solución . . . 12
2.2.1. Optimalidad . . . 13
2.3. Complejidad del algoritmo . . . 14
2.4. Performance del algoritmo . . . 15
2.4.1. Método de experimentación . . . 17

3. Kamehameha **18**
3.1. Explicación formal del problema . . . 18
3.2. Explicación de la solución . . . 20
3.2.1. Árbol de posibilidades . . . 20
3.2.2. Podas realizadas . . . 21
3.2.3. Pseudocódigo . . . 21
3.3. Complejidad del algoritmo . . . 24
3.3.1. Complejidad en peor caso . . . 24
3.3.2. Complejidad en mejor caso . . . 26
3.4. Performance del algoritmo . . . 26
3.4.1. Método de experimentación . . . 30

4. Apéndice **31**
4.1. Demostración del Lema 3.1 . . . 31
4.2. Kamehameha: el caso promedio se parece al peor caso . . . 32

1. Kaio Ken

1.1. Explicación formal del problema

Para entender mejor el problema, lo especificaremos usando términos matemáticos. Se tiene un conjunto de n elementos que representan a los guerreros.

$$G = \{g_1, ..., g_n\}$$

Definiremos una *partición* (*pelea* en el enunciado) como un par de dos conjuntos disjuntos no vacíos de elementos A y B (donde cada uno representa cada bando en una pelea). En cada partición, la unión de dichos conjuntos es el conjunto total de los elementos (es decir, todos los guerreros participan de todas las peleas):

$$P = (A, B)$$

donde A, B son conjuntos de elementos y

$$A \dot{\cup} B = G.$$

Notaremos P_A al conjunto A de elementos que representa al primer elemento de la partición P, y P_B al del otro. Además, para todo elemento g_k, notaremos $B_P(g_k)$ (bando, en los términos del enunciado) al conjunto de la particion P al que g_k pertenece. O sea que los valores que puede tomar $B_P(g_k)$ son P_A o P_B.

Dada una cantidad de elementos n, se pide encontrar el conjunto de tamaño mínimo de particiones de modo que para todo par de elementos (g_i, g_j), exista particion P tal que $B_P(g_i) \neq B_P(g_j)$.

ACA VAN A IR UNOS DIBUJITOS

Notar que puede haber más de una solución correcta. El algoritmo propuesto devuelve una de ellas.

1.2. Explicación de la solución

Para explicar la solución, mezclaremos el lenguaje del problema original con el lenguaje formal, usando el que creamos que hará más clara la explicación en cada momento.

Para resolver el problema, utilizamos un algoritmo usando la técnica de **divide and conquer**, es decir, dividimos el problema en subproblemas más pequeños, los resolvemos, y luego combinamos las soluciones para lograr la solución al problema total.

Para entender cómo lo aplicamos para resolver el problema, consideremos la primera partición: P_1, cuando todavía no comenzaron los guerreros a pelear. Para dicha pelea, se eligen guerreros para el bando P_{1A} y para el bando P_{1B}, de modo que luego de la pelea, todos los guerreros del bando A hubieron peleado contra los del B y viceversa. De este modo, a los guerreros del bando A, les faltará solamente pelear entre sí, al igual que los del bando B.

Ahora bien, tenemos dos subproblemas separados:

- Por un lado, necesitamos que todos los guerreros de P_{1A} peleen entre sí, en la cantidad mínima de peleas posible.
- Por otro lado, buscaremos lo mismo para los guerreros de P_{1B}.

Una vez resueltos estos subproblemas, la manera de combinar las soluciones es simple. Supongamos que para la siguiente pelea, los guerreros de P_{1A} se dividen en dos bandos: C y D, y los guerreros de P_{1B} se dividen en los bandos E y F. Podemos combinar la solución en una única pelea P_2, tomamos $P_2 = (C \cup E,\ D \cup F)$, de modo que en una misma pelea se logren enfrentar los guerreros de C y D, y de E y F. De la misma forma procedemos para las siguientes peleas.

Podría pasar que cuando aplicamos el algoritmo para resolver los dos subproblemas, ambas subsoluciones resulten con distinta cantidad de peleas necesarias para resolver uno y otro. En ese caso necesitaremos como mínimo el máximo entre las dos. Los guerreros que ya pelearon contra todos sus contrincantes para dichas peleas, se asignarán a cualquier bando ya que es indistinto.

Repetimos el mecanismo hasta llegar al caso base, correspondiente a cuando queremos hacer que 2 guerreros peleen en la mínima cantidad de peleas necesarias. En este caso sólo se necesita 1 pelea, un guerrero para cada cada bando. En caso de que la cantidad de guerreros de un conjunto sea impar, llegaremos al caso $n = 1$. Este caso proviene de dividir previamente un conjunto de 3 guerreros g_1, g_2, g_3 en los bandos g_1 y g_2, g_3 respectivamente, de modo que el guerrero que queda solo (g_1) ya peleó contra los otros dos, y hubo completado todas las peleas contra los demás guerreros, así que el caso $n = 1$ está resuelto.

Para dividir en subproblemas, el algoritmo presentado divide al conjunto inicial de guerreros a la mitad, es decir, los subconjuntos generados para cada bando son:

$$A = \{g_1, ..., g_{\frac{n}{2}}\}$$

$$B = \{g_{\frac{n}{2}+1}, ..., g_n\}$$

y luego aplicamos recursión para aplicar el algoritmo a cada bando. Veamos el pseudocódigo:

Los resultados obtenidos se van guardando en una matriz (utilizando índices pasados por parámetro para saber qué lugar corresponde a cada guerrero), de forma que una vez calculadas todas las peleas, se imprima la matriz que indica el bando (1 o 2) de cada uno en cada una de las peleas. Esta matriz tiene un tamaño fijo de n columnas y *cantpeleas* filas, donde

$$cantpeleas = \lceil \log_2 n \rceil.$$

es decir, hay una fórmula cerrada para saber la cantidad de peleas que genera nuestro algoritmo, y proviene de dividir en cada paso el n en mitades iguales. Es decir, por ejemplo:

- Para $n = 4$, $cantpeleas = 2$.
- Para $n = 5, ..., 8$, $cantpeleas = 3$.
- Para $n = 9$, $cantpeleas = 4$.
- etc...

En la siguiente sección demostraremos por qué este algoritmo resuelve el problema de manera óptima.

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)
```

1.2.1. Correctitud

A diferencia del resto de los problemas, en los que probaremos en conjunto correctitud y optimalidad, en este problema optamos por separar las demostraciones para hacerlas más simples para el lector.

Proposición. El algoritmo es correcto. Además, si $n > 1$, siempre va a haber un elemento al que le asigne 1.

Observación. Con que el algoritmo es correcto nos referimos

1.2.2. Optimalidad

Demostraremos que el algoritmo propuesto para resolver el problema es correcto y óptimo, es decir, devuelve un conjunto de peleas de tamaño mínimo para que todos los guerreros hayan peleado entre sí. Usaremos **inducción global en n**:

Sea $p(n)$ = "El algoritmo D&C propuesto resuelve el problema para n guerreros de manera óptima, generando $\lceil \log_2 n \rceil$ peleas".

Caso Base: $p(2)$. El algoritmo lo resuelve en una sola pelea, donde cada guerrero pertenece a un bando distinto, y es la mejor manera de resolverlo dado que no hay otra manera de que peleen entre sí sin generar al menos 1 pelea.

Paso Inductivo: Supongo que vale HI: $\forall k \in \mathbb{N}, k \leq n,\ vale\ p(k)$, con $n \geq 2$. Quiero ver que vale $p(n+1)$. Tengo $n+1$ guerreros. Para la primera pelea, tengo que dividir los guerreros en dos bandos A y B, de modo que se generan 2 subproblemas de tamaño $|A|$ y $|B|$. Por HI, dichos subproblemas los puedo resolver óptimamente, generando $\lceil \log_2 k \rceil$ peleas, donde k es la cantidad de guerreros del subproblema. Quiero ver de qué manera conviene dividir el conjunto inicial de guerreros para que la cantidad de peleas total del problema sea la mínima. Aplicando el algoritmo en cada bando, la *cantpeleas* total del problema se puede calcular como:

$$cantpeleas = 1 + max\{\lceil \log_2 |A| \rceil, \lceil \log_2 |B| \rceil\}$$

Esta cantidad se minimiza tomando $|A| = |B| = \frac{n+1}{2}$, en caso de que $n+1$ sea par, o $|A| = \frac{n+1}{2}$ y $|B| = \frac{n+1}{2} + 1$ en caso de que sea impar. Entonces, demostramos que aplicando el algoritmo de D&C para $n+1$ también llegamos a una solución óptima.

Falta ver que, para $n+1$, $cantpeleas = \lceil \log_2(n+1) \rceil$.

Si $n+1$ es par, entonces,

$$\begin{aligned} cantpeleas =& 1 + \lceil \log_2 |A| \rceil \\ cantpeleas =& 1 + \lceil \log_2 \frac{n+1}{2} \rceil \\ cantpeleas =& 1 + \lceil \log_2(n+1) - 1 \rceil \\ cantpeleas =& 1 + \lceil \log_2(n+1) \rceil - 1 \\ cantpeleas =& \lceil \log_2(n+1) \rceil \end{aligned} \tag{1}$$

que es lo que queríamos probar.

Si $n+1$ no es potencia de dos, entonces puede o no ser par. Si $n+1$ es impar (y no es potencia de 2), entonces existe $k > 0$ tal que $2^k < n+1 < 2^{k+1}$. Además, podemos tomar $A = \lceil \frac{n+1}{2} \rceil = \frac{n+2}{2}$ y $B = \lfloor \frac{n+1}{2} \rfloor = \frac{n}{2}$. Ahora bien, calculemos *cantpeleas*,

$$\begin{aligned} cantpeleas =& 1 + \lceil \log_2 |A| \rceil \\ cantpeleas =& 1 + \lceil \log_2 \frac{n+2}{2} \rceil \\ cantpeleas =& 1 + \lceil \log_2(n+2) \rceil - 1 \\ cantpeleas =& \lceil \log_2(n+2) \rceil \end{aligned} \tag{2}$$

Ahora bien, notemos que, como dijimos antes, $2^k < n+1 < 2^{k+1}$. Entonces, $2^k < n+1 < n+2 \leq 2^{k+1}$. Por lo tanto, $k < \log_2(n+1) < \log_2(n+2) \leq k+1$. Por lo tanto, $\lceil \log_2(n+1) \rceil = \lceil \log_2(n+2) \rceil = k+1$, por lo tanto

$$cantpeleas = \lceil \log_2(n+1) \rceil$$

que es lo que queríamos ver.

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.

Como puede verse en el pseudocódigo, 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 $\Theta(n \log n)$.

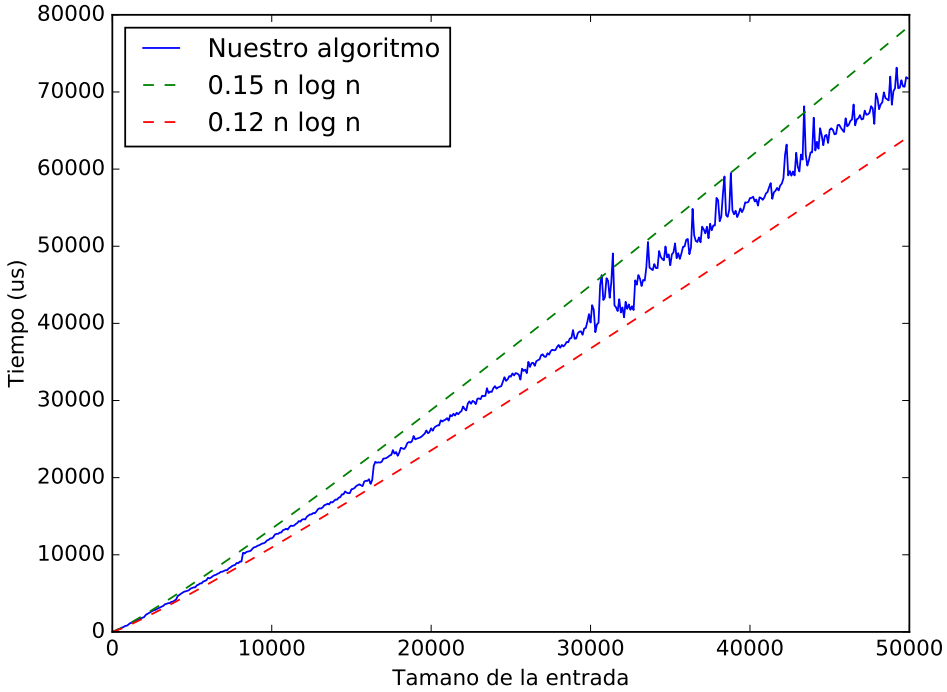

Figura 1: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

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.

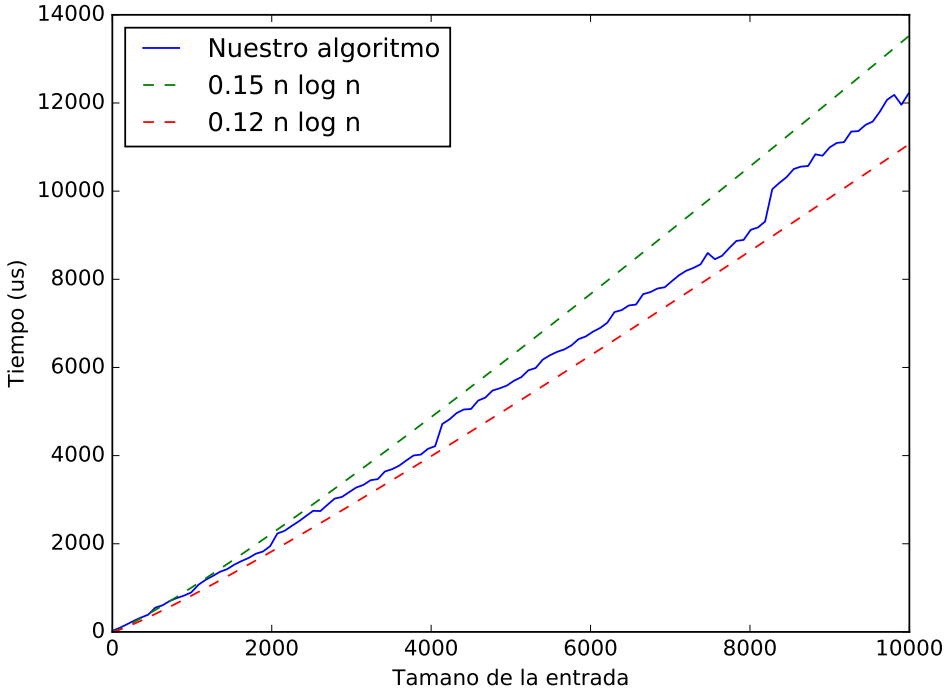

Figura 2: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

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.

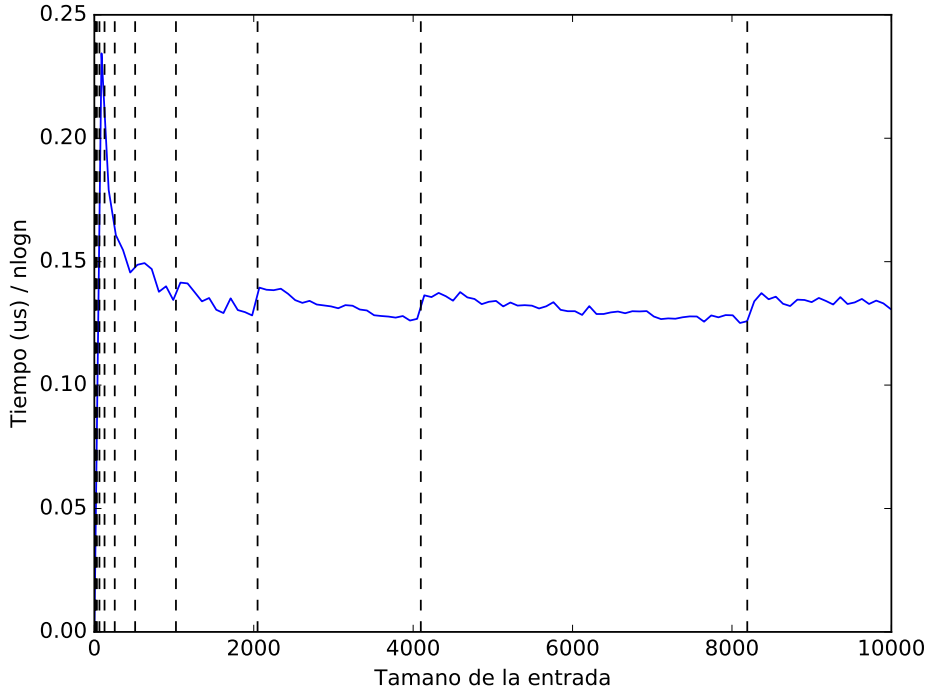

Figura 3: Tiempo que toma el algoritmo en μs dividido $n \log n$para una entrada de tamaño n.

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

El ejercicio tiene una bella historia detras pero a la hora de pensar el problema resulta útil abastraernos un poco y pensarlo en terminos de objetos matemáticos. Sobre el plano tenemos un conjunto de puntos $\{(X_1, Y_1), (X_2, Y_2), (X_3, Y_3), ..., (X_n, Y_n)\}$ todos sus elementos cumplen que $X_1 > X_2 > ... > X_n \geq 0$ y $0 \leq Y_1 < Y_2 < .. < Y_n$. O sea están todos repartidos en el primer cuadrante y formando una especie de escalera decreciente de izquierda a derecha. El objetivo es cubrir a todos los puntos con la menor cantidad de rectangulos que cumplan que sus lados son paralelos a los ejes y sus extremos estan en $(0, 0)$ y $(X_i + T, Y_i + T)$. Donde T es una número entero pasado como parametro y (X_i, Y_i) son las coordenadas de algún punto del conjunto.

Esto se debe resolver en $O(n)$ y expresar cuales puntos del conjunto se utilizan como extremos de los rectángulos.

Veamos algunos ejemplos de soluciones correctas e incorrectas para terminar de entender lo que plantea el ejercicio. Analicemos el conjunto de puntos $\{(4, 2), (3, 3), (2, 4), (1, 5)\}$ y T = 1.

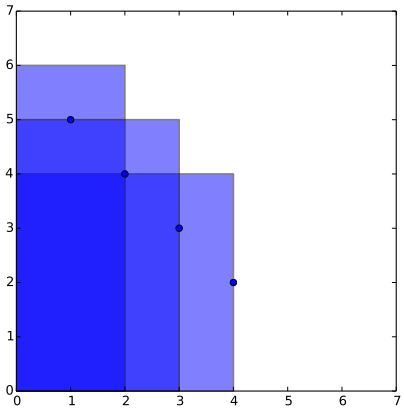

Figura 4: Posible elección de rectangulos no óptima.

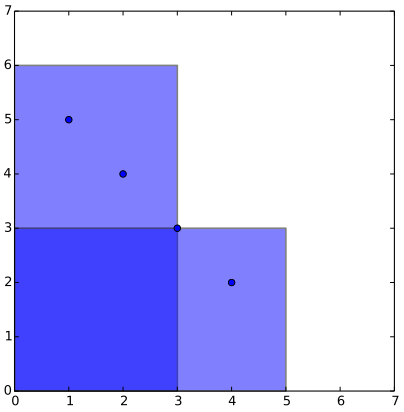

Figura 5: Posible elección de rectangulos incorrecta.

Algunos ejercicios pueden tener varias soluciones posibles. En este caso tenemos estas dos posibilidades.

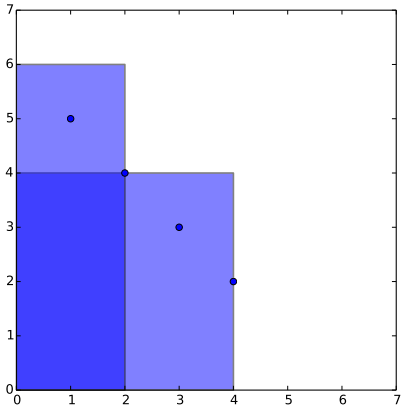

Figura 6: Posible elección de rectangulos óptima y correcta.

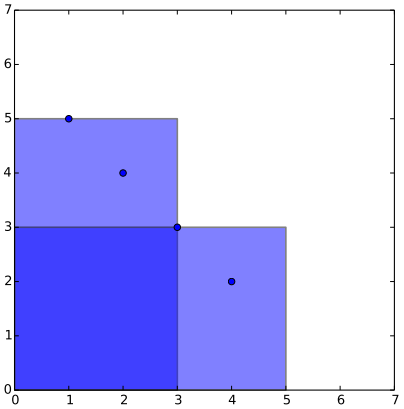

Figura 7: Posible elección de rectangulos óptima y correcta.

2.2. Explicación de la solución

Para resolver el ejercicio utilizamos un *algoritmo goloso*. La idea es ir cubriendo todos los puntos de derecha a izquierda utilizando la menor cantidad de rectangulos posible. La técnica que utilizaremos para lograr esto, la cual veremos luego que es óptima, será por cada iteración del algoritmo buscar un rectángulo que cubra al punto mas a la derecha y al mismo tiempo cubra a la mayor cantidad posible de otros puntos, o sea en cada iteración intentará hacer lo mejor posible para ese momento en particular, por eso lo *greedy* o *goloso*.

Para la implementación del algoritmo tenemos como datos de entrada los puntos y el valor de T, utilizamos una variable j la cual representa el punto mas a la derecha, al cual vamos a cubrir si o si en esa iteración y una variable i la cual representa el candidato a extremo del rectángulo. Comenzamos por el punto mas a la derecha y vamos siguiendo hacia los que tienen una coordenada x menor. Como sabemos que los puntos cumplen que $X_1 > X_2 > ... > X_n \geq 0$ entonces simplemente arrancamos por X_1 y seguimos hasta X_n, esto se hace facilmente recorriendo el vector *puntos* desde su posición 0 hasta $n-1$. El ciclo principal se encarga de buscar en cada iteración el mejor punto para utilizar como extremo de rectángulo y termina cuando j alcanza a n, lo cual implica que ya se cubrieron todos los puntos.

Veamos el pseudocodigo para que termine de quedar claro todo esto.

Algorithm 2 Esbozo del algoritmo de Genkidama

```
procedure aQuienDisparo(vector(int,int) puntos, int t)
    int n = puntos.cantidad()
    int i = 0
    int j = 0
    while j < n do
        i = j
        while ... do
            i++
        respuesta.agregarA(i)
        i++
        while ... do
            j++
```

Para lograr hacer lo dicho previamente el ciclo principal cuenta con dos bucles anidados.

El primer ciclo tiene como objetivo cubrir a j y al mismo tiempo verificar si no puede cubrir algunos mas. En el pseudocódigo se puede ver facilmente como hacemos eso.

Algorithm 3 Ciclo Anidado 1

```
while i < n − 1 ∧ puntos[i + 1].primero + t ≥ puntos[j].first do
    i++
```

Si podemos cubrir a un punto mas al mismo tiempo que seguimos cubriendo al punto (X_i, Y_i) entonces guardamos ese nuevo valor como nuestro nuevo candidato para utilizar como extremo del nuevo rectangulo. Repetimos esto hasta que ya consideramos todos los puntos o que el siguiente a nuestro candidato ya no nos permite cubrir a (X_i, Y_i).

Cuando conseguimos al mejor candidato, lo almacenamos y seguimos con el Ciclo Anidado 2.

El segundo ciclo tiene como tarea verificar si al haber agregado el nuevo rectangulo no se cubrieron algunos puntos mas a la izquierda.

Algorithm 4 Ciclo Anidado 2

```
while j < n ∧ puntos[j].segundo ≤ puntos[i − 1].segundo + T do
    j++
```

En ese caso se aumenta j para que pase a representar al siguiente punto. Se repite esto hasta encontrar uno punto que no haya sido cubierto todavia o hasta que no hayan mas puntos.

2.2.1. Optimalidad

Para demostrar que el algoritmo porpuesto es óptimo supondre que no lo es y llegaré a un absurdo. Supongamos que tenemos un algoritmo que si resuelve el problema de manera

óptima, tiene que hacer algo distinto a lo que plantea nuestro algoritmo ya que supusimos que no era óptimo. Para diferenciarse del nuestro tiene que tomar al menos una desición distinta.

Dada la solución que suponemos es óptima, nos ponemos a ver las elecciónes de rectangulos que tomó este algoritmo de derecha a izquierda. O sea viendo primero al que tiene como extremo al punto de mayor coordenada x y siguiendo hacia los menores.

Al ponernos a ver elecciones de rectangulos en algun momento vamos a notar que el algoritmo escogió tapar al punto que estaba cubrir mas abajo a la derecha pero sin tapar al mismo tiempo a la mayor cantidad posible de otros puntos sin cubrir. Sabemos que esto va a pasar en algun momento ya que si no fuera asi entonces sería exactamente igual a nuestro algoritmo.

Pero al ver esto se vuelve obvio que esa desición es peor o igual que lo que hacemos, ya que con nuestro algoritmo podriamos reemplazar esa acción por una que cubrá una mayor cantidad de puntos. Terminando en el peor de los casos usando la misma cantidad de rectangulos para la solución. Lo cual es absurdo ya que habiamos supuesto que nuestro algoritmo no era óptimo.

2.3. Complejidad del algoritmo

En esta sección demostraremos que el algoritmo tiene una complejidad de $\theta(n)$ donde n es la cantidad de puntos en el plano para cualquier caso (no existe peor ni mejor caso). O sea está acotado inferior y superiormente por n.

Primero se crean variables lo cual cuesta a lo sumo O(n), dependiendo la estructura utilizada, por tener que averiguar la cantidad de puntos en el vector de entrada.

Lo siguiente es el ciclo principal el cual veremos que toma $\theta(n)$. Veamos una serie de proposiciones que utilizaré para demostrarlo, todas son evidentes viendo el pseudocódigo.

1. Los ciclos 1 y 2 cuestan O(1) por iteración.
2. La variable j aumenta una vez por cada iteracion de cada ciclo en que entra. (En cualquier de los 3 ciclos que hay).
3. Si j vale n entonces terminan todos los ciclos.

Por (3) sabemos que el ciclo solo funcionara mientras j sea menor a n, mientras esto se cumpla se iterara en el ciclo principal y por (1) sabemos que se gastará una cantidad constante de pasos cada vez que se itere en uno de los ciclos internos, por la proposición (2) sabemos que al final de cada iteración del ciclo principal el valor de j será aumentado la cantidad de veces que haya ciclado dentro del primer ciclo anidado mas la cantidad de veces que lo haya hecho en el segundo ciclo anidado mas uno por la iteración principal. Si este valor llega a n el ciclo terminará. Si no ocurre entonces se hará una iteración mas en el ciclo principal. Por lo tanto vamos gastando O(1) cada vez que incrementamos en 1 a j. Y j termina cuando vale n por lo tanto este algoritmo tarda $\theta(n)$, solo nos podemos pasar de ese costo si j llega a valer mas de n y eso no puede pasar por la proposición (3). Tampoco vamos a tardar menos que eso por que eso significaría que j vale menos de n y no puede terminar en ese caso.

Habiendo demostrado esto, sumado a que las instrucciones fuera del ciclo son todas a lo sumo O(n) queda demostrado que la función esta acotada superior e inferiormente por n, para cualquier caso.

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.

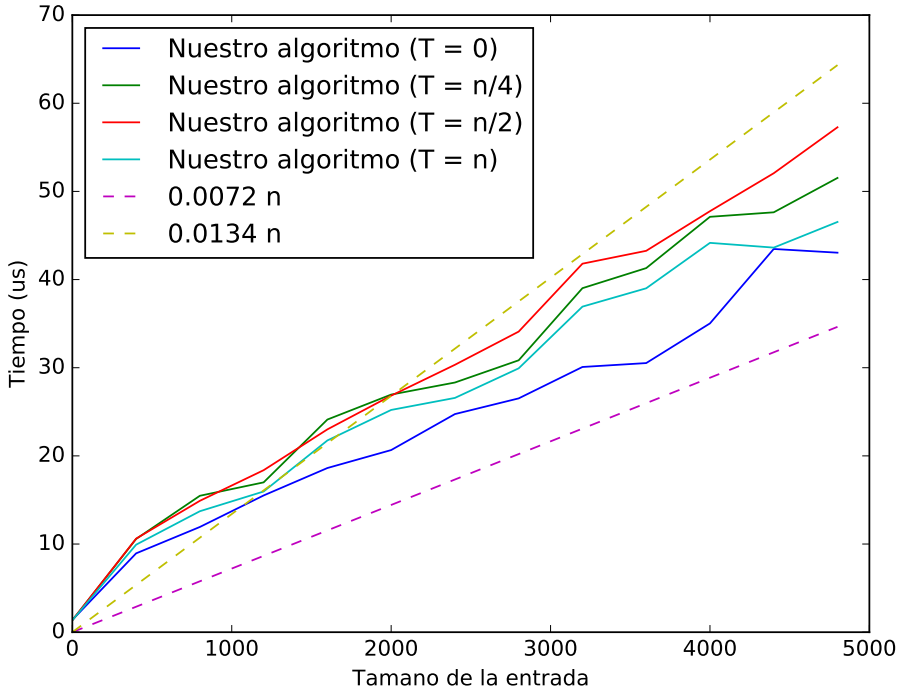

Figura 8: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

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.

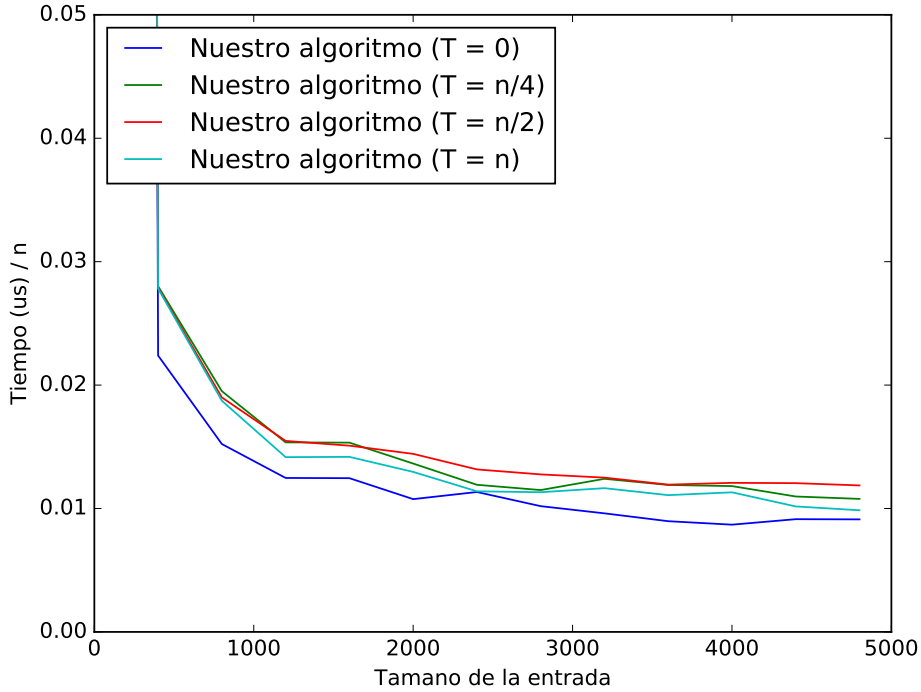

Figura 9: Tiempo que toma el algoritmo en μs dividido n para una entrada de tamaño n.

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).

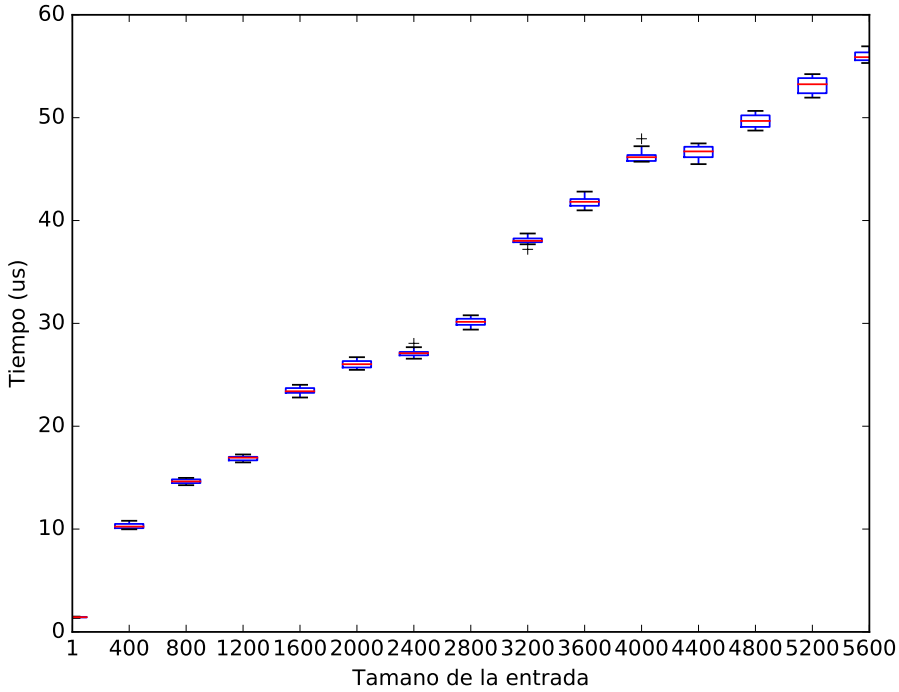

Figura 10: Tiempo que toma el algoritmo en μs para una entrada de tamaño n. $T = n/4$. Se indican los valores del primer al tercer cuatril con un rectángulo azul y la mediana con una linea roja. El máximo y minimo se indican con lineas negras arriba y abajo del rectángulo.

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 10, 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 cumpla las siguientes dos propiedades:

1. P tiene cardinalidad mínima;
2. $(\forall X \in P)(\exists r : \text{semirrecta})(\forall \textit{punto} \in X)\ \textit{punto} \in r$. 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 (12)[1]. Viendo esto es claro que podría atravesarse todos los puntos con solo dos semirrectas en lugar de 3. Inspirados en la figura (11) 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 (13) y (14), donde las elecciones de semirrectas inducen particiones claramente distintas.

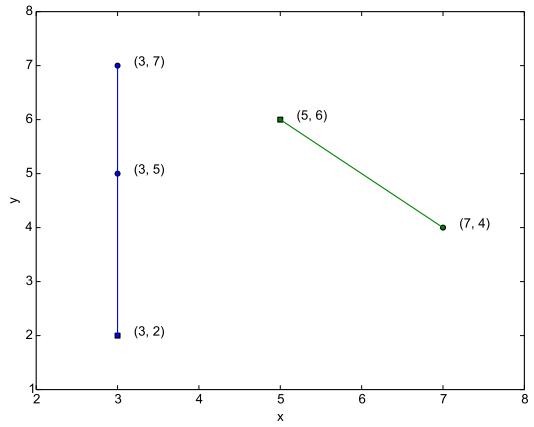

Figura 11: Posible elección de semirrectas para la partición P.

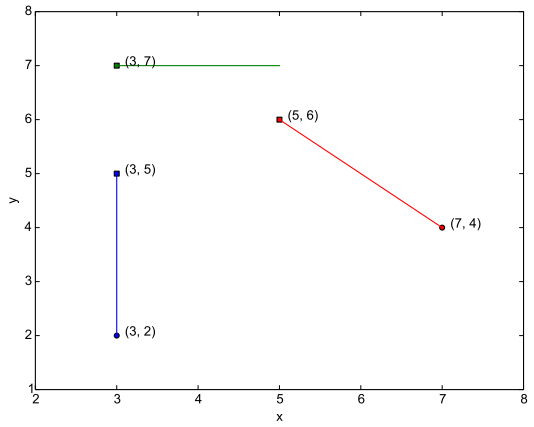

Figura 12: Posible elección de semirrectas para la partición P'.

[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.

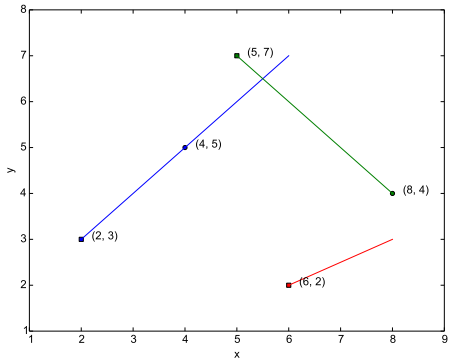

Figura 13: Solución óptima para un conjunto dado de 5 puntos, donde no hay 3 puntos alineados.

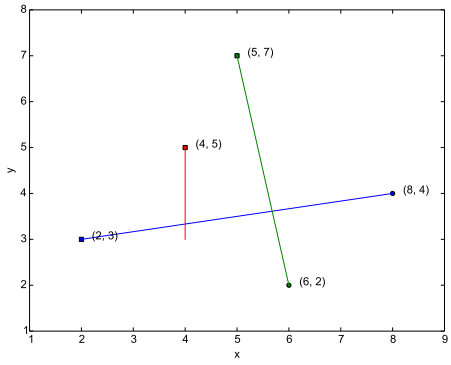

Figura 14: Solución óptima para un conjunto dado de 5 puntos, donde no hay 3 puntos alineados.

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 (13)). Si tuviera al menos dos conjuntos en P con un solo elemento, X y X', 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 X y X'. 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 forma 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 (15) puede verse un ejemplo de esta situación para 16 puntos.

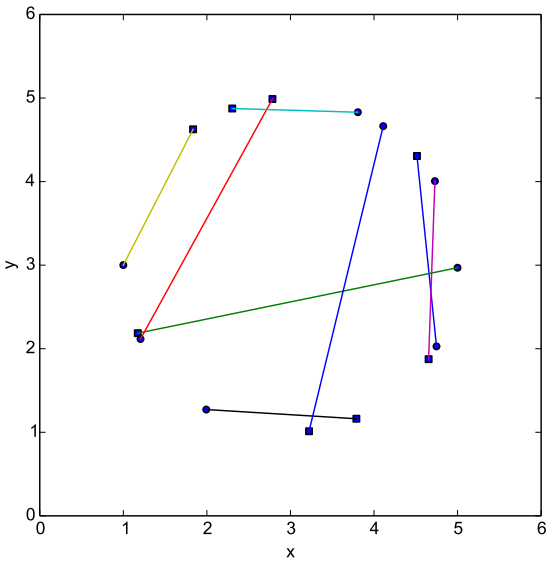

Figura 15: Solución óptima para un conjunto de 16 puntos dispuestos sobre una circunferencia de radio 2 centrada en $(3, 3)$.

3.2. Explicación de la solución

En esta sección daremos una explicación de por que 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.

En lo que sigue utilizaremos la siguiente terminología:

- Solución: Es una partición P de C que satisface ambas condiciones del enunciado.
- Candidato a solución: Son las particiones de C que satisfacen la segunda propiedad, aunque no necesariamente son de cardinal mínimo.
- Candidato a solución parcial: Es algún subconjunto de algún "candidato a solución". Es decir, es un conjunto de subconjuntos de C tal que estos son disjuntos dos a dos, pero donde la unión de los mismos no es C. Siempre es posible extenderlo a un candidato a solución.
- Sucesor: Dado un "candidato a solución parcial", X, un sucesor del mismo es otro conjunto, X', tal que X' es o bien "candidato a solución", o bien "candidato a solución parcial", que cumple $X \subset X'$ (notar que la inclusión debe ser estricta).

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, deberemos ver cuáles tienen cardinalidad mínima.

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 el candidato a solución parcial (o sea, obtenemos un sucesor). 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 el candidato a solución, 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.

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 $s+1$.

La diferencia entre la poda 1 y 2 puede resultar un tanto sútil pero será más clara cuando veamos la implementación de *backtracking*.

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 (al menos después de inicializarlo), 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 a 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.

Función `solve`: Es una función *wrapper*, que se encarga esencialmente de llamar a la función *backtracking*, pasándole el tablero inicial correspondiente al nivel 0 del árbol de posibilidades (en el que el candidato a solución parcial está vacío y tenemos todas las posibilidades disponibles). También se encarga de devolver la solución del problema.

Algorithm 5 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 de `backtracking`: Es la función más importante de la solución. Toma como parámetros un Tablero t y un entero *step*, que cumplen que el tamaño de $t.solucion$ es *step*.

En las líneas 2 y 3 del pseudocódigo estamos realizando la primer poda: si el tamaño de $t.solucion$ (candidata a solución parcial) es mayor o igual que el de la mejor candidata a solución encontrada hasta el momento, entonces no continuamos analizando esta rama del árbol. Caso contrario, verificamos si ya tenemos completa la partición correspondiente. De serlo, por la poda anterior, estamos seguros de que es mejor que cualquier cosa que hubiéramos encontrado antes y por lo tanto actualizamos nuestras variables globales.

Si $t.solucion$ todavía no está completa, entonces queremos hallar todos los sucesores de t[2]. Notar que tenemos dos sucesores por cada par de puntos disponibles (como se trata con semirrectas el orden si importa), pues únicamente consideramos semirrectas que pasen por al menos dos puntos. Si solo quedase un punto entonces simplemente agregamos el conjunto que lo contiene a la solución parcial y obtenemos un candidato a solución.

Para recorrer todas estas posibilidades hacemos lo siguiente por cada punto disponible i: si es el único, agregamos el conjunto que solo lo contiene a él, y llamamos a *backtracking* con el tablero actualizado y $step+1$; luego, retornamos. Si no era el único, entonces existe al menos un punto más con el que seguro se puede trazar una semirrecta. Por cada uno de estos puntos

[2]Si bien definimos sucesor para hablar específicamente de las particiones resulta práctico usarlo en un sentido más amplio para hablar del tablero entero.

j, buscamos que otros puntos disponibles pertenecen a la semirrecta definida por i y j con origen en i. A todos estos puntos (incluidos i y j) los metemos en un conjunto, y generamos un sucesor de t que resulta de a una copia del mismo agregarle este conjunto nuevo[3]. Finalmente, llamamos recursivamente a *backtracking* con el sucesor y $step + 1$.

El procedimiento que realizamos para determinar si un punto k pertenece a la semirrecta con origen en i que pasa por j es, por un lado ver si pertenece a la recta que pasa por i y j, y luego ver si k y j están en el mismo cuadrante respecto de i (un poco más adelante profundizamos esto útlimo). Notar que si i y j están alineados vertical (mismo x) u horizontalmente (mismo y), no se puede aplicar la fórmula clásica para determinar si un tercer punto pertenece a una recta dada por dos: $\frac{x_k - x_i}{x_j - x_i} = \frac{y_k - y_i}{y_j - y_i}$. Por lo tanto esos dos casos los analizamos a parte (el código usado para eso es muy simple de entender).

Notemos que las líneas 25 y 26 aportan la segunda poda mencionada en la subsección anterior. La idea es que si al volver del llamado recursivo $mejor = s + 1$, entonces no tiene sentido continuar ejecutando los ciclos pues todos los sucesores de t van a ser de tamaño $s + 1$ y por lo tanto no pueden mejorar lo que ya encontré. Podría parecer que esto se pisa con la condición de la línea 2 pero en rigor lo que nos ahorramos con la 25 es el costo de construir sucesores destinados a fracasar, cosa que no lográbamos de otra manera. Esto es particularmente útil en el mejor caso del algoritmo como veremos luego.

[3]Notar que es fundamental que sea una copia de t, pues del mismo queremos obtener todos los posibles sucesores y por lo tanto no podemos modificarlo directamente.

Algorithm 6 Pseudocódigo del procedimiento de `backtracking` en Kamehameha

```
1:  procedure BACKTRACKING(Tablero t, int step)
2:      if step ≥ mejor then                                              ▷ O(1)
3:          return
4:      if t.Solucionado() then                                           ▷ O(n)
5:          mejor ← step                                                  ▷ O(1)
6:          mejor_sol ← t.Solucion()                                      ▷ O(n)
7:      else
8:          for i ∈ [0, ..., n), t.EstaVivo?(i) do                        ▷ n veces
9:              if t.SoloQuedaUno?() then                                 ▷ O(1)
10:                 t.Matar(i)                                            ▷ O(n)
11:                 backtracking(t, step + 1)
12:                 return
13:             for j ∈ [0, ..., n), j ≠ i ∧ t.EstaVivo?(j) do            ▷ n veces
14:                 Tablero sucesor ← Copiar(t)                           ▷ O(2n + 1)
15:                 Vector(int) derrotados ← [i, j]                       ▷ O(1)
16:                 for k ∈ [0, ..., n), k ≠ i ∧ k ≠ j ∧ t.EstaVivo?(k) do ▷ n veces
17:                     if x_i ≠ x_j ∧ y_i ≠ y_j then                     ▷ O(1)
18:                         if (x_k − x_i)/(x_j − x_i) = (y_k − y_i)/(y_j − y_i) ∧ MismoCuadrante?(i, j, k) then ▷ O(1)
19:                             derrotados.push_back(k)                   ▷ O(1) [4]
20:                     else
21:                         if (AlinHor?(i, j, k) ∨ AlinVer?(i, j, k)) ∧ MismoCuadrante?(i, j, k) then ▷ O(1)
22:                             derrotados.push_back(k)                   ▷ O(1)
23:                 sucesor.Matar(derrotados)                             ▷ O(n)
24:                 backtracking(sucesor, step + 1)
25:                 if mejor = s + 1 then                                 ▷ O(1)
26:                     return
```

Función `mismoCuadrante`: Su código en si no es particularmente interesante, lo importante es entender la idea. El punto i es el origen de la semirrecta que estamos analizando. En particular, podemos pensar que este punto genera cuatro cuadrantes respecto de él (del mismo modo que lo hace el origen de coordenadas). Por lo tanto, lo que hace esta función es chequear si los puntos j y k están en el mismo cuadrante respecto de i. Si no lo estuvieran, entonces por más que estén alineados no pueden pertenecer a la misma semirrecta con origen en i.

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

[4] En general, hacer push_back a un vector tiene peor caso n. Sin embargo en el código usamos la función reserve (http://www.cplusplus.com/reference/vector/vector/reserve/) que permite que la operación sea constante siempre que no agreguemos más de n elementos, cosa que obviamente no puede ocurrir.

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`, $O(n + T(n)) = O(max\{n, T(n)\})$. Veamos entonces cuál es la cota superior de complejidad en peor caso de `backtracking`.

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

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

Decidimos omitir las constantes que suman y multiplican en el caso $n \geq 2$ pues solo harán más engorrosas las cuentas en la demostración y no modifican la complejidad.

A continuación damos una fórmula cerrada para esta ecuación de recurrecia cuando n es par[5]:

Lema 3.1:

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

Una demostración de este lema puede encontrarse en el apéndice.

[5]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.

Ahora acotemos $F(n)$, con $n \geq 2$:

$$\begin{aligned} F(n) &= \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i) \prod_{j=0}^{i} (n-2j)^2 \right) + \prod_{i=0}^{\frac{n}{2}-1} (n-2i)^2 \\ &\leq \sum_{i=0}^{\frac{n}{2}-1} \left(n \prod_{j=0}^{i} n^2 \right) + \prod_{i=0}^{\frac{n}{2}-1} n^2 \\ &= \sum_{i=0}^{\frac{n}{2}-1} (n^{2i+1}) + n^{2\times(\frac{n}{2}-1)} \\ &= \sum_{i=0}^{\frac{n}{2}-2} (n^{2i+1}) + n^{n-1} + n^{n-2} \\ &\leq \sum_{i=0}^{\frac{n}{2}-2} (n^{n-1}) + n^{n-1} + n^{n-1} \\ &\leq \frac{n}{2} n^{n-1} = \frac{1}{2} n^n \end{aligned} \tag{5}$$

Como $n \geq 2$, esto último está bien definido (solo se invalida en 0).

Luego, por definición de O, $F(n) = T(n)$ es $O(n^n)$. Como $n^n \leq n^{n+2} (\forall n \in \mathbb{N})$, entonces $O(n^n) \subset O(n^{n+2})$, y $T \in O(n^{n+2})$. Dado que la complejidad total del algoritmo era $O(max\{n, T(n)\})$, entonces la misma resulta $O(n^{n+2})$, que era la complejidad pedida.

3.3.2. Complejidad en mejor caso

El mejor caso es indudablemente el caso en que todos los puntos están alineados y el primer punto del vector *puntos* es uno de los puntos extremos, es decir que basta con una semirrecta con origen en ese punto para atravesar todos. En tal situación, solo se entrará una vez a los ciclos de las líneas 8 y 13 (esto es debido a la segunda poda), y n al de la 16.

Si hacemos el seguimiento línea por línea en el pseudocódigo, vemos que la complejidad total será $O(1 + n + 2 + 1 + (2n + 1) + 1 + 3 \times n + n + 1 + n) = O(n)$, por álgebra de órdenes.

Notar que si la línea 25 no estuviera, pese a haber encontrado obviamente la mejor solución posible, se seguiría iterando sobre todos los sucesores, con lo que la complejidad terminaría siendo $O(n^3)$.

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.

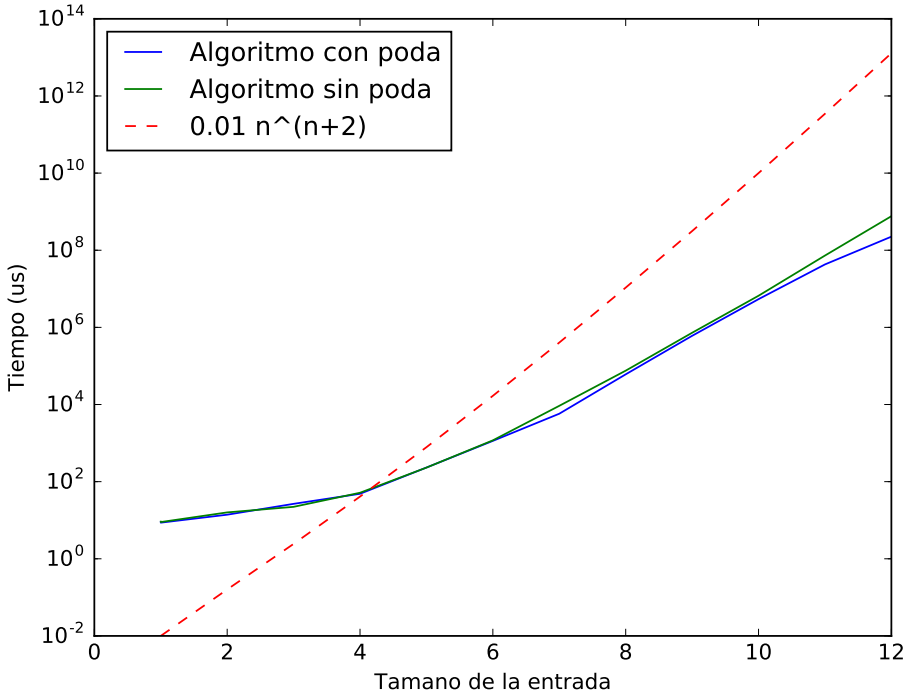

Figura 16: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Como puede verse, el tiempo requerido por el algoritmo crece muy aceleradamente, razón por la cual debimos analizarlo hasta $n = 12$; en tal situación, 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.

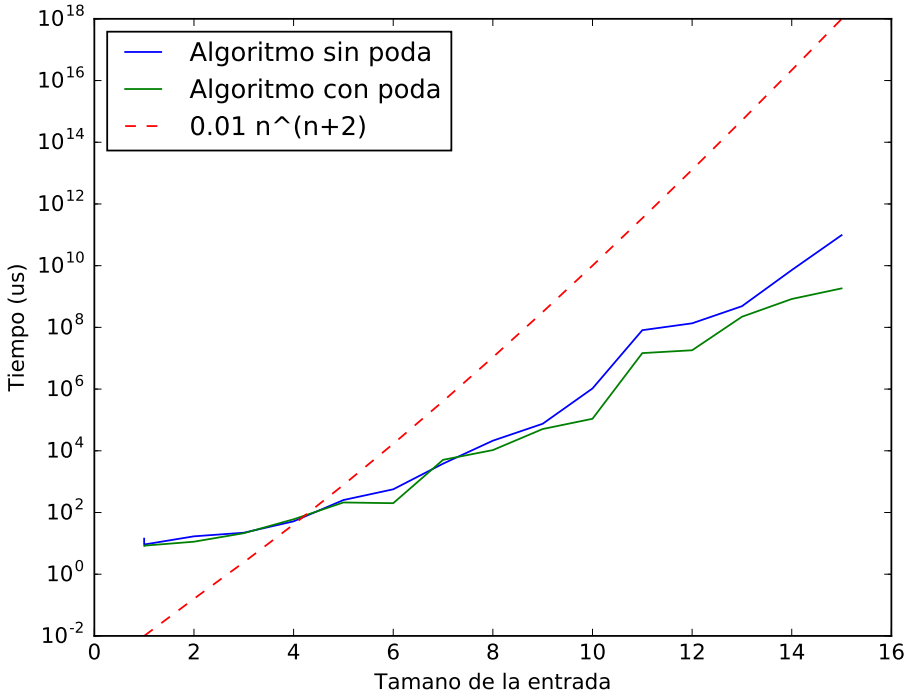

Figura 17: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

Como puede verse en la figura 17, el caso promedio es muy similar al peor caso. Esto se debe, a que lo más 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.

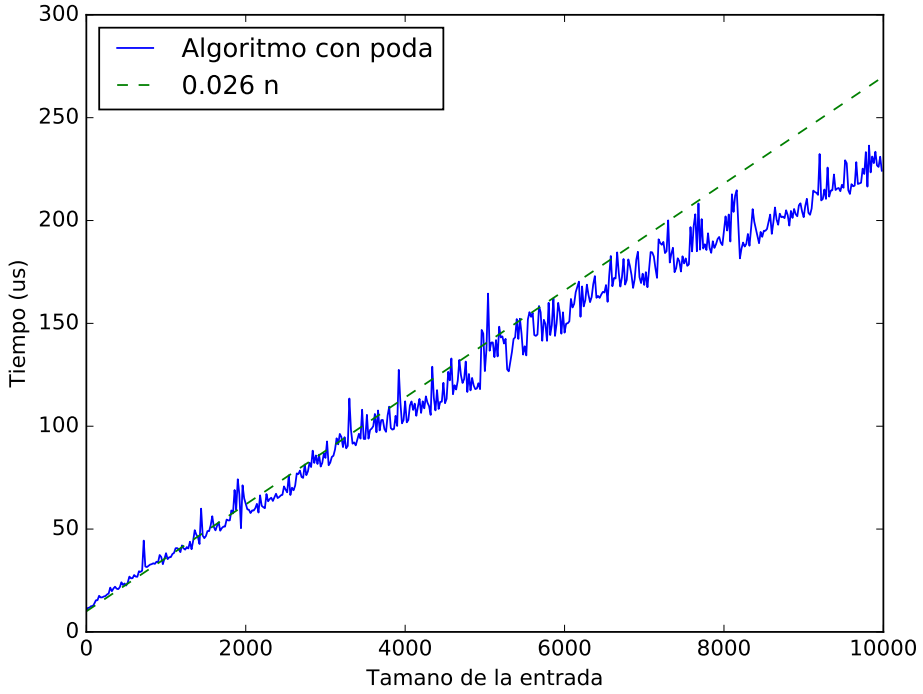

Figura 18: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

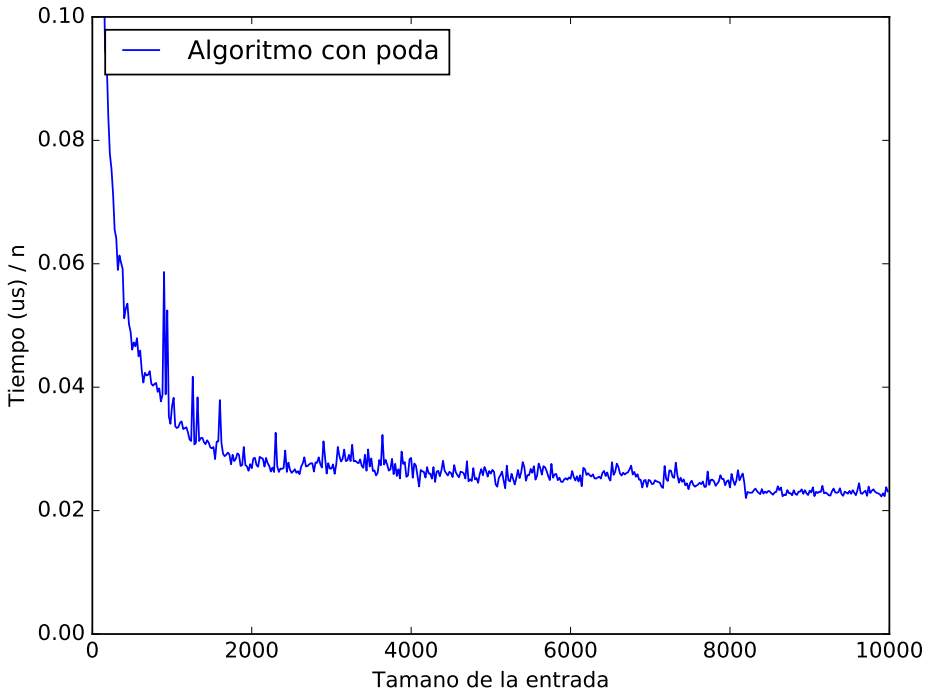

Figura 19: Tiempo que toma el algoritmo en μs para una entrada de tamaño n.

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, tomamos 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 usábamos 1 como seed, para la instancia 2, usábamos 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, tomábamos 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 $T(n+2) = F(n+2)$

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

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:

$$\begin{aligned} (n+2)^2 \prod_{j=0}^{i} (n-2j)^2 &= (n+2)^2 \prod_{t=1}^{i+1} (n-2(t-1))^2 \\ &= (n+2)^2 \prod_{t=1}^{i+1} (n+2-2t))^2 \\ &= \prod_{t=0}^{i+1} (n+2-2t)^2 \end{aligned} \tag{7}$$

$$\begin{aligned} (n+2)^3 + \sum_{i=0}^{\frac{n}{2}-1} \left((n-2i) \prod_{t=0}^{i+1} (\ldots) \right) &= (n+2)^3 + \sum_{r=1}^{\frac{n}{2}} \left((n-2(r-1)) \prod_{t=0}^{r} (\ldots) \right) \\ &= (n+2)^3 + \sum_{r=1}^{\frac{n}{2}} \left((n+2-2r)) \prod_{t=0}^{r} (\ldots) \right) \\ &= \sum_{r=0}^{\frac{n}{2}} \left((n+2-2r)) \prod_{t=0}^{r} (\ldots) \right) \end{aligned} \tag{8}$$

Notar que la igualdad 7 puede instanciarse para $i = \frac{n}{2} - 1$. Aplicando las igualdades 7 (dos veces) y 8 sobre la ecuación 6, llegamos a que $T(n+2) = F(n+2)$. $\square$

4.2. Kamehameha: el caso promedio se parece al peor caso

Formalicemos que queremos decir con esto. Supongamos que los puntos de nuestro *input* provienen, al azar, del conjunto $\mathcal{X}_M = \{1, \cdots ., M\} \times \{1, \cdots, M\}$. Lo que probaremos aquí es que si M es grande, entonces la probabilidad de que necesitemos $\frac{n}{2}$ rectas para cubrir todos los puntos (peor caso) es alta.

Primero, calculemos la cantidad de formas de elegir n puntos de ese conjunto:

$$|\{\text{formas de elegir } n \text{ puntos de } \mathcal{X}_M\}| = \binom{M^2}{n} = \frac{M^2(M^2-1)(M^2-2)(M^2-3)\dots(M^2-n+1)}{n!}$$

Por otro lado, contemos las configuraciones en las que necesitamos menos de $\frac{n}{2}$ semirrectas para cubrir todos los puntos. Estas configuraciones obviamente estan incluidas (de hecho son iguales) a las configuraciones en las que hay (al menos) 3 puntos alineados.

$$|\{\text{formas de elegir } n \text{ puntos de } \mathcal{X}_M \text{ tal que haya 3 alineados}\}| \leq \frac{M^2(M^2-1)n(M^2-3)\dots(M^2-n+1)}{n!}$$

Esta fórmula vale porque $M^2(M^2-1)$ son las formas de elegir los primeros 2 puntos. Luego tengo, como máximo, n puntos que puedo elegir sobre la semirrecta formada por los primeros 2 puntos. Finalmente, elijo los $n-3$ puntos restantes como quiera. Divido por $n!$ para eliminar las permutaciones.

Ahora, calculemos la probabilidad de que, dada una configuración al azar, esta requiera menos de $\frac{n}{2}$ semirrectas para cubrir todos los puntos. Llamaremos a esta probabilidad p.

$$p \leq \frac{|\{\text{formas de elegir } n \text{ puntos de } \mathcal{X}_M \text{ tal que haya 3 alineados}\}|}{|\{\text{formas de elegir } n \text{ puntos de } \mathcal{X}_M\}|} \leq \frac{n}{M^2-2}$$

Entonces, si tomamos los puntos sobre $\mathbb{N}^2$, como el problema indica y queremos calcular la probabilidad p, basta con hacer tender M a ∞, que se ve claramente que tiende a 0.

Más aún, en nuestros casos experimentales, elegimos $M = n^2$, por lo tanto queda que

$$p \leq \frac{n}{n^2-2} \to 0$$

Confirmando formalmente lo observado experimentalmente.