DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio Trabajo Práctico Nº1

29 de abril de 2016 — Algoritmos y Estructuras de Datos III

Grupo:

Integrante	LU	Correo electrónico
Labate, Martín	181/15	martinlabate@gmail.com
Pesaresi, Natalia	636/14	natalia.pesaresi@gmail.com
Tilve Neyra, Dana Marlene	614/14	tilve.dana@gmail.com
Taboh, Sebastián	185/13	sebi_282@hotmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Kaio Ken

1.1. Descripción del problema

El problema se trata de analizar la cantidad de peleas que debe haber entre un grupo de guerreros, si en cada pelea hay dos bandos a los que pueden pertenecer, para que todos peleen con todos y en cada pelea estén en uno de los dos bandos. Matemáticamente lo que proponemos para resolver en problema es: un conjunto, que se divide en dos una cantidad finita de veces tal que todos los elementos hallan estado en el conjunto opuesto a todos los demás elementos en elgun momento.

Observamos que la cantidad de peleas es logarítmica respecto a la cantidad de guerreros o elementos del conjunto principal: si n es potencia de 2 la cantidad de peleas es $\log(n)$ y si n no es potencia de 2 es $\lceil \log_2 n \rceil$. Esto es porque en la primer pelea, si establecemos que $\frac{n}{2}$ elementos pertenecen al bando 1 y $\frac{n}{2}$ elementos pertenecen al bando 2, ya tenemos que cada elemento peleó con la mitad del conjunto original. Podemos observar que resta que cada elemento pelee contra sus compañeros de equipo actual. Por lo que, repetimos el paso anterior para cada conjunto de $\frac{n}{2}$ elementos. Y así en $\log_2 n$ peleas cada guerrero habrá peleado con todos los demás, porque el conjunto de n elementos se "divide" en dos cada vez; decimos "divide" porque los equipos son dos siempre: bando 1 y bando 2, pero uso la expresión para explicar cómo obtenemos el resultado de que todos peleen con todos.

Debemos notar que si la cantidad de elementos es potencia de 2, la cantidad de peleas es exactamente $\log_2 n$. Pero si n no es potencia de 2, voy a necesitar una pelea más para que todos los elementos completen las peleas, es decir que todos terminen de pelear. Esto sucede porque la cantidad de elementos se encuentra entre dos potencias de 2 (entre $2^{\lfloor \log_2 n \rfloor}$ y $2^{\lceil \log_2 n \rceil}$), es decir que con $\lfloor \log_2 n \rfloor$ no alcanza y $\lceil \log_2 n \rceil$ es el menor número que seguro logra que todos peleen contra todos.

1.2. Pseudocódigo

Algorithm 1 distribuirGuerreros

```
function DistribuirGuerreros(filas: int, columnas: int, matriz: Arreglo < Arreglo < int >> )
    int i ← 0                                                          ▷ O(1)
    int j                                                              ▷ O(1)
    int cambio ← 0                                                     ▷ O(1)
    int equipo ← 1                                                     ▷ O(1)
    int x ← 0                                                          ▷ O(1)
    while i < filas do                                                 ▷ O(filas)
        cambio ← pow(2, i)                                             ▷ O(1)
        x ← 0                                                          ▷ O(1)
        j ← 0                                                          ▷ O(1)
        while j < columnas do                                          ▷ O(columnas)
            if x < cambio then                                         ▷ O(1)
                matriz[i][j] ← equipo                                  ▷ O(1)
            else
                equipo ← ((equipo + 1)mod2)                            ▷ O(1)
                matriz[i][j] ← equipo                                  ▷ O(1)
                x ← 0                                                  ▷ O(1)
            end if
            j + +                                                      ▷ O(1)
            x + +                                                      ▷ O(1)
        end while
        i + +                                                          ▷ O(1)
    end while
end function
```

Complejidad: $\mathcal{O}(filas * columnas)$

Algorithm 2 KaioKen

```
function KAIOKEN(n: int)
    int filas ← ⌈log2 n⌉                                   ▷ O(1)
    int m[filas][n]                                        ▷ O(1)
    distribuirGuerreros(filas, n, m)                       ▷ O(filas * n) = O(⌈log2 n⌉ * n)

    int i ← 0                                              ▷ O(1)
    int j                                                  ▷ O(1)

    imprimir filas                                         ▷ O(1)
    while i < filas do                                     ▷ O(filas) = O(⌈log2 n⌉)
        j ← 0                                              ▷ O(1)
        while j < n do                                     ▷ O(n)
            imprimir m[i][j] + 1                           ▷ O(1)
            j++                                            ▷ O(1)
            if j < n then                                  ▷ O(1)
                imprimir - "                               ▷ O(1)
            end if
        end while
        i++                                                ▷ O(1)
    end while
end function
```

Complejidad: $\mathcal{O}(\log_2 n * n)$
Justificación: $\mathcal{O}(\lceil \log_2 n \rceil * n) + \lceil \log_2 n \rceil * \mathcal{O}(n) = 2 * \mathcal{O}(\log_2 n * n) = \mathcal{O}(\log_2 n * n)$

1.3. Demostración de optimalidad

Se parte de un grupo de n guerreros y se divide en dos grupos de $n/2$ (si n es par; si no, se divide en un grupo de $\lceil n/2 \rceil$ y en otro de $\lfloor n/2 \rfloor$). A su vez, cada uno de estos se partirá a la mitad. Esto se debe a que para pelear a los n guerreros todos con todos se puede organizar una pelea de $n/2$ contra $n/2$ y luego pelear a cada uno de esos conjuntos todos con todos.

Así, se consigue un árbol exactamente binario porque siempre los grupos se dividen en 2 equipos. Las divisiones en equipos terminan cuando quedan n grupos de 1 guerrero porque en ese caso ya habrán peleado todos con todos los n guerreros, es decir, cuando queda un árbol binario con n hojas. Por otro lado, la altura de este árbol es la cantidad de peleas globales que se necesitó organizar, y sabemos que la altura del árbol es mínima cuando el árbol es balanceado, que de hecho es una propiedad que cumple el árbol resultante de dividir siempre a la mitad. Así, se concluye que esta forma de división resulta en el óptimo de peleas globales, que es igual a $\lceil \log_2 n \rceil$.

Esta es la misma explicación de por qué en mergeSort se divide el arreglo a la mitad.

1.4. Cota de complejidad

Como está detallado en el Algoritmo 3, la complejidad del algoritmo propuesto para KaioKen resulta $\mathcal{O}(\log_2 n * n)$ y de hecho también es $\theta(\log_2 n * n)$.

Este algoritmo no tiene ni mejor ni peor caso dado que siempre se realiza la misma cantidad de operaciones en función de n. Cabe resaltar que en este problema para un determinado tamaño de entrada no se pueden tener diferentes situaciones y el programa siempre lleva a cabo cada instrucción una cantidad fija de veces que depende del n.

Como se verá más adelante, esto no es así en el Problema 2, en el que puede haber un mismo tamaño de entrada con situaciones muy distintas y disposiciones de androides totalmente diferentes, dando lugar a un mejor y un peor caso del algoritmo propuesto para resolver ese problema. Como en ese caso la información puede ser distinta para un mismo tamaño de entrada cada instrucción se correrá una cantidad de veces que depende de esa información, dando lugar a mayores y menores tiempos de ejecución para el mismo tamaño de entrada.

1.5. Experimentación

Realizamos la medición del tiempo de ejecución de la parte del código que calcula la solución y recolectamos los datos expuestos en la siguiente tabla:

n (Tamaño de la entrada)	Tiempo promedio (nanosegundos)
10000	1689
15000	1889
20000	3051
25000	3616
30000	3794
35000	5612
40000	5735
45000	6550
50000	5798
55000	9041
60000	8217
65000	9236
70000	8781
75000	8025
80000	8841
85000	12973
90000	16051
95000	17312
100000	18256

Cuadro 1: Tabla que muestra los tiempos de ejecución correspondientes a distintos tamaños de entrada.

El gráfico asociado a este experimento es:

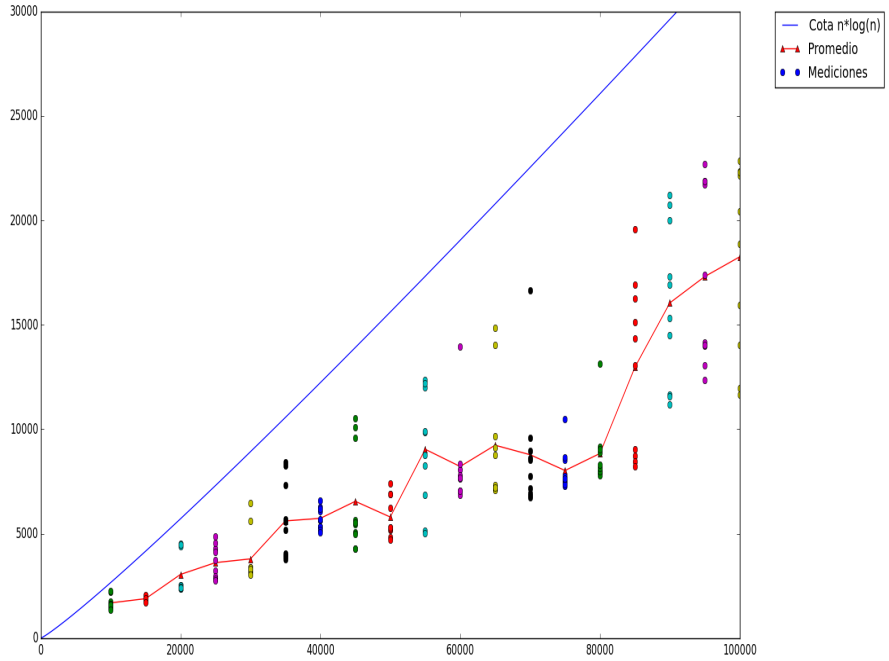

Figura 1: Tiempos de ejecución para distintas entradas. La curva azul es la curva de la complejidad, $n \times \log n$, multiplicada por 0.02, y la roja caso aleatorio.

1.6. Apéndice: código de KaioKen

```
#include <math.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

void kaioKen(int);

int main(int argc, char* argv[]){
   int i;
   cin >> i;
   kaioKen(i);
   return 0;
}

void kaioKen(int n){
   int filas = ceil(log2(n));
   int equipo = 1;
   int cambio = 0;
   int x = 0;
   int m [filas][n];

   //Distribuimos los guerreros
   for (int i = 0; i < filas; i++){
      cambio = pow(2,i);
      x = 0;
      for(int j = 0; j < n; j++){
         if(x < cambio){
            m[i][j] = equipo;
         }else{
            equipo = ((equipo +1) % 2);
            m[i][j] = equipo;
            x = 0;
         }
         x++;
      }
   }

   cout << filas << endl;
   for(int i = 0; i < filas; i++){
      for(int j = 0; j < n; j++){
         cout << m[i][j]+1;
         if (j < n) {
            std::cout << " ";
         }
      }
      cout << endl;
   }
}
```

2. Genkidama

2.1. Descripción del problema

El problema a resolver es el siguiente: Tenemos androides ubicados en una secuencia de puntos (x, y) en el espacio para los cuales si la coordenada x de un punto es mayor a la de otro punto entonces la coordenada y del segundo punto es menor. Dado un valor T, se desea calcular la mínima cantidad de genkidamas a tirar para destruir a todos los androides. Si se tira a un punto (x_0, y_0) la genkidama alcanza a destruir a todos los puntos que posean $x \leq x_0 + T$ y posean $y \leq y_0 + T$.

2.2. Ideas para desarrollar el algoritmo

Definamos el área de impacto como

$$\mathrm{A} : {N_0}^2 \times N \to Z^2$$
$$\mathrm{A}(P, T) = \{(x, y) \in Z^2 : 0 \leq x \leq P_x + T \ \wedge \ 0 \leq y \leq P_y + T\}$$

es decir, el conjunto de puntos del plano en los que en caso de haber androides estos morirían por la Gendikama tirada en P. Con esta idea de área de impacto se puede pensar que no aporta tener en cuenta el área por sobre el objetivo vivo de mayor coordenada en y.

Supongamos que tenemos un conjunto de androides situados en los puntos A, B, C, ... con $A_y > B_y > C_y > ...$ y $A_x < B_x < C_x < ....$ Va a haber que eliminar al objetivo situado en A. Se puede tirar una Gendikama al punto A, pero surge la pregunta de si se podría hacer algo mejor. ¿Qué pasa si se dispara a B? Si $A_y \leq B_y + T$ entonces el área de impacto A(B,T) incluye al punto A. Así, sería mejor disparar la Gendikama a B que a A dado que como $A_x < B_x$ entonces $A_x + T < B_x + T$, lo que nos da una idea intuitiva de inclusión de áreas, $A(A, T) \subset A(B, T)$ (nótese que la inclusión es estricta) y podemos decir que diparar a B es más destructivo.

Así, volvemos a preguntarnos: ¿Qué pasa si se dispara a C? ¿Esto también mataría a A?

Supongamos que sí, seguimos preguntándonos por el siguiente punto de menor coordenada y, y podemos tener dos situaciones:

- Recorrer hasta el último punto y que una Gendikama lanzada ahí también elimine al androide situado en A, necesitando sólo un disparo para destruir a todos.
- Recorrer hasta eventualmente encontrar un androide posicionado en un punto K al que si se lanzara una Genkidama ésta no matara al androide en A. En este caso habría que disparar al androide situado en la posición con la coordenada y inmediatamente anterior a la de K, llamémoslo J, y calcular $J_x + T$ para ver cuál sería el objetivo de mayor coordenada y que quedara vivo y repetir el proceso como si este fuera el nuevo A. Si no quedara ningún objetivo por eliminar entonces ya no habría necesidad de seguir disparando Genkidamas.

Este algoritmo es un algoritmo goloso: se elige al androide vivo de mayor coordenada y, llamémoslo A, y para este se elige al androide ubicado en el punto de menor coordenada y al que al tirarle una genkidama, ésta destruya a A.

2.3. Ejemplos

Supongamos que tenemos los puntos mostrados en la Figura 1 y $T = 1$.

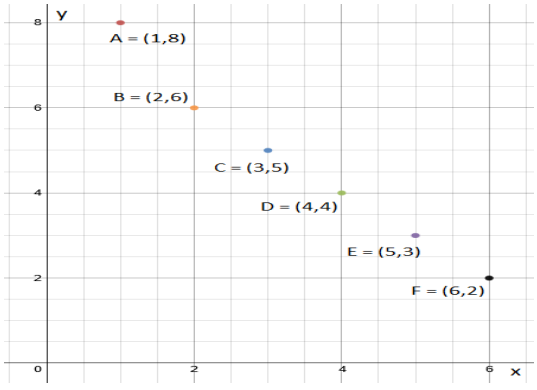

Figura 2: Puntos distribuidos inicialmente.

Primero queremos destruir al objetivo situado en A. Miramos el objetivo B, que tiene la mayor coordenada y de los que tienen menor coordenada y que A (que son todos dado que se empieza queriendo destruir al objetivo situado en el punto con mayor y). En la Figura 2 vemos que $A_y > B_y + T$ y que A no pertenece al área de impacto A(B, 1), o sea que disparar una genkidama al punto B no eliminaría al androide de A. Por transitividad, como $B_y > C_y > ... > F_y$, ninguno de los disparos a los puntos siguientes a B eliminará al objetivo de A, por lo que es necesario disparar una genkidama a A. El área de impacto A(A, T) = A((1,8), 1) de esta genkidama, sombrada con rojo en la Figura 2, incluye al punto B, que además es el de mayor coordenada x de los destruidos por esta gendikama.

El siguiente a B, C, es el de mayor coordenada y de los que están vivos, y se repite el proceso que se llevó a cabo con A. En la Figura 3 se ve que la segunda gendikama se disparará a D, destruyendo a los objetivos de C, D y E. Finalmente queda la situación mostrada en la Figura 4 y se dispara una tercera genkidama a F.

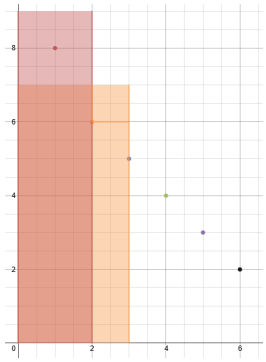

Figura 3: Área de impacto de A en rojo y de B en naranja.

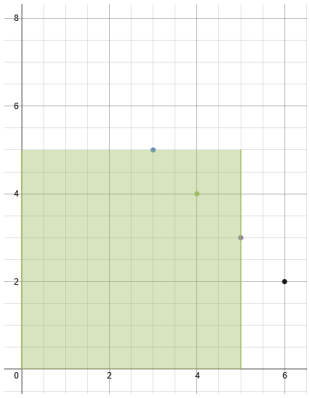

Figura 4: Área de impacto de D en verde.

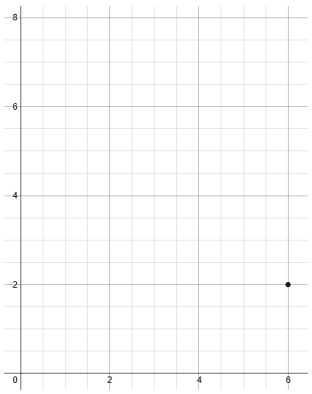

Figura 5: Androide restante por destruir, en F.

Ahora veamos cómo sería este proceso con $T = 2$. De vuelta se comienza queriendo destruir al objetivo en A. Como se ve en la Figura 5, disparar a C no destruye a A pero disparar a B sí. Así, morirán los androides situados en los puntos incluidos en la región sombreada naranja en la Figura 6 (el área de impacto A(B, 2)).

Finalmente queda la situación mostrada en la Figura 7 en la que el objetivo es destruir al objetivo de E y se terminará disparando a F eliminando a los dos androides restantes.

En este caso se habrá destruido a todos los androides utilizando dos genkidamas.

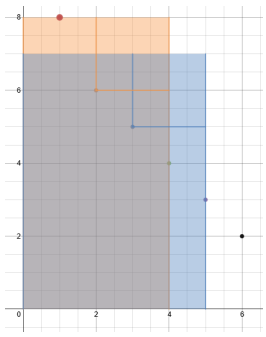

Figura 6: Área de impacto de B en naranja y de C en azul.

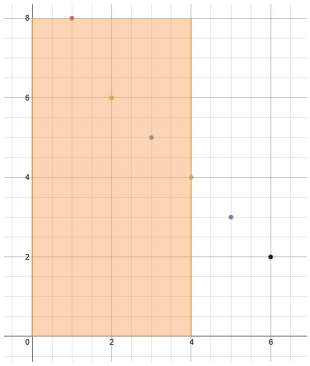

Figura 7: Área de impacto de B en naranja.

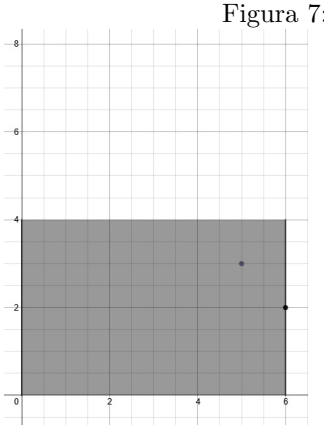

Figura 8: Área de impacto de F en negro.

2.4. Pseudocódigo

Algorithm 3 IndiceDeMenorYQueLoMata

function IndiceDeMenorYQueLoMata(t: int, indiceDeObjetivo: int, e: $Arreglo < Tupla < int, int >>$) $\rightarrow j$: `int`

- $int\ j \leftarrow indiceDeObjetivo - 1$ ▷ $\mathcal{O}(1)$
- **while** $(j \geq 0) \wedge (((e[j]).y + t) \geq (e[indiceDeObjetivo]).y)$ **do** ▷ $\mathcal{O}(indiceDeObjetivo)$
  - j-- ▷ $\mathcal{O}(1)$
- **end while**
- j++ ▷ $\mathcal{O}(1)$
- return j ▷ $\mathcal{O}(1)$

end function

Complejidad: $\mathcal{O}(indiceDeObjetivo)$

Algorithm 4 IndiceDeMayorXQueMata

function IndiceDeMayorXQueMata(t: int, indiceDeObjetivo: int, e: $Arreglo < Tupla < int, int >>$) $\rightarrow j$: `int`

- $int\ distanciaDeDano \leftarrow e[indiceDeObjetivo].x + t$ ▷ $\mathcal{O}(1)$
- $int\ i \leftarrow indiceDeObjetivo - 1$ ▷ $\mathcal{O}(1)$
- **while** $(i \geq 0) \wedge (((e[i]).x) \leq distanciaDeDano$ **do** ▷ $\mathcal{O}(indiceDeObjetivo)$
  - i-- ▷ $\mathcal{O}(1)$
- **end while**
- i++ ▷ $\mathcal{O}(1)$
- return i ▷ $\mathcal{O}(1)$

end function

Complejidad: $\mathcal{O}(indiceDeObjetivo)$

Algorithm 5 Genkidama

```
function GENKIDAMA(t: int, n: int, e: Arreglo < Tupla < int, int >>)
    int [ ] atacados                                                          ▷ O(1)
    int genkidamasUtilizadas ← 0                                              ▷ O(1)
    int indiceDeObjetivoPorArea ← n − 1                                       ▷ O(1)
    // el de arriba es aquel al que quiero que le llegue la onda expansiva
    bool hayAlgunoVivo ← true                                                 ▷ O(1)
    while hayAlgunoVivo do                                                    ▷ O(menorY)
        // el de abajo es aquel al que le voy a tirar la bomba
        int menorY ← indiceDeMenorYQueLoMata(t, indiceDeObjetivoPorArea, e)   ▷
O(indiceDeObjetivoPorArea)
        agregarAtras(atacados, indiceDeObjetivo + 1)                          ▷ O(1)
        // el + 1 de arriba surge de que en el enunciado se enumeran desde el 1
        y nosotros enumeramos desde 0
        int indiceDeMayorXMatado ← indiceDeMayorXQueMata(t, menorY, e)        ▷ O(menorY)
        hayAlgunoVivo ← ¬(indiceDeMayorXMatado == 0)                          ▷ O(1)
        indiceDeObjetivoPorArea ← indiceDeMayorXMatado − 1                    ▷ O(1)
        genkidamasUtilizadas++                                                ▷ O(1)
    end while
    imprimir genkidamasUtilizadas
    int h ← 0
    while h < genkidamasUtilizadas do
        imprimir atacados[h]
        h++
        if h < genkidamasUtilizadas then
            imprimir " "
        end if
    end while
end function
```

Complejidad: $\mathcal{O}(n)$

Justificación:

Teniendo en cuenta que

- *int menorY* ← $indiceDeMenorYQueLoMata(t, indiceDeObjetivoPorArea, e)$ es $\mathcal{O}(indiceDeObjetivoPorArea - menorY + 1)$
- *int indiceDeMayorXMatado* ← $indiceDeMayorXQueMata(t, indiceDeObjetivo, e)$ es $\mathcal{O}(indiceDeObjetivo - indiceDeMayorXMatado + 1)$
- y que si notamos
  $indiceDeObjetivoPorArea_i$ a indiceDeObjetivoPorArea al comienzo de la iteración i,
  $indiceDeMayorXMatado_i$ a indiceDeMayorXMatado luego de la asignación realizada en la iteración i,
  $menorY_i$ a menorY luego de la asignación realizada en la iteración i,
  entonces se tiene

$$indiceDeObjetivoPorArea_{i+1} = indiceDeMayorXMatado_i - 1 \tag{1}$$

el cálculo de la complejidad se simplifica.

Cada iteración i tiene complejidad

$$\begin{aligned}\mathcal{O}(indiceDeObjetivoPorArea_i - menorY_i + 1) + \mathcal{O}(menorY_i - indiceDeMayorXMatado_i + 1)\\ = \mathcal{O}(indiceDeObjetivoPorArea_i - indiceDeMayorXMatado_i + 2)\end{aligned} \tag{2}$$

Para la primera iteración se tiene

$$indiceDeObjetivoPorArea_1 = n - 1 \tag{3}$$

y para la *ú*ltima se tiene

$$indiceDeMayorXMatado_{\#iteraciones} = 0 \tag{4}$$

Así, la complejidad total del algoritmo es la suma sobre las iteraciones de la complejidad para cada iteración:

$$\begin{aligned}
&\sum_{i=1}^{\#iteraciones} \mathcal{O}(indicePorArea_i - MayorXMatado_i + 2) \\
&\qquad = \mathcal{O}(indicePorArea_1 - MayorXMatado_1 + 2) \\
&\qquad\quad + \sum_{i=2}^{\#iteraciones-1} \mathcal{O}(indicePorArea_i - MayorXMatado_i + 2) \\
&\qquad + \mathcal{O}(indicePorArea_{\#iteraciones} - MayorXMatado_{\#iteraciones} + 2) \\
&= \mathcal{O}(n-1) + \sum_{i=1}^{\#iteraciones-1} \mathcal{O}(-MayorXMatado_i + 2 + indicePorArea_{i+1}) + \mathcal{O}(0+2) \\
&= \mathcal{O}(n) + \sum_{i=1}^{\#iteraciones-1} \mathcal{O}(-MayorXMatado_i + 2 + (MayorXMatado_i - 1)) \\
&\qquad = \mathcal{O}(n) + \mathcal{O}(\#iteraciones - 1) = \mathcal{O}(n) + \mathcal{O}(n) \\
&\qquad\qquad = \mathcal{O}(n)
\end{aligned} \tag{5}$$

Como se puede ver en el pseudocódigo y se explica en la sección "Ideas para desarrollar el algoritmo" se recorren todos los puntos ordenados, de mayor a menor coordenada y, una vez, ya sea para encontrar a que enemigo atacar o para ver cuales murieron en el último ataque.
Cuando se busca un enemigo a atacar, se verifica que al atacarlo se maten todos los anteriores, así al lanzar una Genkidama sabemos que todos los enemigos por los que ya iteramos estarán muertos y solo queda ver cuales de los siguentes al enemigo atacado tambíen muerieron. Esto se logra iterando por los enemigos subsiguientes al atacado hasta encontrar el primero al que la Genkidama no mató y con él se vuelve a comenzar el proceso. Se puede ver claramente que se itera por todos los enemigos y que se hace solo una vez haciendo que la complejidad sea $\mathcal{O}(n)$ siendo n la cantidad de enemigos.

2.5. Análisis de complejidad

Como ya se ha explicado, el algoritmo Genkidama tiene complejidad temporal $\mathcal{O}(n)$, pero también podemos afirmar que tiene complejidad temporal $\theta(n)$ porque para cada androide chequea o bien si la genkidama tirada en esa ubicación mata a otro o bien si su ubicación es alcanzada por otra explosión. Esto da una cantidad de chequeos de al menos n pues hay n androides.

El mejor caso se tiene cuando todos los androides mueren con una sola genkidama utilizada pues luego de fijar un objetivo sólo se realiza un chequeo para cada uno de los $n-1$ androides.

El peor caso se tiene cuando para matar a cada androide se requiere una genkidama distinta y por tanto se hacen dos chequeos para destruir cada uno de los $n-1$ androides.

2.6. Experimentación

Para la experimentación medimos los tiempos de ejecución para distintos tamaños de entrada y lo hicimos varias veces para cada tamaño de entrada para tener resultados más confiables. Este proceso lo realizamos para las situaciones que creíamos que daban lugar a mejor y peor caso y verificamos que efectivamente eran las situaciones que pensábamos.

Además hicimos un generador de instancias aleatorias y medimos los tiempos de ejecución para muchas entradas generadas aleatoriamente y efectivamente eran tiempos para tamaños de entrada grandes solían estar comprendidos entre los tiempos de ejecución para mejor caso y para peor caso.

A continuación se exponen la tabla y el gráfico resultante de esas mediciones

n (Tamaño de la entrada)	Tiempo promedio (nanosegundos)
10000	1245
15000	1502
20000	2024
25000	2582
30000	3082
35000	3614
40000	4433
45000	5056
50000	5246
55000	5954
60000	6505
65000	6346
70000	6816
75000	7065
80000	7951
85000	8516
90000	9463
95000	10662
100000	11851

Cuadro 2: Tabla que muestra los tiempos de ejecución promedio correspondientes a distintos tamaños de entrada con T aleatorio entre cada rango de valores.

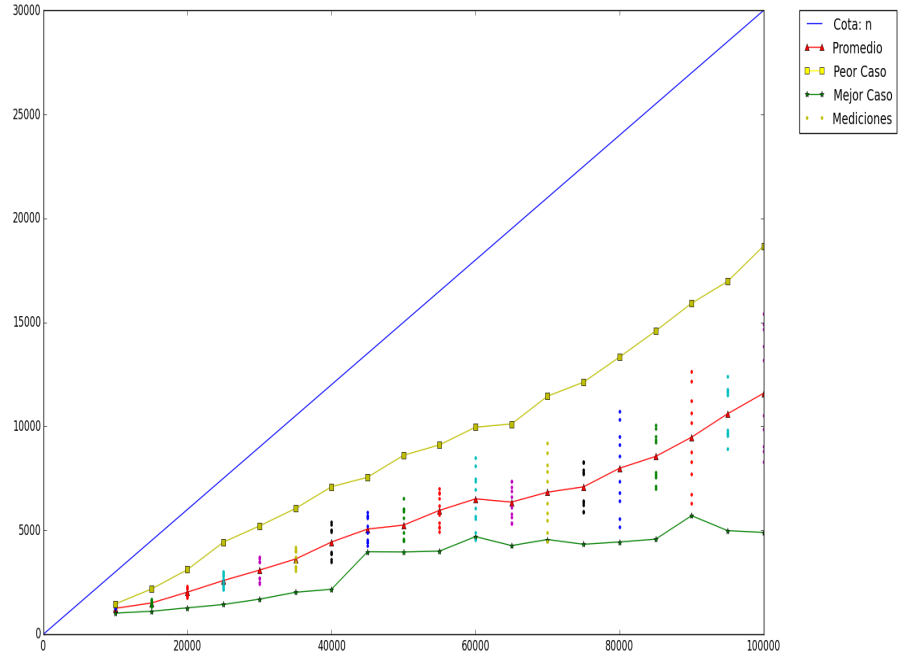

Figura 9: Tiempos de ejecución para distintas entradas. La curva azul es la curva de la complejidad, $0{,}3 * n$ (la constante se incluyó para lograr mostrar claramente que efectivamente el algoritmo es $\mathcal{O}(n)$), la línea verde es mejor caso, la roja caso aleatorio y la amarilla peor caso.

2.7. Apéndice: código de Genkidama

```
#include <tuple>
#include <vector>
#include <iostream>
#include <cassert>
using namespace std;

void genkidama(int, int, vector<tuple<int, int>>);
int indiceDeMenorYQueLoMata(int, int, vector<tuple<int,int>>);
int indiceDeMayorXQueMata(int, int, vector<tuple<int,int>>);

int main(){
   int t;
   int n;
   cin >> n >> t;
   vector<tuple<int,int>> e;
   tuple<int,int> en;
   int x;
   int y;
   for (int i = 0; i < n; i++) {
      cin >> x >> y;
      en = make_tuple(x,y);
      e.push_back((en));
   }
   genkidama(t, n, e);
   return 0;
}

// Toma el t, el indice del objetivo en e y el vector de tuplas.
int indiceDeMenorYQueLoMata(int t, int indiceDeObjetivo, vector<tuple<int,int>> e){
   int j = indiceDeObjetivo - 1;
   while((j >= 0) && ((get<1>(e[j]) + t) >= get<1>(e[indiceDeObjetivo]))){
      j--;
   }
   j++;
```

```
   return j;
}

int indiceDeMayorXQueMata(int t, int indiceDeObjetivo, vector<tuple<int,int>> e){
   int distanciaDeDano = get<0>(e[indiceDeObjetivo]) + t;
   int i = indiceDeObjetivo - 1;
   while((i >= 0) && (get<0>(e[i]) <= distanciaDeDano)){
      i--;
   }
   i++;
   return i;
}

void genkidama(int t, int n, vector<tuple<int,int>> e){
   assert (n > 0 && n == e.size() && "La cantidad de enemigos es distinta a la cantidad de
       posisciones");
   vector<int> atacados;
   int genkidamasUtilizadas = 0;
   int indiceDeObjetivoPorArea = n-1;
   // el de arriba es aquel al que quiero que le llegue la onda expansiva
   bool hayAlgunoVivo = true;
   while(hayAlgunoVivo){
      // el de abajo es aquel al que le voy a tirar la bomba
      int indiceDeObjetivo = indiceDeMenorYQueLoMata(t, indiceDeObjetivoPorArea, e);
      atacados.push_back(indiceDeObjetivo + 1);
      // el +1 de arriba surge de que en el enunciado se enumeran desde el 1 y
      // nosotros enumeramos desde 0
      hayAlgunoVivo = !(indiceDeMayorXQueMata(t, indiceDeObjetivo, e) == 0);
      indiceDeObjetivoPorArea = indiceDeMayorXQueMata(t, indiceDeObjetivo, e) - 1;
      genkidamasUtilizadas++;

   }
   std::cout << genkidamasUtilizadas << std::endl;
   int h = 0;
   while (h < genkidamasUtilizadas) {
         std::cout << atacados[h];
         h++;
         if (h < genkidamasUtilizadas) {
               std::cout << " ";
         }
   }
   std::cout << std::endl;
}
```

3. Kamehameha

3.1. Descripción del problema

El problema consiste en calcular la mínima cantidad de Kamehamehas que debe lanzar Goku para destruir un conjunto de andriodes usando Backtracking. Al usar backtracking, el problema se reduce a trazar todas las posibles rectas que pasan una sola vez por cada punto y elegir la combinación con menor cantidad de rectas. En el siguiente gráfico se muestra un ejemplo con 5 androides en las siguientes posiciones 1 (1,2) 2 (2,1) 3 (3,2) 4 (4,2) 5 (3,3)

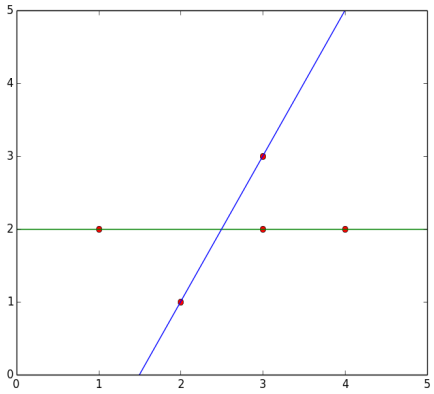

Figura 10: Solución para un caso con 5 androides.

Una posible solución óptima sería lanzar dos Kamehameha, uno que mate a 3 androides (al 1, 3 y 4), la recta verde, y otro que mate a los otros 2, la recta azul. En este caso la solución es única.

3.2. Descripción de la resolución

La idea del algoritmo es que recursivamente vaya creando todas las combinaciones de rectas. Esto lo logramos utilizando una función principal a la que llamamos recursivamente, Kamehameha, y dos funciones, una que se fija si un androide puede ser atacado con un Kamehameha ya lanzado (si está alineado con algún otro grupo de androides ya atacados) y vuelve a llamar a la función principal con los nuevos valores una vez por cada grupo con el que está alineado este androide; y otra función que comienza un nuevo ataque con ese androide y vuelve a llamar a la función principal.

De esta forma, probando con un enemigo, volviendo atrás, probando con el siguiente, volviendo atrás y así sucesivamente hasta probar con todos los enemigos, podemos generar todas las posibles combinaciones de líneas. Lo único que resta es verificar cuál de estas es la óptima. Para conseguirlo, cada vez que generamos una solución completa, verificamos su tamaño (su cantidad de líneas) y la guardamos si es la menor hasta el momento, si no la descartamos.

Para mejorar el tiempo de ejecución decidimos agregar una poda que descarta soluciones que están siendo generadas en el momento que su tamaño supera el tamaño de la solución mínima alamacenada, ya que sabemos que es mayor o igual a la que ya tenemos y por lo tanto no nos sirve.

3.3. Cota de complejidad

Como la cantidad de posibles combinaciones distitnas de n andoides con n androides distintos es análoga a la cantidad de números de n digitos distintos que se pueden armar con n números de un digito distintos, ya que el primer androide que tomemos, lo podemos dejar solo o juntarlo con cualquiera de los otros $n-1$ androides lo que le da n posibles opciones. Una vez que el primero esta fijo, hacemos lo mismo con el segundo androide, puede quedarse donde está o juntarse con cualquiera de los otros $n-2$ androides que aún no tienen posición fija, lo que le da $n-1$ posibilidades. Luego hacemos lo mismo con el resto de los androides, cada uno tendra la cantidad de posibilidades que tenía el androide anterior -1, hasta llegar al último androide que queda con 1 sola posibilidad, quedarse dodne está. Así se puede ver que la cantidad todal de combinasciones es $n*(n-1)*(n-2)*...*(n-(n+2))*(n-(n+1))$ que da $n!$.

Por lo tanto, la cantidad de hojas de nuestro árbol de recursión, es decir la cantidad de soluciones posibles, como no se llega a la misma solución más de una vez, está acotada por $n!$.

Por cómo planteamos la resolución del problema, por cada solución que generamos, volvemos atrás y reutilizamos lo ya calculado para generar todas las posibles. De esta forma las hojas de nuestro árbol tienen siempre al menos un hermano. Suponiendo el peor caso, donde cada hoja tiene solamente un hermano, como cada sub solución es una solución posible de un problema de menor tamaño, podemos asumir que todo nodo tiene solamente un hermano, convirtiendo a nuestro árbol de recursión en un árbol binario. Teniendo $n!$ hojas y siendo la cantidad total de nodos $= ((\#hojas) * 2 - 1)$ nos da que la cantidad de nodos es del orden de $2 * n!$ por lo tanto la complejidad del algoritmo termina siendo $\mathcal{O}(n!)$.

Teniendo en cuenta las podas el mejor caso ocurre cuando están todos los androides alineados ya que al comparar el tamaño de las soluciones se descartan todas las que tienen más que un Kamehameha. Por otro lado, el peor caso se da cuando no hay más de 2 androides para una misma recta debido a que las soluciones en su mayoría van a tener la misma cantidad de rectas.

3.4. Pseudocódigo

listaPos $= lista < (int, int) >$, kamehamehas $= lista < (listaPos) >$

Algorithm 6 Kamehameha

```
function KAMEHAMEHA(enemigos: listaPos, ataques: Kamehamehas, nroAtaque: int)
    if minimoGlobal ≤ size(ataques) then                                   ▷ O(1)
        return                                                             ▷ O(1)
    end if
    if size(enemigos) = 0) then                                            ▷ O(1)
        if minimoGlobal > size(ataques) then                               ▷ O(1)
            minimoGlobal ← size(ataques)                                   ▷ O(1)
            mejorConfiguracion ← ataques                                   ▷ O(1)
        end if
    else
        enemigo ← enemigos[0]                                              ▷ O(1)
        erase(enemigo, enemigos)                                           ▷ O(size(enemigos))
        AtacarEnAtacados(enemigo, enemigos, ataques, nroAtaques)           ▷ O(nroAtaques)
        AtacarEnNuevoAtaque(enemigo, enemigos, ataques, nroAtaques)        ▷ O(1)
        insert(enemigo, enemigos)                                          ▷ O(size(enemigos))
    end if
end function
```

Complejidad: $\mathcal{O}(n!)$

Algorithm 7 AtacarEnAtacados

function AtacarEnAtacados(*enemigo*: posición, *restoEnemigos*: listaPos, *ataques*: Kamehamehas, *nroAtaques*: int)
- $i \leftarrow 0$ ▷ $\mathcal{O}(1)$
- $atacados \leftarrow nuevaListaPos()$ ▷ $\mathcal{O}(1)$
- **while** $i \leq nroAtaques$ **do** ▷ $\mathcal{O}(nroAtaques)$
  - $atacados \leftarrow ataques[i]$ ▷ $\mathcal{O}(1)$
  - **if** size($atacados$) $= 0$ **then** ▷ $\mathcal{O}(1)$
    - $comenzarAtaque \leftarrow$nuevaListaPos() ▷ $\mathcal{O}(1)$
    - pushBack($enemigo, comenzarAtaque$) ▷ $\mathcal{O}(1)$
    - $ataque[i] \leftarrow comenzarAtaque$ ▷ $\mathcal{O}(1)$
    - Kamehameha($restoEnemigos$, $ataques$, $nroAtaques$) ▷ $\mathcal{O}(1)$
    - $restablecerAtaque \leftarrow$nuevaListaPos() ▷ $\mathcal{O}(1)$
    - $ataque[i] \leftarrow restablecerAtaque$ ▷ $\mathcal{O}(1)$
  - **else if** Alineados($atacados$, $enemigo$) **then** ▷ $\mathcal{O}(1)$
    - pushBack($enemigo$, $ataque[i]$) ▷ $\mathcal{O}(1)$
    - Kamehameha($restoEnemigos$, $ataques$, $nroAtaques$) ▷ $\mathcal{O}(1)$
    - popBack($ataque[i]$) ▷ $\mathcal{O}(1)$
  - **end if**
  - $i \leftarrow i + 1$ ▷ $\mathcal{O}(1)$
- **end while**

end function

Complejidad: $\mathcal{O}(nroAtaques)$

Algorithm 8 AtacarEnNuevoAtaque

function AtacarEnNuevoAtaque(*enemigo*: posicion, *restoEnemigos*: listaPos, *ataques*: Kamehamehas, *nroAtaque*: int)
- **if** size($ataque[nroAtaque]$) > 0 **then** ▷ $\mathcal{O}(1)$
  - $comenzarAtaque \leftarrow$nuevaListaPos() ▷ $\mathcal{O}(1)$
  - pushBack($enemigo, comenzarAtque$) ▷ $\mathcal{O}(1)$
  - pushBack($comenzarAtaque, ataques$) ▷ $\mathcal{O}(1)$
  - $nroAtaque \leftarrow nroAtaque + 1$ ▷ $\mathcal{O}(1)$
  - Kamehameha($restoEnemigos$, $ataques$, $nroAtaques$) ▷ $\mathcal{O}(1)$
  - $nroAtaque \leftarrow nroAtaque - 1$ ▷ $\mathcal{O}(1)$
  - popBack($ataque$) ▷ $\mathcal{O}(1)$
- **end if**

end function

Complejidad: $\mathcal{O}(1)$

Algorithm 9 Alineados

function Alineados(*atacados*: kamehameha, *enemigo*: posicion)Bool
- **if** size($atacados$) ≤ 1 **then** ▷ $\mathcal{O}(1)$
  - return $true$ ▷ $\mathcal{O}(1)$
- **else**
  - $primero \leftarrow atacados[0]$ ▷ $\mathcal{O}(1)$
  - $segundo \leftarrow atacados[1]$ ▷ $\mathcal{O}(1)$
  - $termino1 \leftarrow segundo[1] - primero[1]$ ▷ $\mathcal{O}(1)$
  - $termino2 \leftarrow enemigo[0] - primero[0]$ ▷ $\mathcal{O}(1)$
  - $termino3 \leftarrow enemigo[1] - primero[1]$ ▷ $\mathcal{O}(1)$
  - $termino4 \leftarrow segundo[0] - primero[0]$ ▷ $\mathcal{O}(1)$
  - return $termino1 * termino2 = termino3 * termino4$ ▷ $\mathcal{O}(1)$
- **end if**

end function

Complejidad: $\mathcal{O}(1)$

3.5. Experimentación

Experimentando con el algoritmo nos dimos cuenta de que el tiempo de ejecución variaba un poco dependiendo del momento particular en que se corriera el programa, de la distribución de los puntos o del funcionamiento de la máquina. Es por esto que decidimos ejecutarlo 5 veces para cada grupo de n puntos y tomar al promedio de los mismos como una referencia más confiable. Lo hicimos con $n = 1$ a 10.

A continuación se exponen la tabla y el gráfico resultante de esas mediciones

n (Tamaño de la entrada)	Tiempo (segundos)
1	$2{,}78 \times 10^{-5}$
2	$3{,}02 \times 10^{-5}$
3	$4{,}30 \times 10^{-5}$
4	$5{,}54 \times 10^{-5}$
5	$1{,}43 \times 10^{-4}$
6	$1{,}58 \times 10^{-4}$
7	$6{,}88 \times 10^{-4}$
8	$9{,}09 \times 10^{-4}$
9	$3{,}31 \times 10^{-3}$
10	$4{,}38 \times 10^{-3}$

Cuadro 3: Tabla que muestra los tiempos de ejecución correspondientes a distintos tamaños de entrada.

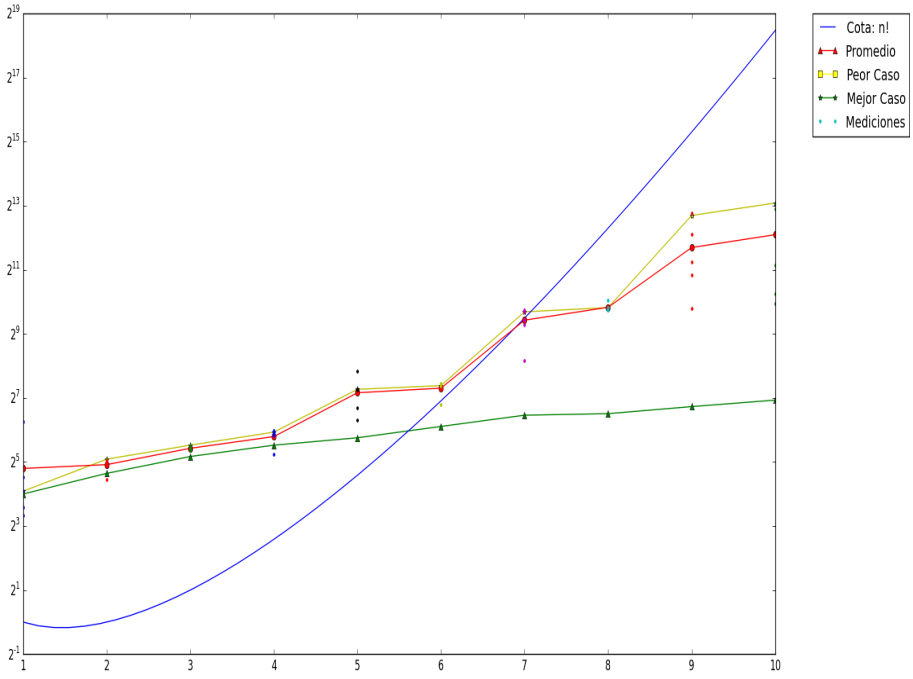

Figura 11: Tiempos de ejecución para distintas entradas. La curva azul es la curva de la complejidad, $n!$, la línea verde es mejor caso, la roja caso aleatorio y la amarilla peor caso.

3.6. Apéndice: código de Genkidama

```
#include <iostream>
#include <deque>
#include <utility>
#include <vector>
#include <stdlib.h>
#include <time.h>
#include <algorithm>
#include <limits>
using namespace std;

typedef pair<int,int> posicion_t;
typedef vector<posicion_t> Kamehameha_t;
typedef vector<Kamehameha_t> Kamehamehas_t;
typedef deque<posicion_t> listaPos_t;

int minimo_global = std::numeric_limits<int>::max();
listaPos_t enemigos_global;
Kamehamehas_t mejor_configuracion;

void Kamehameha(listaPos_t enemigos,
              Kamehamehas_t enLaMira,
              int indexRectaActual);
void atacarEnAtaqueActual(posicion_t enemigo,
                       listaPos_t restoEnemigos,
                       Kamehamehas_t ataques,
                       int nroAtaque);
void atacarEnNuevoAtaque(posicion_t enemigo,
                      listaPos_t restoEnemigos,
                      Kamehamehas_t ataques,
                      int nroAtaque);
bool alineados (Kamehameha_t atacados, posicion_t enemigo);
void reporte(listaPos_t enemigos, Kamehamehas_t ataques, int nroAtaque);
int buscarPosicion(const posicion_t& enemigo);
void mostrarSolucion();

int main() {
    int cantEnemigos;
    cin >> cantEnemigos;
    listaPos_t enemigos;
    Kamehamehas_t enLaMira;
    Kamehameha_t kamehameha;
    enLaMira.push_back(kamehameha);
    int indexRectaActual;
    posicion_t posicion;

    for (int i = 0; i < cantEnemigos; i++) {
        int x;
        int y;
        cin >> x >> y;
      posicion = make_pair(x, y);
      enemigos.push_back((posicion));
    }

    enemigos_global = enemigos;
    for (listaPos_t::iterator it = enemigos.begin(); it != enemigos.end(); ++it) {
           cerr << (*it).first << " " << (*it).second << endl;
    }
    Kamehameha(enemigos, enLaMira, 0);
    mostrarSolucion();
    return 0;
}

void Kamehameha(listaPos_t enemigos, Kamehamehas_t ataques, int nroAtaque) {
```

```
    if (minimo_global <= (int) ataques.size()) {
      return;
    }
    if (enemigos.size() == 0) {
        if (minimo_global > (int) ataques.size()) {
            minimo_global = (int) ataques.size();
            mejor_configuracion = ataques;
        }
    } else {
        for (int i = 0; i < enemigos.size(); i++) {
            posicion_t const enemigo = enemigos[i];
            enemigos.erase(enemigos.begin()+i);
            atacarEnAtaqueActual(enemigo, enemigos, ataques, nroAtaque);
            atacarEnNuevoAtaque(enemigo, enemigos, ataques, nroAtaque);
            listaPos_t::iterator it = enemigos.insert(enemigos.begin()+i, enemigo);
        }
    }
}

void atacarEnAtaqueActual(posicion_t enemigo, listaPos_t restoEnemigos, Kamehamehas_t ataques, int
     nroAtaque) {
    Kamehameha_t atacados = ataques[nroAtaque];
    if (atacados.size() == 0){
        Kamehameha_t comenzarAtaque;
        comenzarAtaque.push_back(enemigo);
        ataques[nroAtaque] = comenzarAtaque;
        Kamehameha(restoEnemigos, ataques, nroAtaque);
        Kamehameha_t reestablecerAtaque;
        ataques[nroAtaque] = reestablecerAtaque;
    } else if (alineados(atacados, enemigo)) {
        ataques[nroAtaque].push_back(enemigo);
        Kamehameha(restoEnemigos, ataques, nroAtaque);
        ataques[nroAtaque].pop_back();
    } else {
        return;
    }
}

void atacarEnNuevoAtaque(posicion_t enemigo, listaPos_t restoEnemigos, Kamehamehas_t ataques, int
     nroAtaque) {
    if (ataques[nroAtaque].size() > 0) {
        Kamehameha_t comenzarAtaque;
        comenzarAtaque.push_back(enemigo);
        ataques.push_back(comenzarAtaque);
        nroAtaque++;
        Kamehameha(restoEnemigos, ataques, nroAtaque);
        nroAtaque--;
        ataques.pop_back();
    }
}

bool alineados (Kamehameha_t atacados, posicion_t enemigo) {
    if (atacados.size() == 0 || atacados.size() == 1) {
        return true;
    } else {
        posicion_t primero = atacados[0];
        posicion_t segundo = atacados[1];
        int termino1 = segundo.second - primero.second;
        int termino2 = enemigo.first - primero.first;
        int termino3 = enemigo.second - primero.second;
        int termino4 = segundo.first - primero.first;
        return termino1*termino2 == termino3*termino4;
    }
}

void reporte(listaPos_t enemigos, Kamehamehas_t ataques, int nroAtaque) {
```

```
    cout << "enemigos: ";
    for (listaPos_t::iterator itL = enemigos.begin(); itL != enemigos.end(); ++itL) {
        cout << "(" << (*itL).first << ", " << (*itL).second << ")";
        if (itL!= enemigos.end()) {
            cout << ", ";
        }
    }
    cout << endl ;
    for (int i = 0; i < ataques.size(); i++) {
        Kamehameha_t ataqueEnIdx = ataques[i];
        cout << "[";
        for (Kamehameha_t::iterator it = ataqueEnIdx.begin(); it != ataqueEnIdx.end(); ++it) {
            cout << "(" << (*it).first << ", " << (*it).second << "), ";
            if (it!= ataqueEnIdx.end()){ cout << ", "; };
        }
        cout << "]" << endl;
    }
}

void mostrarSolucion() {
    cout << mejor_configuracion.size() << endl;
    for (int i = 0; i < mejor_configuracion.size(); i++) {
        Kamehameha_t ataqueEnIdx = mejor_configuracion[i];
        cout << ataqueEnIdx.size() << " ";
        for (Kamehameha_t::iterator it = ataqueEnIdx.begin(); it != ataqueEnIdx.end(); ++it) {
            cout << buscarPosicion(*it)+1 << " ";
        }
        cout << endl;
    }
}

int buscarPosicion(const posicion_t& enemigo) {
    listaPos_t::iterator it = find (enemigos_global.begin(), enemigos_global.end(), enemigo);
    return distance(enemigos_global.begin(), it);
}
```