Algoritmos y Estructuras de Datos III

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

8 de Abril de 2016

Trabajo Práctico Número 1

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

Índice

1. Kaio Ken

1.1. Explicación formal del problema

1.2. Explicación de la solución

1.3. Complejidad del algoritmo

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

1.3.1. Esbozo del algoritmo

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

Algorithm 1 Esbozo del algoritmo de KaioKen

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

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

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

1.3.2. Análisis temporal

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

$$T(1) = 1$$

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

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

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

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

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

La complejidad de este algoritmo es siempre $\Theta(nlogn)$, 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 $\Theta(nlogn)$, sin distinción entre casos, por lo que el análisis de performance es simple.

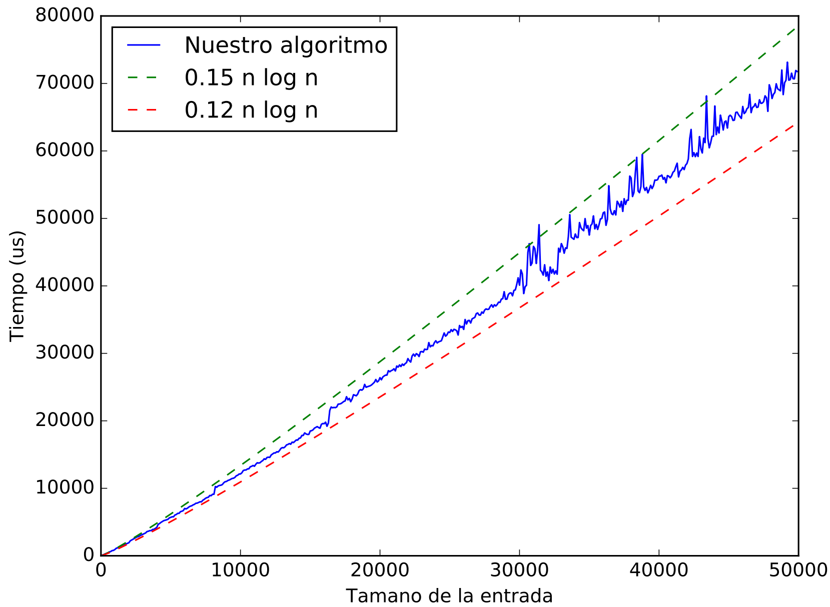

Figura 1: asdfg.

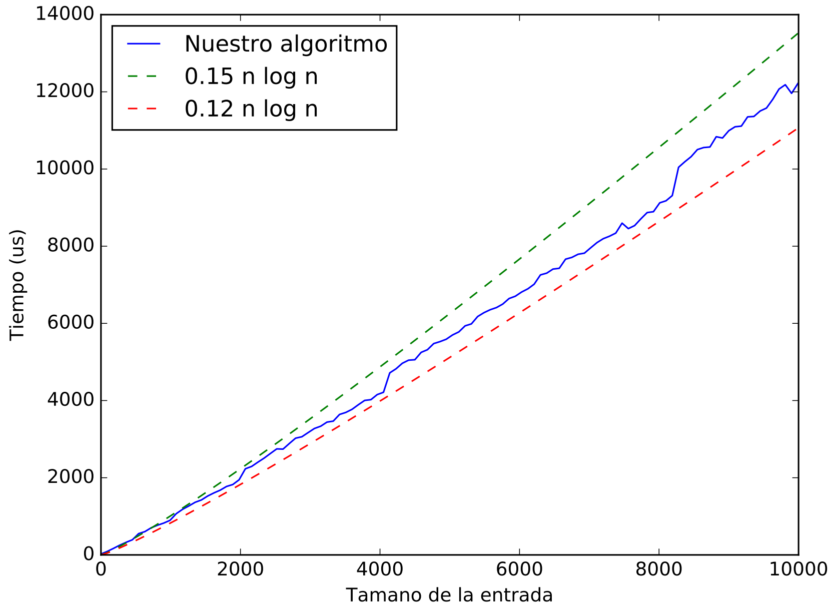

Figura 2: asdfg.

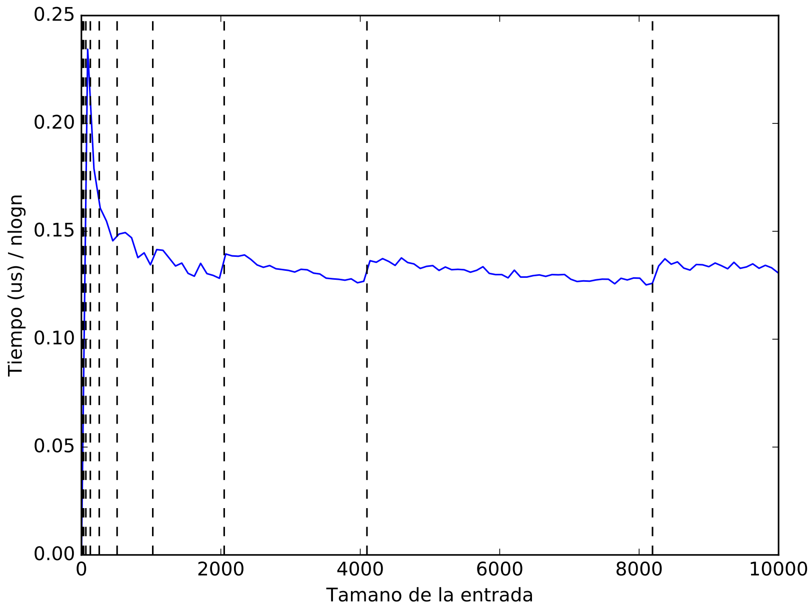

Figura 3: asdfg.

2. Genkidama

2.1. Explicación formal del problema

2.2. Explicación de la solución

2.3. Complejidad del algoritmo

2.4. Performance del algoritmo

3. Kamehameha

3.1. Explicación formal del problema

Sean n puntos distintos, $(x_1, y_1), \ldots, (x_n, y_n)$, todos pertenecientes al primer cuadrante. Es posible trazar semirrectas de modo que eventualmente todos los puntos estén atravesados por alguna de ellas. La idea es hacerlo de tal forma que cuando se agrega una semirrecta se anota cuántos puntos atravesó y más específicamente cuáles, con la condición de que si un punto es atravesado por 2 o más semirrectas solo se anota para una de ellas.

En particular, se desea obtener alguna solución óptima para este problema, es decir, una que utilice la mínima cantidad de semirrectas posibles, listando los puntos de la forma antes explicada.

A continuación, se profundiza la explicación con algunos ejemplos

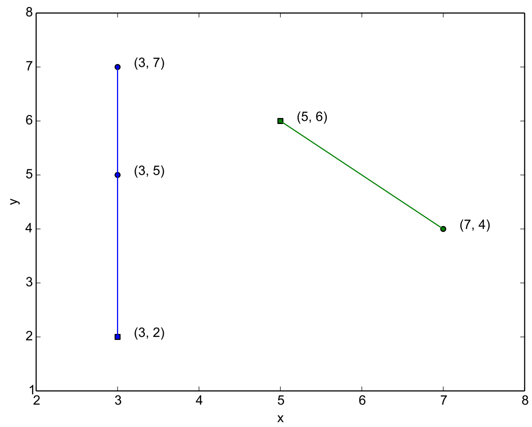

Figura 4: Solución óptima para el conjunto dado de 5 puntos, usando 2 semirrectas.

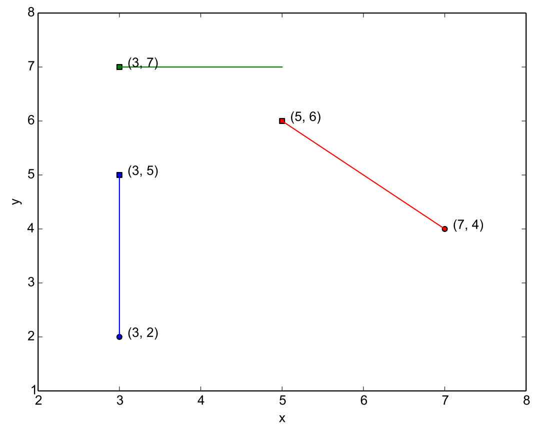

Figura 5: Solución no óptima para el conjunto dado de 5 puntos pues usa 3 semirrectas.

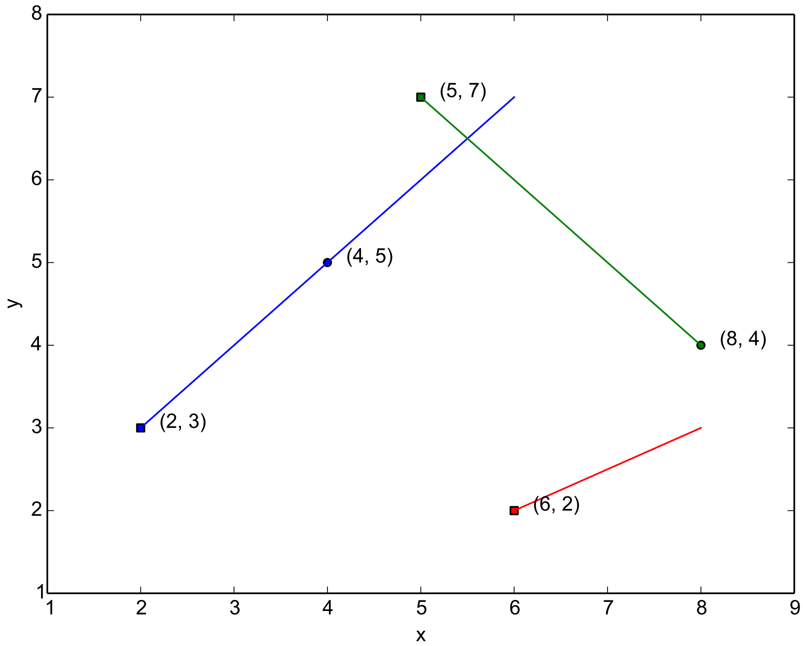

Figura 6: Solución óptima para un conjunto dado de 5 puntos, donde no hay 3 puntos alineados.

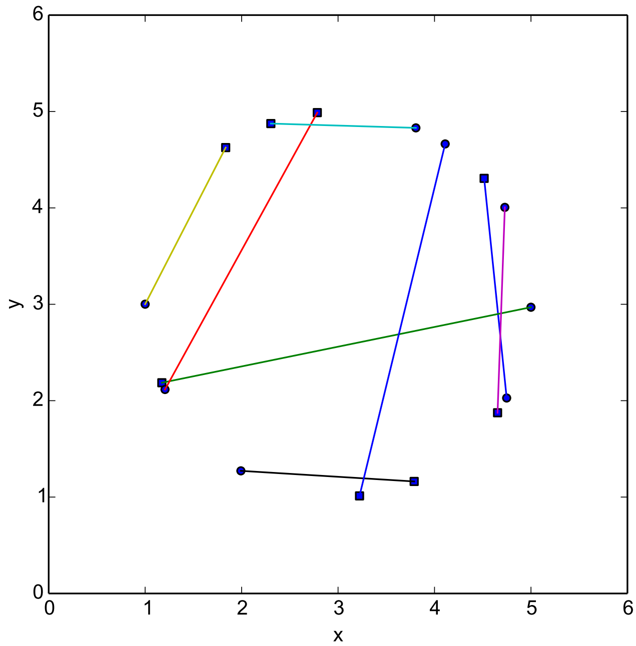

Figura 7: Solución óptima para un conjunto de 16 puntos dispuestos sobre una circunferencia de radio 2 centrada en $(3,3)$.

3.2. Explicación de la solución

Algorithm 2 Pseudocódigo del procedimiento de `backtracking` en Kamehameha

procedure BACKTRACKING(Tablero t, int $step$)
    if $step \geq mejor$ **then**
        $return$
    if $t.Solucionado()$ **then**
        $mejor = step$
        $mejor\_sol = t.Solucion()$
    else

3.3. Complejidad del algoritmo

3.4. Performance del algoritmo

4. Apéndice