INF564 - Diseño avanzado de algoritmos

Tarea 2

Roberto Fuentes Zenteno

21 de junio de 2019

Resumen

En este trabajo se discutirá como administrar los vuelos diarios de un aeropuerto internacional. Para esto, por cada altura se debe analizar cuales son los aviones mas cercanos entre si y advertirles que sus posiciones pueden conducir a una posible colisión. Este problema se reduce entonces a estructurar un algoritmo que sea capaz de entregar los aviones mas cercanos dada una cierta altura mediante dos algoritmos: Fuerza bruta y usando la técnica "Dividir y conquistar". Por otra parte, se estudiarán mejoras asintóticas al algoritmo *Insertion Sort* estudiado en clases, de tal manera que saque ventaja de ciertas entradas que están casi ordenadas.

Keywords: Fuerza Bruta, Dividir y conquistar, Ordenamiento.

1. Controlador de Vuelos

Un aeropuerto internacional debe administrar miles de vuelos diarios. Entre las tareas de la torre de control se encuentra asignar rutas y alturas de vuelo para evitar que los aviones colisionen. En cualquier momento puede haber una gran cantidad de aviones volando a una misma altura, por lo que para un operador puede ser difícil preocuparse de todos ellos al mismo tiempo. Para ayudar a los operadores con este problema se plantea diseñar un algoritmo que indique, por cada altura, los aviones que están más cerca y que por lo tanto tienen mayores probabilidades de colisionar. Así los operadores podrán saber cuales son los aviones críticos a observar. Se debe tener en cuenta que los recursos de la torre de control son limitados (por ejemplo, la memoria de los computadores). Se asume además para los enfoques siguientes que la distancia entre aviones a usar será la distancia euclidiana:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Antes de analizar las distintas técnicas se debe analizar la representación de los aviones. Existen M alturas, donde esta variable será ingresada por el usuario. Para cada altura m, $m \in \{0, 1, \ldots, M\}$ se utiliza el mismo algoritmo para saber la distancia mas corta entre aviones, por lo que nos centraremos en un altura m en particular, ya que para las demás alturas el proceso será el mismo.

Por otra parte, para una altura fija diremos que existen N aviones en vuelo, siendo N ingresado por el usuario. Representaremos estos aviones de la siguiente forma:

$$\begin{aligned} Aviones &= \{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_N\} \\ \mathbf{p}_i &= (x_i, y_i) \\ i &\in \{0, 1, \ldots, N\} \end{aligned}$$

1.1. Fuerza Bruta

En esta sección se buscará implementar un algoritmo para resolver el problema de controlador de vuelos mediante fuerza bruta.

Para esto, la primera idea es elaborar una matriz de distancia que calcule la distancia euclidiana de un avión i con un avión j, con $i \in \{0, 1, \ldots, N\}, j \in \{0, 1, \ldots, N\}, i \neq j$, donde el valor de la distancia es 0 para los casos $i = j$. Esto nos quedaría de la siguiente forma:

$$\begin{bmatrix} 0 & d(\mathbf{p}_!, \mathbf{p}_2) & \ldots & d(\mathbf{p}_1, \mathbf{p}_N) \\ d(\mathbf{p}_2, \mathbf{p}_1) & 0 & \ldots & d(\mathbf{p}_2, \mathbf{p}_N) \\ \vdots & \vdots & \ddots & \vdots \\ d(\mathbf{p}_N, \mathbf{p}_1) & d(\mathbf{p}_N, \mathbf{p}_2) & \ldots & 0 \end{bmatrix} \tag{1}$$

Sin embargo, observamos que el valor de $d(\mathbf{P}_i, \mathbf{P}_j)$ y $d(\mathbf{P}_j, \mathbf{P}_i)$ son los mismos, ya que la distancia del avión i a j es la misma que desde j a i. Siendo así, nos interesará la diagonal superior de (1):

$$\begin{bmatrix} 0 & d(\mathbf{p}_1, \mathbf{p}_2) & \ldots & d(\mathbf{p}_1, \mathbf{p}_N) \\ - & 0 & \ldots & d(\mathbf{p}_2, \mathbf{p}_N) \\ \vdots & \vdots & \ddots & \vdots \\ - & - & \ldots & d(\mathbf{p}_{N-1}, \mathbf{p}_N) \end{bmatrix} \tag{2}$$

Se debe notar que en (2) se omite la última fila de (1), ya que basta con las filas superiores para conocer todas las distancias entre aviones. El pseudocódigo de este algoritmo es el siguiente:

Algorithm 1: *Closest distance brute force*

```
Result: P1,P2
Input: P[], N
int i, j, i_min = 0, j_min = 0;
float dist, aux;
for i = 0, 1, ..., N − 1 do
    for j = i + 1, i + 2, ..., N − 1 do
        if i_min == 0 and j_min == 0 then
            dist = euclidian_distance(P[i], P[j]);
            i_min = i;
            j_min = j;
        else
            aux = euclidian_distance(P[i], P[j]);
            if aux < dist then
                dist = aux;
                i_min = i;
                j_min = j;
            end
        end
    end
end
Return P[i_min], P[j_min]
```

El costo del algoritmo 1 viene dado por:

$$
\begin{aligned}
C_1 &= N-1+N-2+N-3+\cdots+1 \\
&= \sum_{i=1}^{N-1} N-i \\
&= \sum_{i=1}^{N-1} N - \sum_{i=1}^{N-1} i \\
&= N\cdot(N-1) - \sum_{i=1}^{N-1} i \\
&= N\cdot(N-1) - \frac{N(N+1)}{2} \\
&= N^2 - N - \frac{N^2}{2} - \frac{N}{2} \\
&\approx \frac{1}{2}N^2 \\
&\approx N^2 \\
&\therefore \Theta(n^2)
\end{aligned}
$$

Ya que el algoritmo siempre debe recorrer toda la diagonal superior de la matriz, la complejidad del algoritmo es $\Theta(n^2)$. Sin embargo, esto puede ser reducido en la siguiente sección.

1.2. Dividir, Conquistar y Combinar

En esta sección se buscará implementar un algoritmo para resolver el problema de controlador de vuelos mediante dividir, conquistar y combinar. Los pasos son los siguientes:

1. Ordenamos los puntos con respecto a la coordenada x y a la coordenada y, denotando estos puntos como P_x y P_y.

2. Dividimos el conjunto de puntos en dos mitades (izquierda y derecha) tomando como mitad el punto medio del arreglo de puntos P_x $\left(P_x\left[\frac{N}{2}\right]\right)$. Este punto denotará una linea vertical imaginaria que delimitará ambas mitades, al cual llamaremos *mid_point*.

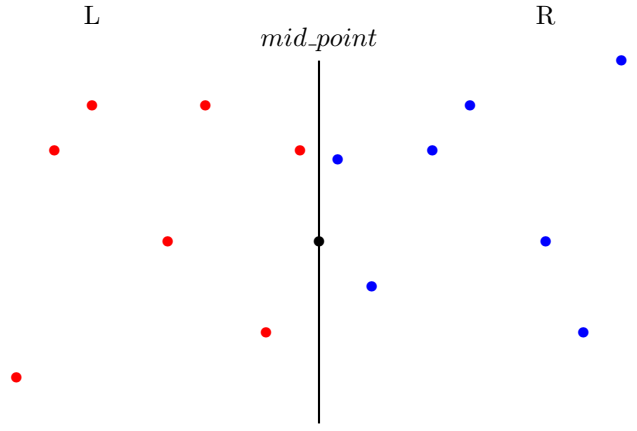

Figura 1: Puntos ordenados y linea divisora por punto medio $P_x\left[\frac{N}{2}\right]$

3. Recursivamente encontramos la distancia más pequeña en ambas mitades, llamando dl la distancia mas pequeña del lado izquierdo y dr la del lado derecho. Si la cantidad de puntos es menor a 3 en alguna mitad, podemos aplicar fuerza bruta para resolverlo, ya que la cantidad de puntos en esta instancia es bastante baja.

4. Tomamos el mínimo entre ambas mitades, llamando a esta distancia d ($d = \min(dl, dr)$).

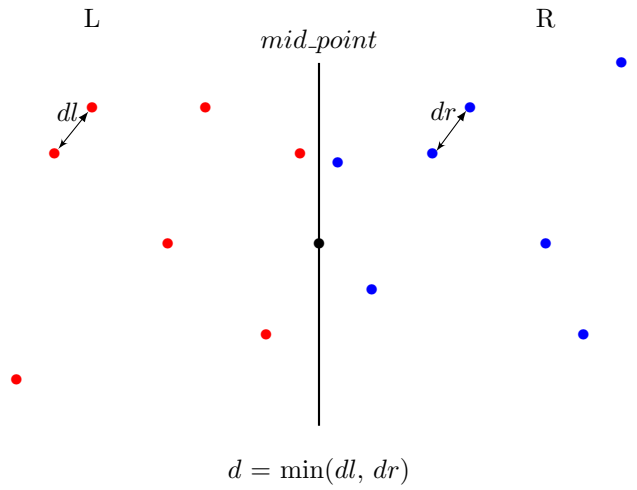

Figura 2: Distancia d obtenida del mínimo entre dl y dr de ambos sectores L y R.

5. Si bien esto parece cubrir todos los puntos, pueden haber distancias entre ambas mitades que se nos escapan por dividir nuestro conjunto de puntos (y que pueden tener distancia menor a d). Por ello, creamos un arreglo temporal *strip* que guardará todos los puntos que estén a una distancia d de la linea que divide ambos conjuntos, ya que el resto de puntos no nos interesan puesto que tienen distancia mayor que d).

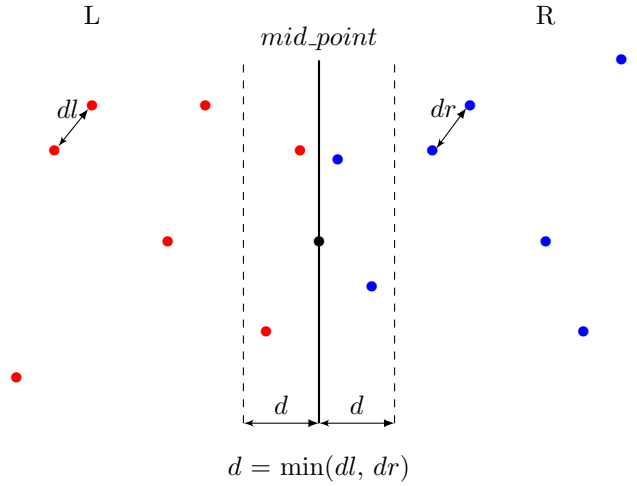

Figura 3: Sector de tamaño $2d$ de puntos a analizar, agregados a *stip*.

6. Encontramos la distancia mas pequeña en *strip*, la cual llamaremos d'. Este paso parece tener en el peor caso complejidad $O(n^2)$ si todos los puntos se ubican en esta zona. Sin embargo, debido a la estructura especial del problema, esto puede ser reducido a complejidad $O(n)$. Para esto, debemos recordar que nuestra cota superior de búsqueda es distancia d, y que el arreglo P_y habia sido previamente ordenado. Consideramos un punto $p \in$ L, y todos los puntos de R con distancia d deben estar en un rectángulo de $d \times 2d$. Con esto dicho, la pregunta es ¿Cuantos puntos pueden estar dentro de este rectángulo si cada par tiene distancia de al menos d? Podemos ver en la figura 4 que el máximo de puntos es 6.

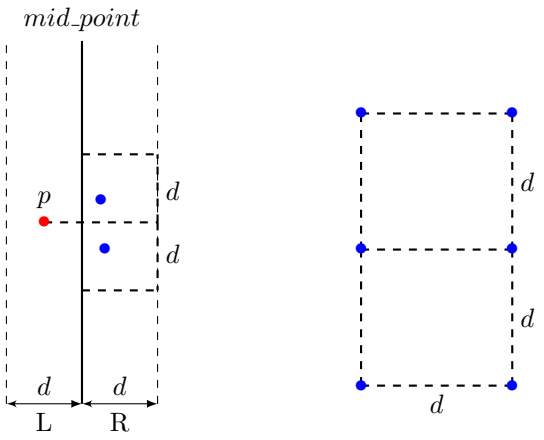

Figura 4: Sector de tamaño $2d$ de puntos a analizar, agregados a *stip*.

Por lo tanto, necesitamos realizar a lo más $6 \cdot \frac{N}{2}$ comparaciones de distancias (6 por cada punto). Para cada p, podemos recorrer ordenadamente todos los puntos en las áreas L y R, en tiempo $O(n)$. Existen otros métodos y explicaciones que mencionan 7 (Cormen et al., 2009, pg 957 [1]) o 15 comparaciones por cada punto (Kleinberg & Tardos, 2006, pg 225 [2]), sin embargo lo importante es que este proceso tomará $O(kn)$ comparaciones y no $O(n^2)$.

7. Finalmente combinamos ambas soluciones, retornando el mínimo entre d y d' (min(d,d')).

El pseudocódigo del algoritmo explicado anteriormente es el siguiente:

Algorithm 2: *Closest distance divide and conquer*

Result: d
Input: P_x, P_y, N
int $i, j, d, k = 0, l = 0, r = 0$;
float *dist*;
if $N < 3$ **then**
  Return closest_distance_brute_force(P_x, N);
end
int mid = $\frac{N}{2}$;
Point mid_point = P[mid];
Pyl[N] ← Arreglo ordenado (por coord. y) de los puntos a la izquierda de mid_point;
Pyr[N] ← Arreglo ordenado (por coord. y) de los puntos a la derecha de mid_point;
for *i = 0,1,. . . , N* **do**
  if *Py[i].x <= mid_point.x* **then**
    Pyl[l] = Py[i];
    l = l+1;
  else
    Pyr[r] = Py[i];
    r = r+1;
  end
end
dlx,dly = divide_and_conquer(Px,Pyl, mid);
drx,dry = divide_and_conquer(Px + mid,Pyr, N - mid);
dl =euclidian_distance(dlx, dly) ;
dr =euclidian_distance(drx, dry) ;
if $dl < dr$ **then**
  $d = d_l$;
  $x_{min} = dlx$;
  $y_{min} = dly$;
else
  $d = d_r$;
  $x_{min} = drx$;
  $y_{min} = dry$;
end
strip ← Arreglo con puntos con distancia en x menor a d;
for $i = 0, 1, \ldots, N$ **do**
  if $|P[i].x$ - *mid_point.x*$| < d$ **then**
    strip[k] = Py[i];
    k = k + 1;
  end
end
for $i = 0, 1, \ldots, k$ **do**
  j = i+1 ;
  while $j < k$ *and* $|strip[j].y - strip[i].y| < d$ **do**
    dist = euclidian_distance(strip[i], strip[j]);
    if $dist < d$ **then**
      d = *dist*;
      $x_{min} = strip[i]$;
      $y_{min} = strip[j]$;
    end
  end
end
Return x_{min}, y_{min}

Para el análisis de complejidad del algoritmo, se tomará como supuesto un caso promedio para el algoritmo de ordenamiento. Siendo así, sea $T(n)$ el costo del algoritmo 2. Pre-ordenamos la data con el algoritmo *QuickSort* ya que en casos promedios este algoritmo es bastante eficiente (complejidad $O(nlog(n))$) y es un algoritmo *in-place*. Sin embargo, para asegurar un menor tiempo de cálculo podemos usar *MergeSort* y así asegurar una complejidad $O(nlog(n))$ como peor caso, aunque para recursos limitados se debe estudiar cuanta memoria extra requerirá el algoritmo 2 junto a *MergeSort* para concluir si es mejor usar esta técnica que la anterior. El algoritmo divide todo el conjunto de puntos en dos grandes mitades, aplicando recursivamente el algoritmo a los subconjuntos respectivos ($2T\left(\frac{n}{2}\right)$). Además, le toma $O(n)$ dividir P_y en torno a la linea vertical. Luego de dividir, crea el arreglo *strip* en $O(n)$. Finalmente el cálculo de la distancia mas pequeña en el arreglo *strip* es de orden $O(n)$. Por lo tanto, $T(n)$ queda como:

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

Aplicando el teorema maestro, tenemos que $a = 2$, $b = 2$ y $d = 1$. Siendo así, la complejidad de nuestro algoritmo es:

$$T(n) = O(nlog(n))$$

Puesto que $d = log_b a$ (ya que $1 = log_2 2$) .

1.3. Resultados

Para las pruebas del algoritmo, se corrieron dos experimentos. El primero se corrió con el 10 alturas cada experimento, con un número de aviones $N = \{1000, 2000, 3000, \ldots, 9000, 10000\}$. El segundo experimento se ejecuto con solo 1 altura para cada set de prueba, con número de aviones $N = \{10, 100, 1000, 10000, 10000, 1000000\}$, olvidándonos de las alturas y viendo la eficiencia del algoritmo solo con el número de aviones. En la figura 5 se muestran los resultados del primer experimento, mientras que la figura 6 refleja los resultados del segundo experimento.

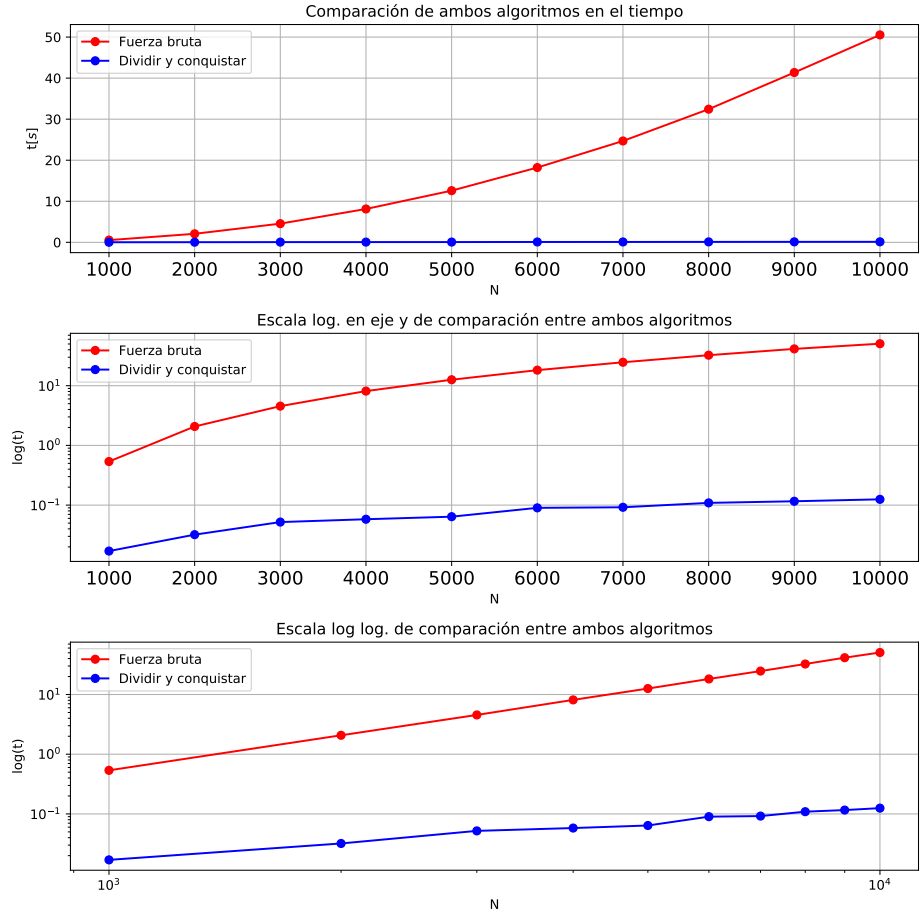

Figura 5: Resultados para el primer experimento, con 10 alturas iguales y $N = \{1000, 2000, 3000, \ldots, 9000, 10000\}$.

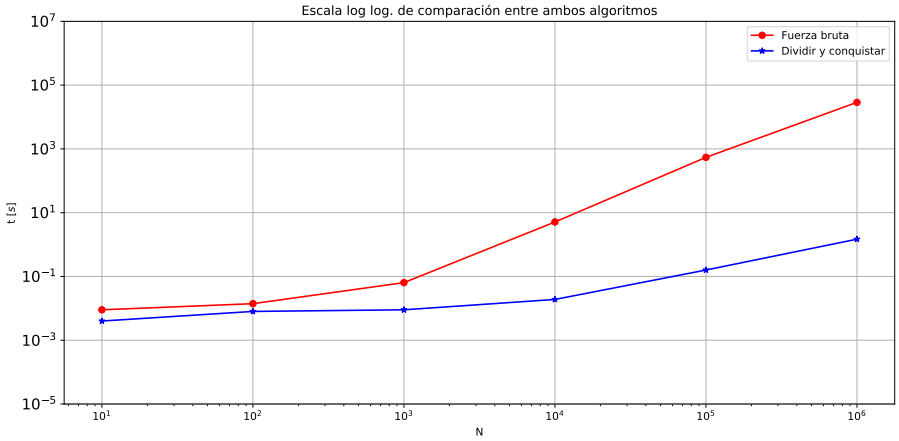

Figura 6: Resultados para el segundo experimento con una altura por experimento, y $N = \{1e^1, 1e^2, 1e^3, 1e^4, 1e^5, 1e^6\}$.

De la figura 5 podemos observar que para el caso del algoritmo 1, la curva se comporta similar a orden $O(n^2)$, mientras que el algoritmo 2 tiende a un comportamiento $O(nlog_2(n))$, como se había estimado anteriormente. Cabe destacar que Shamos y Hoey [3] propusieron el algoritmo dividir y conquistar para encontrar el par de puntos más cercano, siendo la cota inferior para este problema de $\Omega(nlog(n))$.

2. *Insertion Sort*

En esa sección se discutirá una mejora para el algoritmo *insertion sort* tomando en cuenta un arreglo casi ordenado.

Insertion sort es un algoritmo de ordenamiento que toma un arreglo de datos A y entrega el mismo arreglo de forma ordenada (ascendente o descendentemente según se requiera). La idea del algoritmo es avanzar por cada elemento y comparar este con el anterior. Si el antecesor de la posición i ($A[i-1]$) es mayor al elemento actual ($A[i]$), entonces se procede a mover el elemento $A[i]$ hasta que $A[i-1] < A[i]$ o que $j = 0$, mediante la función `swap`:

Algorithm 3: *swap*

Input: A, i, j
int $temp = A[i]$;
$A[i] = A[j]$;
$A[j] = A[temp]$;

La estructura del algoritmo explicado es la siguiente:

Algorithm 4: *Insertion Sort*

Input: A
int i, j, N ;
N = `lenght(A)` ;
for $i = 1, 2, \ldots, N$ **do**
    j = i ;
    while $j > 0$ *and* $A[j-1] > A[j]$ **do**
        `swap(A, j, j - 1);`
        j = j - 1;
    end
end

El peor caso para este algoritmo es similar a varios otros algoritmos de ordenamiento: $O(n^2)$, el cual ocurre si el algoritmo esta ordenado de forma inversa. Sin embargo, el mejor caso es ordenar un arreglo que ya viene ordenado, tomando como tiempo $O(n)$. El caso promedio de este algoritmo es $O(n^2)$, por lo que pierde competitividad frente a otros algoritmo como *QuickSort* o *MergeSort*. No obstante, este algoritmo tiene ciertas ventajas:

- El algoritmo es de tipo *in-place*, es decir que no necesita de memoria extra para ordenar un arreglo.
- Es muy eficiente para arreglos casi ordenados.
- Eficiente para arreglos con pocos datos en comparación con otros algoritmos de ordenamiento.
- Sencillo de implementar en código.

Cabe destacar que es un algoritmo *On-line*, ya que a medida que llegan datos, estos son insertados en el arreglo usando la lógica del algoritmo 4.

La modificación que se propone a este algoritmo es preguntar si $A[i-1] > A[i]$ en cada revisión. Si esta condición se cumple, se procede a realizar las asignación correspondientes *aux* y j. Además, se opta por guardar en *aux* el elemento a ordenar, moviendo todos los elementos una posición a la derecha hasta encontrar la posición adecuada y posicionar dicho elemento.

La modificación se presentan a continuación:

Algorithm 5: *Modified Insertion Sort*

```
Input: A
int i, j, N, aux ;
N = lenght(A) ;
for i = 1, 2, ..., N do
    if A[i − 1] > A[i] then
        aux = A[i];
        j = i;
        while j > 0 and A[j − 1] > aux do
            A[j] = A[j − 1];
            j = j − 1;
        end
        A[j] = aux;
    end
end
```

2.1. Resultados

Se prueban ambos algoritmo con arreglos casi ordenados de tamaño $N = \{1e^1, 1e^2, 1e^3, 1e^4, 1e^5, 1e^6\}$), obteniendo los siguientes resultados:

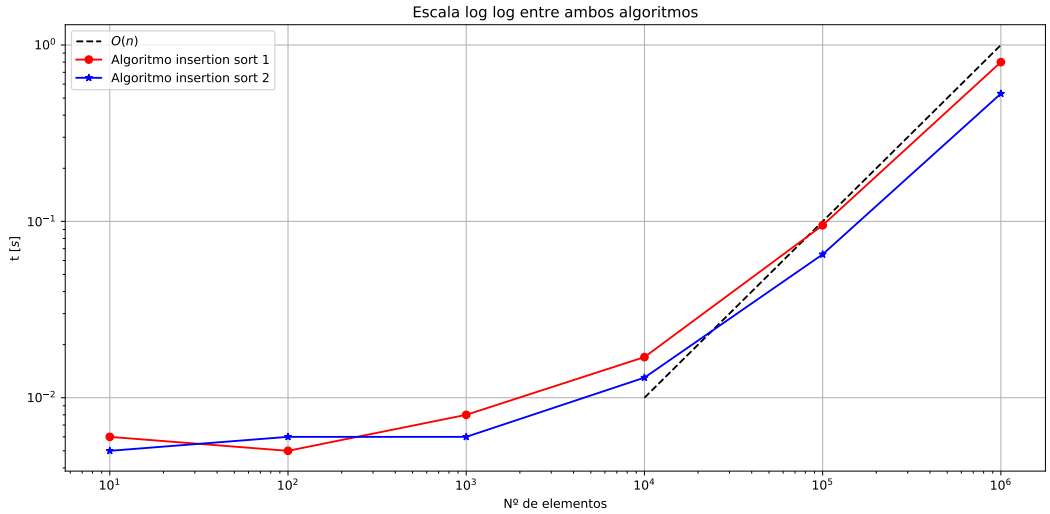

Figura 7: Comparación entre los algoritmos de inserción 4 y 5 con arreglos de tamaño $N = \{1e^1, 1e^2, 1e^3, 1e^4, 1e^5, 1e^6\}$.

Se puede ver que ambos algoritmos se comportan de forma lineal ($O(n)$). Sin embargo, se puede apreciar que el tiempo del algoritmo 5 es menor al algoritmo 4, haciendo más eficiente este algoritmo. Cabe destacar que esto solo tiene sentido para arreglos casi ordenados, ya que para el peor caso y caso promedio, la complejidad sigue siendo $O(n^2)$.

3. Conclusiones

Para el primer problema presentado en este informe, se aprecia el gran impacto que tiene dividir y conquistar el problema a resolverlo por fuerza bruta, quedando demostrados en las curvas de tiempo que tienen ambos algoritmo a medida que aumenta la cantidad de aviones. Si bien el segundo algoritmo es *not in-place*, puesto que por cada llamada a la función se crean arreglos temporales (lo cual puede perjudicar al algoritmo si la cantidad de divisiones es muy grande), esto se ve compensado por el corto tiempo de ejecución que tiene con respecto a la fuerza bruta. La cantidad de divisiones es equivalente a la altura del árbol generado al resolver recursivamente el problema, haciendo que la cantidad de memoria no sea mucha.

Por otra parte, si bien es pequeña la modificación hecha al algoritmo 5, se puede apreciar que en arreglos de gran tamaño esto puede ser beneficioso, sobretodo si es una tarea que se ejecuta regularmente.

Referencias

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[2] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[3] Michael Ian Shamos and Dan Hoey. Closest-point problems. *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 151–162, 1975.