Práctica 2

Divide y vencerás

Algorítmica

segfault

Celia Arias Martínez

Miguel Ángel Fernández Gutiérrez

Sergio Quijano Rey

Lucía Salamanca López

Práctica 2

Divide y vencerás

Algorítmica

segfault

Celia Arias Martínez

Miguel Ángel Fernández Gutiérrez

Sergio Quijano Rey

Lucía Salamanca López

Índice

I Introducción

Esta **práctica 2**, de divide y vencerás, consiste en dos partes principales:

- **Problema común:** traspuesta de una matriz de orden 2^k.
- **Problema asignado:** mezcla de k vectores ordenados.

Objetivo de esta práctica

En esta práctica, pretenderemos aproximar la resolución de problemas mediante la técnica **divide y vencerás**, consistente en la división del problema en subproblemas, y en la resolución de éstos para poder resolver el problema último.

Mediante la aplicación de esta técnica en la resolución de dos problemas diferentes, apreciaremos la utilidad de esta técnica para resolver problemas de forma más eficiente que otras iniciativas sencillas o directas.

Por otra parte, también veremos que esta técnica puede no ser la más favorable en algunas ocasiones, es decir, que las versiones más sencillas pueden tener una mejor complejidad y ser más apropiadas.

Problema asignado

Para el análisis de la eficiencia híbrida, hemos tomado los datos de uno de los alumnos del grupo y hemos hallado la K (constante oculta). Para ello, hemos procedido de dos maneras:

- Tal y como está especificado en el guion de prácticas: hemos realizado la división $\frac{T(n)}{f(n)}$ para cada una de las muestras medias.
- Por otro lado, hemos calculado la constante oculta K mediante la búsqueda de la mejor función de la clase de equivalencia de las funciones $O(T(n))$, y tomando su coeficiente, con la ayuda de *Jupyter*.

II Desarrollo

A continuación, estudiaremos los dos algoritmos propuestos.

1 Traspuesta de una matriz de orden 2^k

> Dada una matriz de tamaño $n = 2^k$, diseñar el algoritmo que devuelva la traspuesta de dicha matriz.

1.1 Enfoque sin divide y vencerás

Para trasponer una matriz cualquiera, basta recorrer la parte triangular superior e ir intercambiando los términos a_{ij} por los a_{ji} de la matriz.

```
void traspuesta_noDyV(int **mat, int N) {
    int aux;

    for ( int i = 0; i < N; i++ )
        for ( int j = i+1; j < N; j++ )
            if ( i != j ) {
                aux = mat[j][i];
                mat[j][i] = mat[i][j];
                mat[i][j] = aux;
            }
}
```

En este código, tenemos que:

- **`mat`** es la matriz que se va a trasponer.
- **`N`** es el tamaño de la matriz.

Procederemos ahora a realizar el análisis teórico y empírico de este algoritmo.

1.1.1 Análisis teórico

Vemos que, al tener dos bucles anidados, el orden de complejidad es:

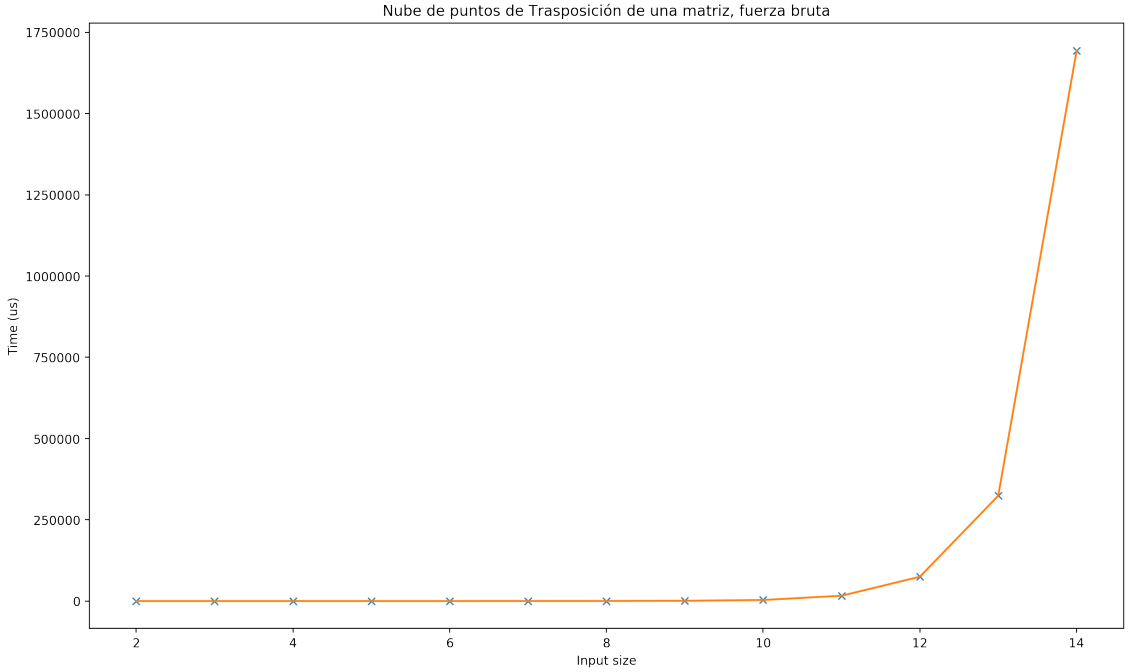
$$T(n) \in O(n^2)$$

Por tanto, tenemos un algoritmo $\boldsymbol{O(n^2)}$.

1.1.2 Análisis empírico

Realizando 100 ejecuciones por cada tamaño de matriz, con tamaños de matriz desde 2^2 hasta 2^{15}. Obtenemos de este modo los siguientes datos:

Gráfico 1.1. Datos empíricos para trasposición de matrices (no divide y vencerás)



Los datos de la gráfica se encuentran en el *Anexo*.

1.2 Enfoque con divide y vencerás

El enfoque divide y vencerás, en este caso, es recursivo: dividiremos la matriz en cuatro trozos e iremos intercambiando las submatrices superior derecha e inferior izquierda, y así sucesivamente.

```
void trasponerRec(int **mat, int inicio_c, int fin_c, int fila) {
    if ( fin_c - inicio_c > 1 ) {
        int aux;

        for ( int i = fila; i < fila + (fin_c - inicio_c)/2; i++ ) {
            for ( int j = inicio_c + (fin_c - inicio_c)/2; j < fin_c; j++ ) {
                aux = mat[i + (fin_c - inicio_c)/2][j-(fin_c - inicio_c)/2];
                mat[i + (fin_c - inicio_c)/2][j-(fin_c - inicio_c)/2] = mat[i][j];
                mat[i][j] = aux;
            }
        }

        trasponerRec(mat, inicio_c, inicio_c+(fin_c-inicio_c)/2, fila);
        trasponerRec(mat, inicio_c+(fin_c-inicio_c)/2, fin_c, fila);
        trasponerRec(mat, inicio_c, inicio_c+(fin_c-inicio_c)/2, fila+(fin_c-inicio_c)/2);
        trasponerRec(mat, inicio_c+(fin_c-inicio_c)/2, fin_c, fila+(fin_c-inicio_c)/2);
    }
}

void trasponer(int **mat, int tam) {
    trasponerRec(mat, 0, tam, 0);
}
```

En este código, tenemos que:

- **`mat`** es la matriz que se va a trasponer.
- **`inicio_c`** es la columna que acota la matriz que se dividirá por la izquierda.
- **`fin_c`** es la columna que acota la matriz que se dividirá por la derecha.
- **`fila`** es la fila donde comienza la submatriz que se dividirá. La altura se puede calcular mediante los parámetros anteriores, al ser las submatrices cuadradas.

1.2.1 Análisis teórico

Para calcular la eficiencia teórica, analizaremos la función `trasponerRec`, que es el que realiza la recursividad. Por tanto, primeramente calcularemos la eficiencia de los dos `for` anidados.

Podemos ver que el `for` interno tiene una eficiencia de:

$$\sum_{inicio_c+\frac{fin_c-inicio_c}{2}}^{fin_c-1} 1 = fin_c - 1 - \left(inicio_c + \frac{fin_c - inicio_c}{2}\right) + 1 = \frac{fin_c - inicio_c}{2}$$

Llamaremos $n = \frac{fin_c-inicio_c}{2}$.

La eficiencia del `for` externo será de:

$$\sum_{fila}^{fila+n-1} n = n(fila + n - 1 - fila + 1) = n^2$$

Obteniendo de este modo la siguiente ecuación en recurrencia:

$$T(n) = n^2 + 4T(n/2)$$

Que resolveremos a continuación:

$$T(n) = n^2 + 4T(n/2)$$
$$T(2^k) = (2^k)^2 + 4T(2^{k-1})$$

$$T(2^k) - 4T(2^{k-1}) = 4^k$$
$$(x-4)(x-4) = (x-4)^2 = 0$$

$$T_k = c_1 * 4^k + c_2 * k * 4^k$$
$$T_n = 2c_1 * n^2 + 2c_2 * n^2 \log n$$

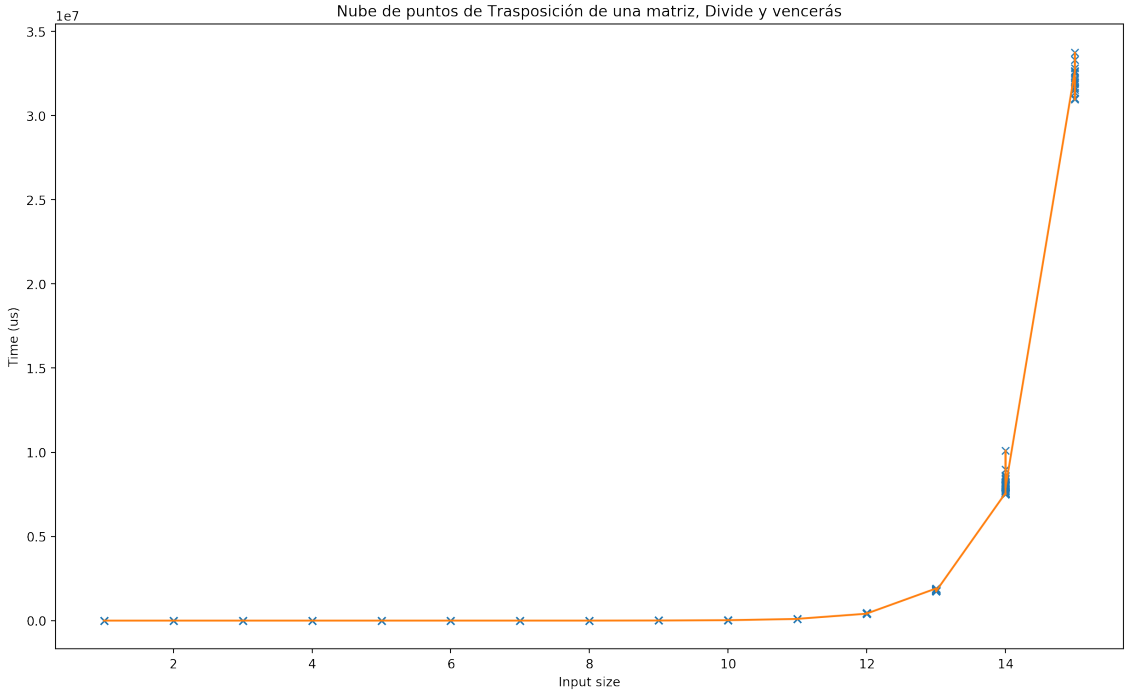
Por lo que:

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

1.2.2 Análisis empírico

Del mismo modo, realizamos 100 ejecuciones por cada tamaño de matriz, con tamaños de matriz desde 2^2 hasta 2^{15}. Obtenemos de este modo los siguientes datos:

Gráfico 1.2. Datos empíricos para trasposición de matrices (divide y vencerás)



Los datos de la gráfica se encuentran en el *Anexo*.

1.3 Comparación de enfoques

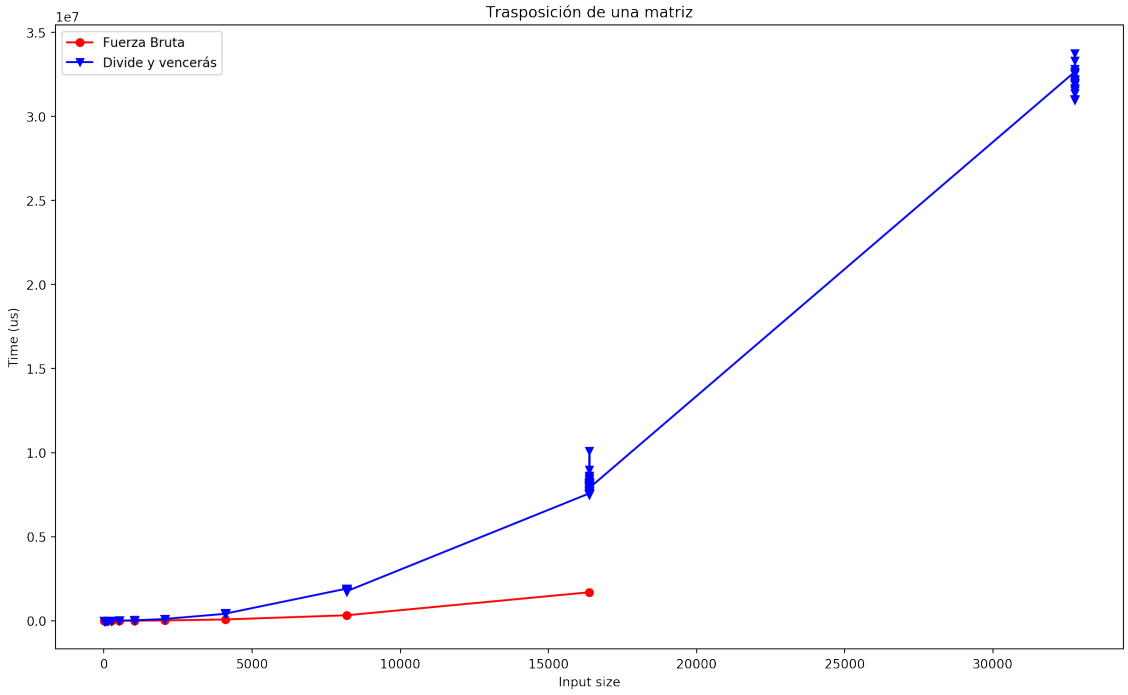
Vemos que tenemos la siguiente complejidad en los algoritmos:

- La versión no divide y vencerás tiene una complejidad $O(n^2)$.
- La versión divide y vencerás tiene una complejidad $O(n^2 \log n)$.

Tenemos que:

$$n^2 \log n \geq n^2 \qquad \forall n \geq 1$$

Gráfico 1.3. Contraste de datos empíricos para trasposición de matrices



Por tanto, tenemos que **la versión no divide y vencerás es más eficiente**. Por tanto, no tiene sentido realizar un estudio de valor umbral alguno, pues esta versión es mejor en todos los casos.

2 Mezcla de vectores ordenados

> Se tienen k vectores ordenados (de menor a mayor), cada uno con n elementos, y queremos combinarlos en un único vector ordenado (con kn elementos).

2.1 Enfoque sin divide y vencerás

Para trasponer una matriz cualquiera, basta recorrer la parte triangular superior e ir intercambiando los términos a_{ij} por los a_{ji} de la matriz.

```
vector<vector<int> > merge_first_two_vectors(vector<vector<int> > matrix) {
    // Matriz con un vector menos, resultado de la mezcla
    vector<vector<int> > merged_matrix(matrix.size() - 1);

    // Vector que hemos mezclado
    vector<int> merged = merge(matrix[0], matrix[1]);

    // Calculamos los datos de la nueva matriz
    merged_matrix[0] = merged;
    for ( int i = 1; i < merged_matrix.size(); i++ )
        merged_matrix[i] = matrix[i+1];

    return merged_matrix;
}

vector<int> merge_vectors_basic(vector<vector<int> > matrix){
    // Caso base para parar la recursividad
    if ( matrix.size() == 1 )
        return matrix[0];
    else {
        matrix = merge_first_two_vectors(matrix);
        return merge_vectors_basic(matrix);
    }
}
```

En este código, tenemos que:

- **`mat`** es la matriz que se va a trasponer.
- **`N`** es el tamaño de la matriz.

Procederemos ahora a realizar el análisis teórico y empírico de este algoritmo.

2.1.1 Análisis teórico

Vemos que, al tener dos bucles anidados, el orden de complejidad es:

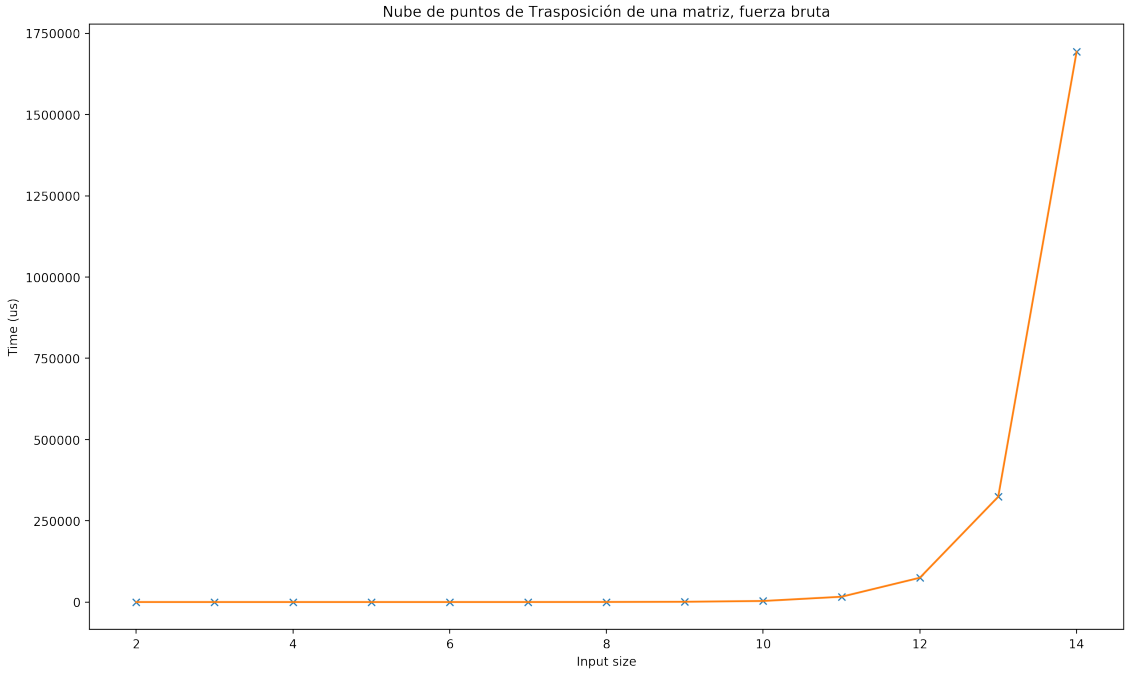
$$T(n) \in O(n^2)$$

Por tanto, tenemos un algoritmo $\boldsymbol{O(n^2)}$.

2.1.2 Análisis empírico

Realizando 100 ejecuciones por cada tamaño de matriz, con tamaños de matriz desde 2^2 hasta 2^{15}. Obtenemos de este modo los siguientes datos:

Gráfico 1.1. Datos empíricos para trasposición de matrices (no divide y vencerás)



Los datos de la gráfica se encuentran en el *Anexo*.

2.2 Enfoque con divide y vencerás

El enfoque divide y vencerás, en este caso, es recursivo: convertimos la matriz original en otras con la mitad de filas y el doble de columnas, donde las filas que obtenemos son los vectores ordenados.

```
vector<int> merge(vector<int> v1, vector<int> v2){
    vector<int> merged(v1.size() + v2.size());
    int pos1 = 0, pos2 = 0;

    while(pos1 < v1.size() || pos2 < v2.size()){
        if(pos1 == v1.size()){
            merged[pos1 + pos2] = v2[pos2]; pos2++;
        }else if(pos2 == v2.size()){
            merged[pos1 + pos2] = v1[pos1]; pos1++;
        }else{
            if(v1[pos1] < v2[pos2]){
                merged[pos1 + pos2] = v1[pos1]; pos1++;
            }else{
            merged[pos1 + pos2] = v2[pos2]; pos2++;
            }
        }
    }
    return merged;
}

vector<vector<int> > merge_two_by_two(vector<vector<int> > matrix){
    // Generamos la nueva matriz
    int new_size = (matrix.size() / 2) + matrix.size() % 2;
    vector<vector<int> > merged_matrix(new_size);

    // Tomamos los datos de la nueva matriz
    for(int i = 0; i < merged_matrix.size(); i++){
        // Ultimo elemento de la matriz sin hacer merge
        if(2*i + 1 >= matrix.size()){
            merged_matrix[i] = merged_matrix[2*i + 1];
        }

        vector<int> new_vector = merge(matrix[2*i], matrix[2*i + 1]);
        merged_matrix[i] = new_vector;
    }
    return merged_matrix;
}

vector<int> merge_divide_and_conquer(vector<vector<int> > matrix){
    // Caso base para finalizar la recursividad
    if(matrix.size() == 1){
        return parse_matrix_to_vector(matrix);
    }
    // Reduzco el tama o del problema a la mitad y aplico recursividad
    matrix = merge_two_by_two(matrix);
    return merge_divide_and_conquer(matrix);
}
```

En este código, tenemos que:

- **`matrix`** es la matriz de vectores que queremos ordenar: el número de filas es el número de vectores que tenemos y el número de columnas el tamaño de vector.

2.2.1 Análisis teórico

Para calcular la eficiencia teórica, analizaremos la función `merge_divide_and_conquer`, que es la función principal, y la función `merge_two_by_two`. La función `merge` es la utilizada en el algoritmo sin divide y vencerás y sabemos que es de tiempo constante para tamaño de vector constante, por tanto es O(1).

- **`merge_divide_and_conquer`** realiza una comprobación de parada que es O(1). Después llama a la función `merge_two_by_two` y vuelve a llamarse a si misma con tamaño de matriz igual a la mitad.
- **`merge_two_by_two`** genera una nueva matriz de tamaño la mitad. Después recorre esta nueva matriz en un bucle for donde llama a funciones que son O(1). Por tanto la eficiencia de la función será O($\frac{n}{2}$).

Por tanto la recurrencia del algoritmo será:

$$T(n) = T(\frac{n}{2}) + \frac{n}{2} * k + 1$$

$$T(n) = T(\frac{n}{2}) + nk + 1$$

$$T(2^m) - T(2^{m-1}) = 2^m k + 1$$

$$(x-1)^2(x-2) = 0$$

$$T_m = c_1 + c_2 m + c_3 2^m$$

$$T_n = c_1 + c_2 log(n) + c_3 n$$

Por lo que:

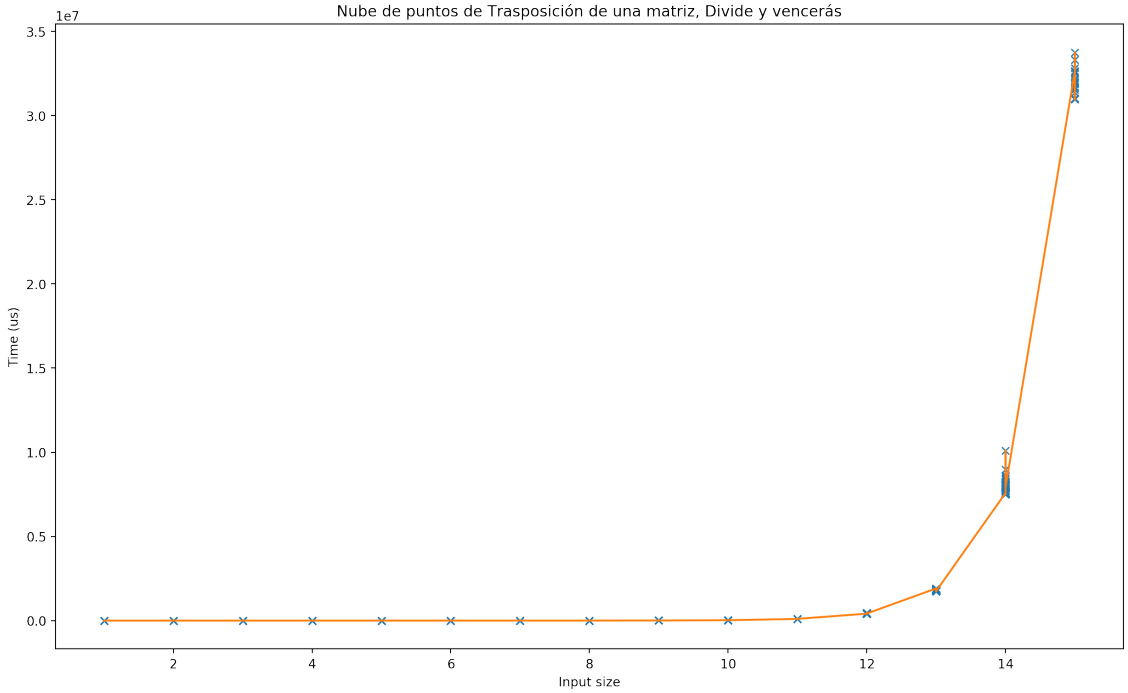
$$T(n) \in O(n)$$

2.2.2 Análisis empírico

Para estudiar el análisis empírico hemos dejado el tamaño de vector constante y hemos variado el número de vectores desde 2^1 hasta 2^{15}. Por cada valor de n hemos realizado 100 ejecuciones y hemos hecho la media para obtener resultados más precisos.

Obtenemos de este modo los siguientes datos:

Gráfico 1.2. Datos empíricos para trasposición de matrices (divide y vencerás)



Los datos de la gráfica se encuentran en el *Anexo*.

2.3 Comparación de enfoques

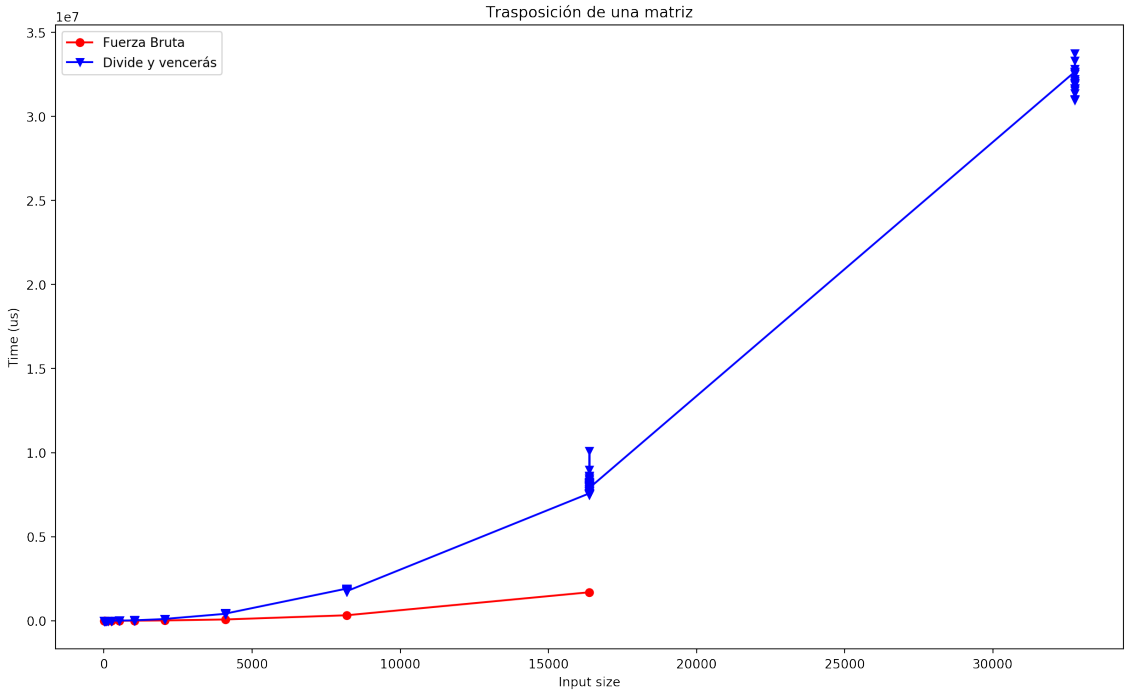
Vemos que tenemos la siguiente complejidad en los algoritmos:

- La versión no divide y vencerás tiene una complejidad $O(n^2)$.
- La versión divide y vencerás tiene una complejidad $O(n^2 \log n)$.

Tenemos que:

$$n^2 \log n \geq n^2 \qquad \forall n \geq 1$$

Gráfico 1.3. Contraste de datos empíricos para trasposición de matrices



Por tanto, tenemos que **la versión no divide y vencerás es más eficiente**. Por tanto, no tiene sentido realizar un estudio de valor umbral alguno, pues esta versión es mejor en todos los casos.

III | Conclusiones

Tras realizar los análisis teóricos de los diversos algoritmos, hemos visto que con los datos obtenidos empíricamente y utilizando una curva de regresión podemos ajustar adecuadamente a la función que deberíamos obtener. Con todo esto, deducimos que nuestro análisis teórico ha sido correcto.

También hemos podido comprobar que los datos obtenidos en la práctica dependen de la arquitectura del ordenador de cada componente del grupo, aún así las funciones son las mismas, siendo esto de mayor relevancia en el análisis de algoritmos.

Comparación entre algoritmos de búsqueda

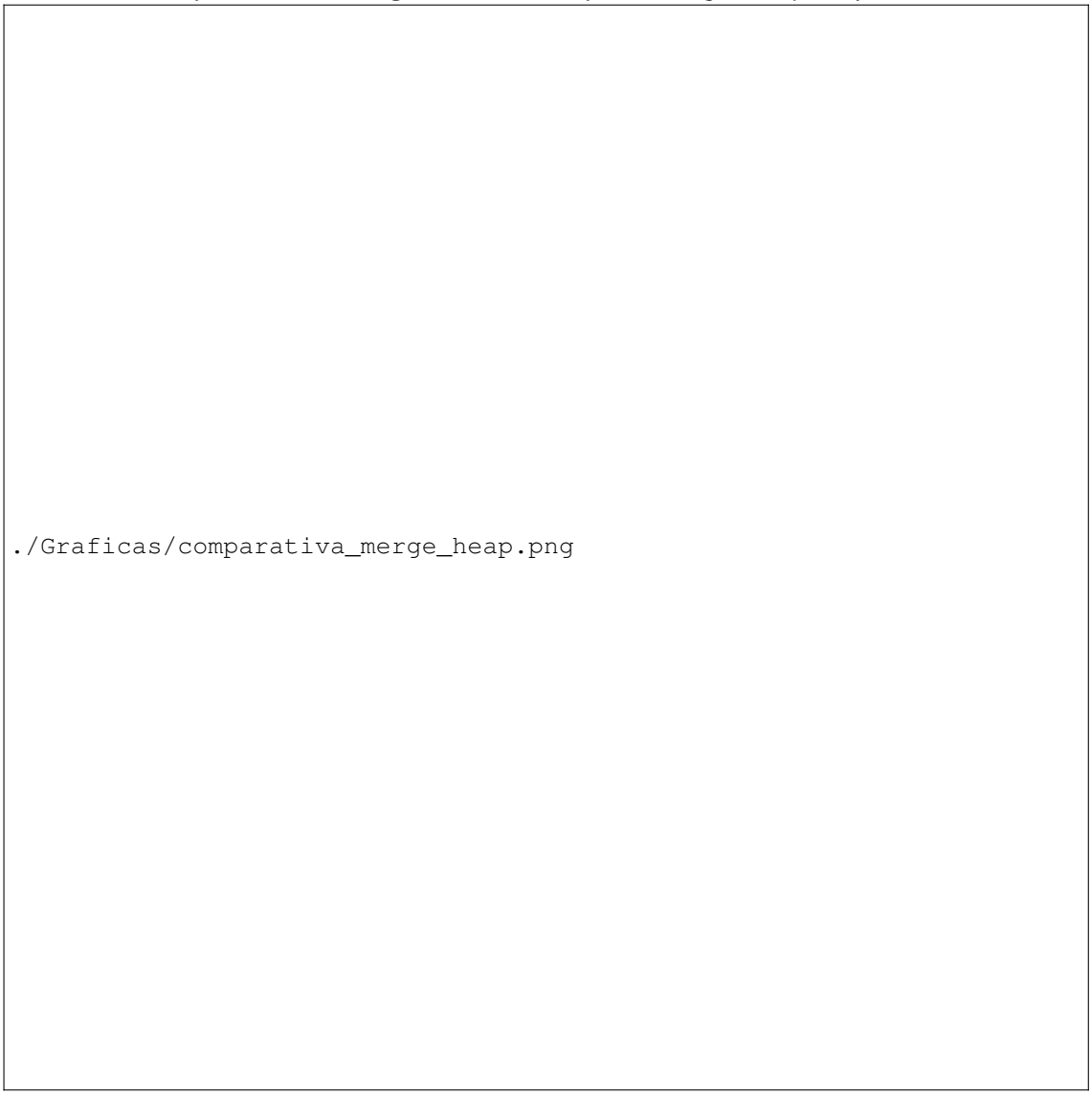
Tras haber ejecutado los códigos y haber analizado los algoritmos teóricamente hemos comprobado que *Heap Sort* y *Merge Sort* son notablemente mejores que *Bubble Sort*. Esto se debe a que el orden de eficiencia de los primeros es $n \log n$, al contrario que *Bubble Sort*, que es n^2.

Gráfico III.1. Comparación entre algoritmos de búsqueda

```
./Graficas/comparativa_todas.png
```

Ya que la diferencia es aplastante, nos centraremos en los algoritmos *Merge Sort* y *Heap Sort*.

Gráfico III.2. Comparación entre algoritmos de búsqueda: *Merge Sort* y *Heap Sort*



Entre estos dos algoritmos con orden de eficiencia mejor, podemos ver que, tras haber realizado un estudio experimental de ambos, el mejor de ellos es *Heap Sort*.

IV | Anexo: datos

Datos 1. Datos de trasposición de matrices

Versión no DyV	
n	**Tiempo** (ms)
4	0
8	0
16	0
32	2
16	2269
32	5248
64	11440
128	25138
256	52225
512	117243

Versión DyV	
n	**Tiempo** (ms)
1	9
2	85
4	312
8	918

Datos 2. Datos de mezcla de vectores

Versión no DyV	
n	**Tiempo** (ms)
4	0
8	0
16	0
32	2
16	2269
32	5248
64	11440
128	25138
256	52225
512	117243

Versión DyV	
n	**Tiempo** (ms)
1	9
2	85
4	312
8	918