UNIVERSIDAD DE GRANADA

Algoritmos Divide y Vencerás

Algorítmica. Práctica 2

Celia Arias Martínez
Miguel Ángel Fernández Gutiérrez
Sergio Quijano Rey
Lucía Salamanca López

segfault

Contenidos

Introducción

Algoritmos Divide y Vencerás

- **Problema común:** traspuesta de una matriz.
- **Problema asignado:** mezclando k vectores ordenados.

Objetivo práctica

Apreciar la utilidad de la técnica *divide y vencerás* (DyV) para resolver problemas de forma más eficiente que otras alternativas más sencillas o directas.

Problema común

Traspuesta de una matriz

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

Problema a asignar

Mezclando k 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).

Desarrollo y análisis de los algoritmos

Algoritmos analizados

Como hemos explicado anteriormente, hemos desarrollado y analizado los siguientes algoritmos:

1. Traspuesta de una matriz
2. Mezclar *k* vectores ordenados

Traspuesta de una matriz (sin DyV)

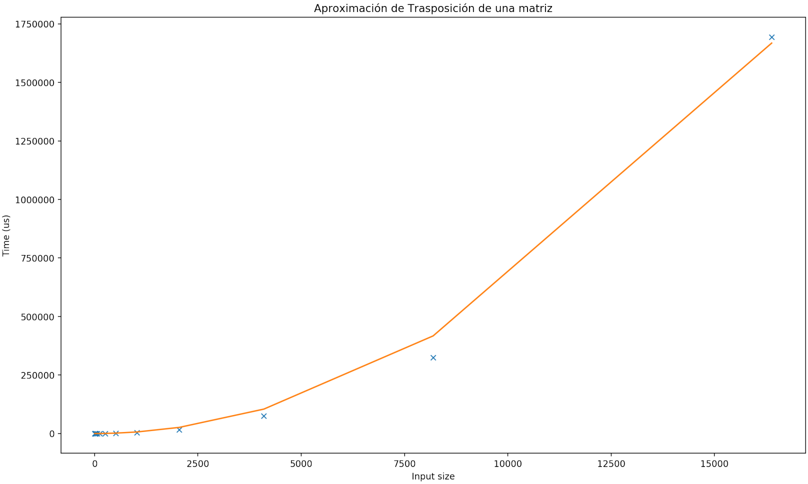
```
void traspuesta_noDyV(int **mat, int N) {
    int aux;

    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;
        }
}
```

Traspuesta de una matriz (sin DyV). Eficiencia teórica

Al ser dos bucles anidados es evidente que

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

Traspuesta de una matriz (sin DyV). Eficiencia empírica



Traspuesta de una matriz

```
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;

    ...
}
```

Traspuesta de una matriz

```
void trasponerRec(int **mat, int inicio_c, int fin_c, int fila){
    ...

    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);
}
```

Traspuesta de una matriz. Eficiencia teórica

Primero vamos a calcular 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 $n = \frac{fin_c-inicio_c}{2}$.

Traspuesta de una matriz. Eficiencia teórica

Eficiencia del `for` externo:

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

Obteniendo de este modo la siguiente ecuación en recurrencia:

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

Traspuesta de una matriz. Eficiencia teórica

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

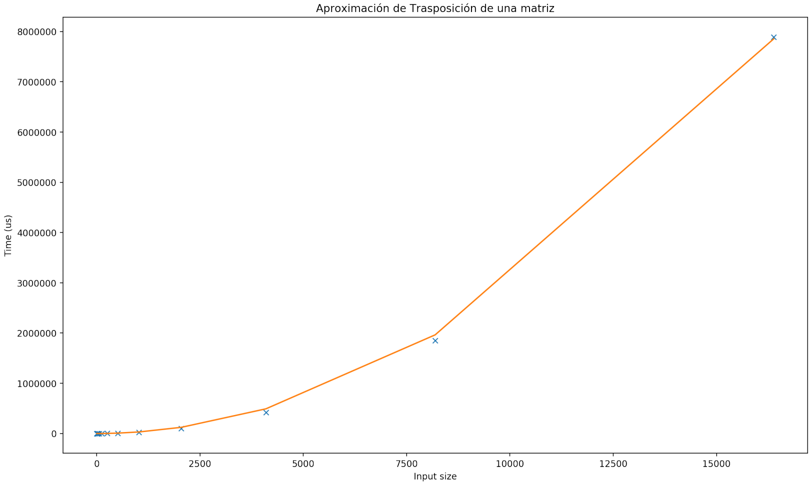
$$T(2^k) - 4T(2^{k-1}) = 4^k$$
$$(x-4)(x-4) = (x-4)^2 = 0$$

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

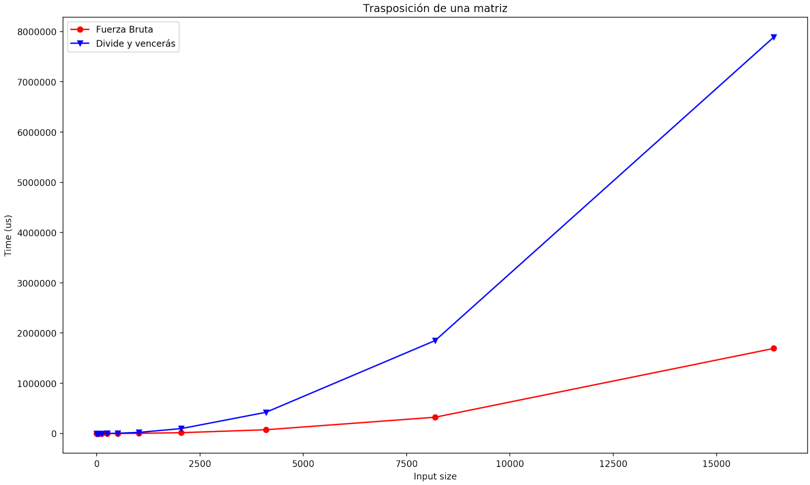
Por lo que:

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

Traspuesta de una matriz. Eficiencia empírica



Traspuesta de una matriz. Gráfica ambos



Mezclar *k* vectores ordenados (sin DyV)

```
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;
}
```

Mezclar *k* vectores ordenados (sin DyV)

```
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;
}
```

Mezclar *k* vectores ordenados (sin DyV)

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

Mezclar *k* vectores ordenados. Eficiencia teórica (sin DyV)

`merge`

En el peor de los casos el algoritmo recorre los dos vectores, el tiempo va a ser $2k$, que es una constante, por lo que:

$$T(n) \in O(1)$$

Mezclar *k* vectores ordenados (sin DyV). Eficiencia teórica

`merge_first_two_vectors`

En la primera parte del código destaca la llamada a la función `merge`, que como hemos calculado anteriormente es $O(1)$, el bucle for también es $O(n)$, por lo que

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

Mezclar *k* vectores ordenados (sin DyV). Eficiencia teórica

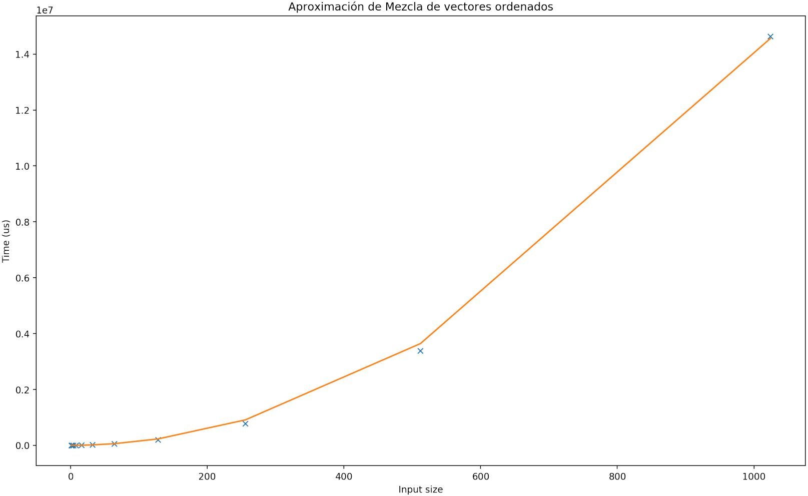
`merge_vectors_basic`

Podemos observar que el algoritmo hace $(n-1)$ veces la función `merge_first_two_vectors` por lo que $T(n) = (n-1) * n$.

Concluimos así que

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

Mezclar *k* vectores ordenados (sin DyV). Eficiencia empírica



Mezclar *k* 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;
}
```

Mezclar *k* vectores ordenados

```
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;
}
```

Mezclar *k* vectores ordenados

```
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);
}
```

Mezclar *k* vectores ordenados. Eficiencia teórica

`merge_two_by_two`

Podemos ver que el tamaño del vector se divide en dos, y luego el for hace $\frac{n}{2}k$ iteraciones, por lo que

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

Mezclar *k* vectores ordenados. Eficiencia teórica

`merge_divide_and_conquer`

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

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

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

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

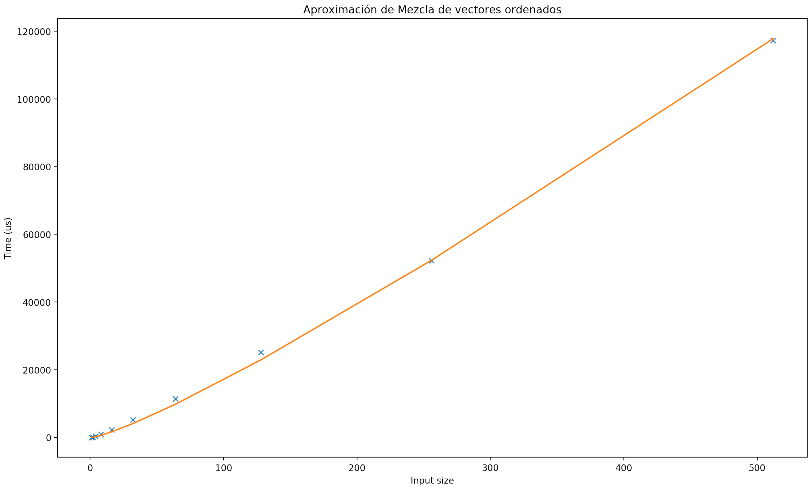
$$T_m = c_1 + c_2 m + c_3 2^m$$

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

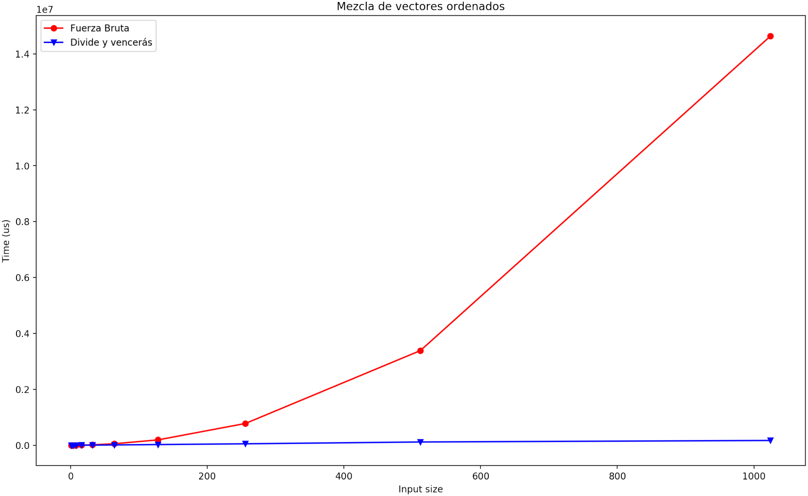
Por lo que:

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

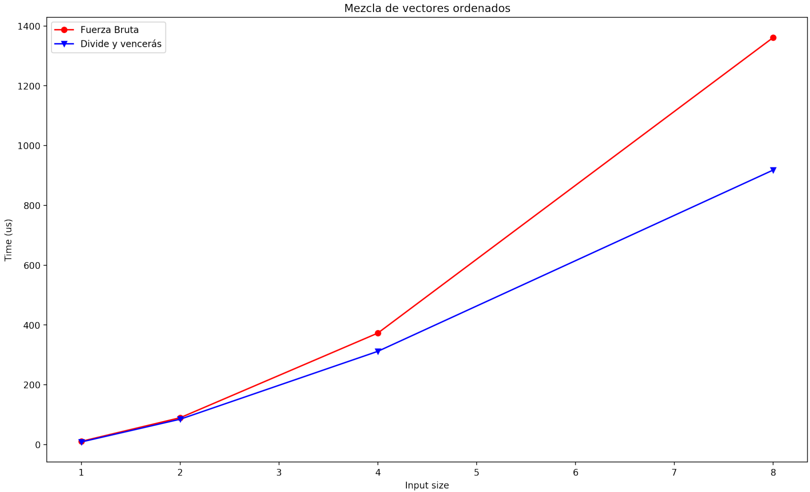
Mezclar *k* vectores ordenados. Eficiencia empírica



Mezclar *k* vectores ordenados. Gráfica ambos



Mezclar *k* vectores ordenados. Estudio del umbral



Mezclar *k* vectores ordenados. Estudio del umbral

En la gráfica anterior podemos ver que el punto de corte de ambas gráficas es 2.

Por lo que podemos afirmar que:

UMBRAL = 2

Conclusión

Conclusión

Podemos observar que la técnica *divide y vencerás* no funciona en algunos casos, como es el de trasponer matrices. Sin embargo, nos ofrece un código con mejor legibilidad y con una consistencia mayor a la hora de mantenerlo.

Por ello es de máxima importancia el análisis, tanto empírico como teórico, previo a la utilización de dicho algoritmo.