

Algorítmica: práctica 2

Mezclando k vectores ordenados

Grupo 2

Sofía Almeida Bruno Antonio Coín Castro María Victoria Granados Pozo
Miguel Lentisco Ballesteros José María Martín Luque

6 de abril de 2017

Introducción

El objetivo de esta práctica es diseñar un algoritmo *divide y vencerás* que se encargue de combinar k vectores ordenados. Además de implementarlo, analizaremos su eficiencia y lo compararemos con un algoritmo clásico, para poder apreciar las ventajas del diseño basado en la técnica *divide y vencerás*.

Algoritmo clásico

A continuación se proporciona el código de la función `mezcla_vectores`, que utiliza un algoritmo clásico para mezclar k vectores en uno solo. El código del programa completo se puede encontrar en la carpeta `src`.

```
1 void merge(int T1[], int T2[], int S[], int n1, int n2) {
2     int p1 = 0, p2 = 0, p3 = 0;
3
4     while (p1 < n1 && p2 < n2) {
5         if (T1[p1] <= T2[p2]) {
6             S[p3] = T1[p1];
7             p1++;
8         }
9         else {
10            S[p3] = T2[p2];
11            p2++;
12        }
13
14        p3++;
15    }
16
17    while (p1 < n1) {
18        S[p3++] = T1[p1++];
19    }
20
21    while (p2 < n2) {
22        S[p3++] = T2[p2++];
23    }
24 }
25 int* mezcla_vectores(int** T, int k, int n) {
26     int* S = new int[k*n]; // Vector mezcla
27     assert(S);
28
29     if (k > 1) {
30         int* aux = new int [k*n];
31         assert(aux);
32
33         // Primera mezcla
34         merge(T[0], T[1], S, n, n);
35
36         // Resto de mezclas
37         for (int i = 2; i < k; i++) {
38             merge(S, T[i], aux, i*n, n);
39             swap(S, aux); // Intercambiamos punteros
40         }
41
42         delete [] aux;
43     }
44
45     else {
46         for (int i = 0; i < n; i++) {
47             S[i] = T[0][i];
48         }
49     }
```

```

50
51     return S;
52 }

```

Eficiencia teórica

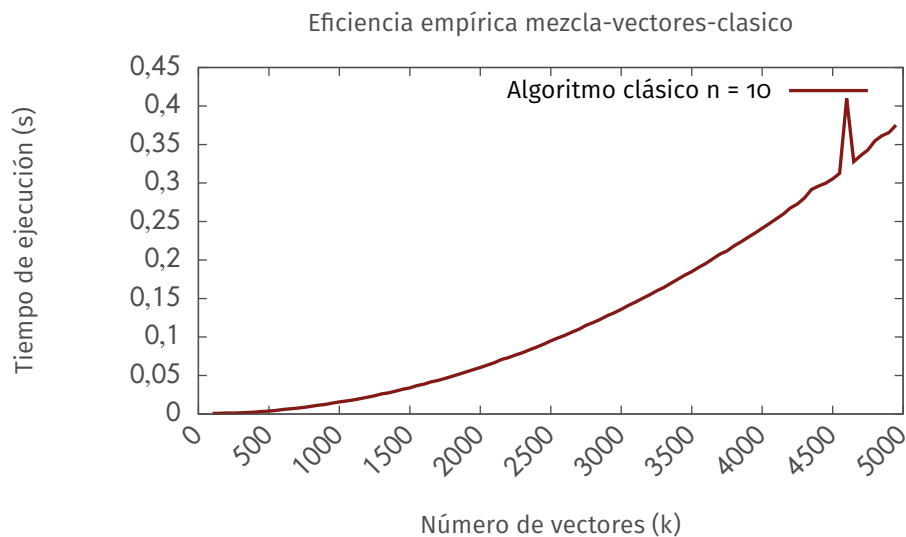
Para calcular la eficiencia teórica de este algoritmo, notaremos primero que solo debemos fijarnos en el bucle `for` de la función `mezcla_vectores`. Además, la función `swap` que intercambia dos punteros tiene eficiencia $O(1)$. Por otro lado, es fácil ver que la función `merge` tiene eficiencia $O(m)$, donde $m = \max\{n_1, n_2\}$. Por tanto, la eficiencia del algoritmo clásico es:

$$T(k) = \sum_{i=2}^{k-1} ni = n \sum_{i=2}^{k-1} i = n \left(\frac{(k-2)(k+1)}{2} \right) = \frac{n}{2}(k^2 - k - 2) \sim \frac{n}{2}k^2$$

Es decir, el orden de eficiencia del algoritmo es $O(k^2)$, donde k es el número de vectores a mezclar.

Eficiencia empírica

En el siguiente gráfico se muestran los resultados de la ejecución del algoritmo *clásico* con vectores de 10 elementos.



Eficiencia híbrida

Algoritmo clásico - versión 2

Como alternativa al algoritmo clásico, hemos planteado una segunda versión donde se crea un vector de tamaño k que guarda las posiciones máximas de cada vector.

Inicializamos el vector de índices con $n-1$ (ya que están todos los vectores ordenados) y pasamos en cada iteración del bucle cogiendo el índice que corresponda hasta obtener el índice del vector que contiene el

máximo valor. Lo añadimos empezando por el final del vector y decrementamos el índice en el vector de índices.

El código del programa completo se puede encontrar en la carpeta `src`.

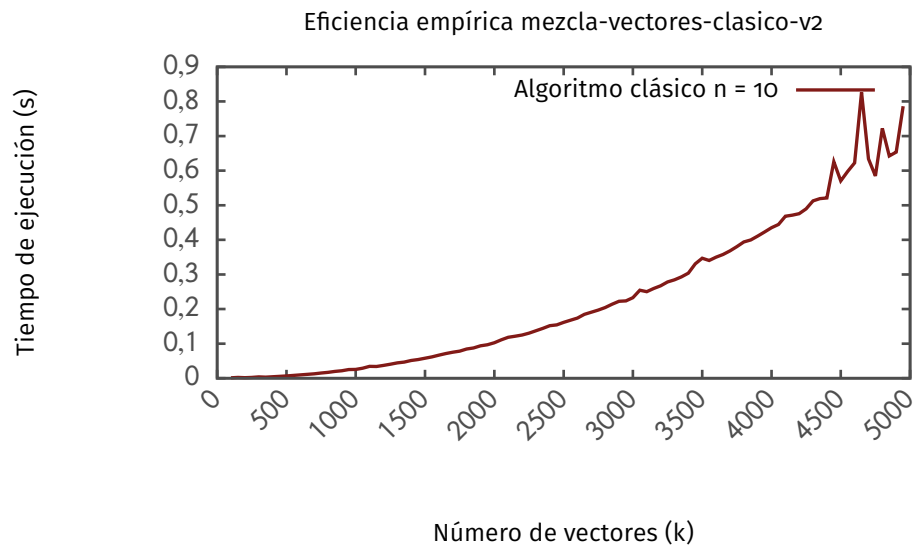
```
1 void mezclarK(int** T, int* res, int n, int k)
2 {
3     // Tamaño total del vector
4     int N = n * k;
5
6     // Vector donde vamos a guardar el índice del valor que falta por meter de cada vector
7     int* v_indices = new int[k];
8
9     // Inicializamos al último índice de cada vector (los máximos)
10    for (int i = 0; i < k; ++i)
11        v_indices[i] = n - 1;
12
13    // El bucle dura hasta N = nk colocados; empezamos en N y acabamos en -1
14    int indice_colocar = N - 1;
15    while (indice_colocar >= 0)
16    {
17        // Este índice indica que vector es el que contiene el valor max.
18        int indice_max = 0;
19        // Recorremos buscando el índice que contenga el valor mayor
20        for (int i = 0; i < k; ++i)
21            if (T[indice_max][v_indices[indice_max]] < T[i][v_indices[i]])
22                indice_max = i;
23        // Guardamos en res con el valor correspondiente
24        res[indice_colocar] = T[indice_max][v_indices[indice_max]];
25        // Decrementamos el índice correspondiente al vector que acabamos de usar
26        --v_indices[indice_max];
27        // Hemos colocado uno
28        --indice_colocar;
29    }
30
31    delete [] v_indices;
32 }
```

Eficiencia teorica

La eficiencia teórica es la misma que la del algoritmo anterior, es decir, $O(k^2)$.

Eficiencia empírica

En el gráfico que se muestra a continuación se muestran los resultados de la ejecución del algoritmo clásico con vectores de 10 elementos.



Eficiencia híbrida

Algoritmo *divide y vencerás* con vectores dinámicos

A continuación se proporciona el código de la función `mezclaDV`, que utiliza un algoritmo *divide y vencerás* (con vectores dinámicos) para mezclar k vectores en uno solo. El código del programa completo se puede encontrar en la carpeta `src`.

```

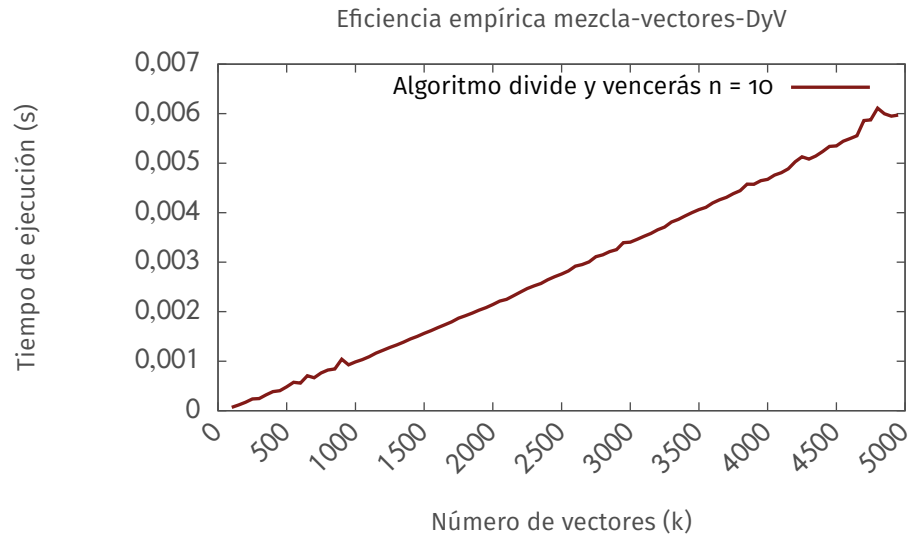
1 int* mezclaDV(int** T, int n, int start, int end) {
2     int k = end - start + 1; // Número de vectores
3
4     // Caso base
5     if (k == 1) {
6         return T[start];
7     }
8
9     // Caso general
10    else {
11        int middle = (start + end) / 2;
12        int n1 = middle - start + 1;
13        int n2 = end - (middle + 1) + 1;
14
15        // Divide
16        int* izqda = mezclaDV(T, n, start, middle);
17        int* dcha = mezclaDV(T, n, middle + 1, end);
18
19        // Vencerás
20        return merge(izqda, dcha, n * n1, n * n2);
21    }
22 }

```

Eficiencia teórica

Eficiencia empírica

En el gráfico que se muestra a continuación se muestran los resultados de la ejecución del algoritmo *divide y vencerás* con vectores dinámicos de 10 elementos.



Eficiencia híbrida

Algoritmo *divide y vencerás* con vectores de la STL

A continuación se proporciona el código de la función `mezclaDV`, que utiliza un algoritmo *divide y vencerás* (con vectores de la STL) para mezclar k vectores en uno solo. El código del programa completo se puede encontrar en la carpeta `src`.

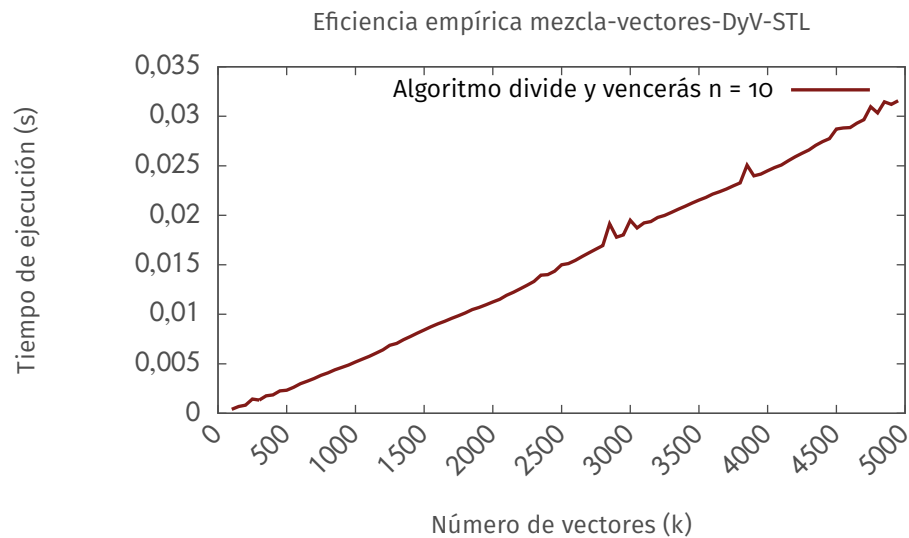
```
1 vector<int> mezclaDV(vector<vector<int>> vectores) {
2
3     // Casos base
4     if (vectores.size() < 1) {
5         vector<int> sol;
6         return sol;
7     } else if (vectores.size() == 1) {
8         return vectores[0];
9     } else if (vectores.size() == 2) {
10        return merge(vectores[0], vectores[1]);
11    }
12
13    vector<vector<int>>::iterator half = vectores.begin() + vectores.size() / 2;
14    vector<vector<int>> firstHalf(vectores.begin(), half), secondHalf(half + 1, vectores.end());
15
16    // Divide
17    vector<int> s1 = mezclaDV(firstHalf);
18    vector<int> s2 = mezclaDV(secondHalf);
19
20    // Vencerás
21    return merge(s1, s2);
```

Eficiencia teórica

Puesto que el programa es muy similar al anterior, la eficiencia de nuevo es $O(k \log k)$.

Eficiencia empírica

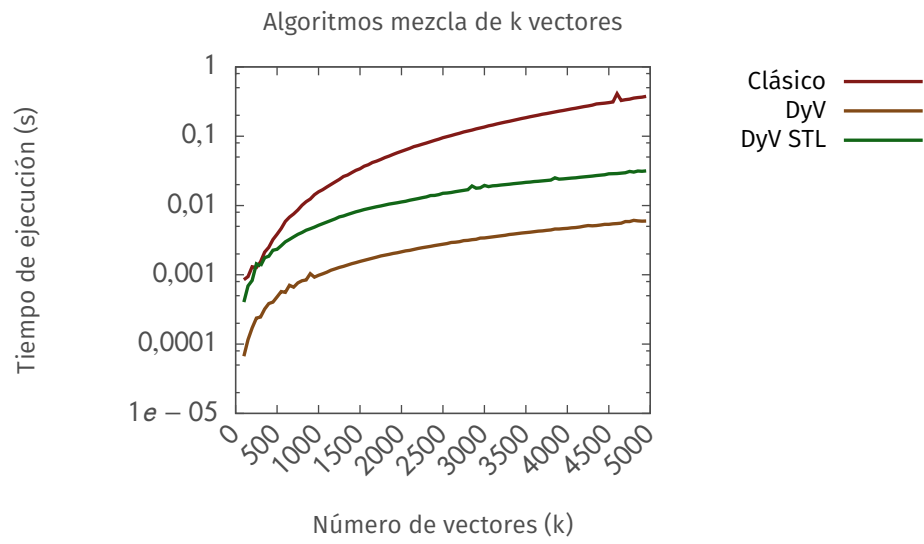
En el gráfico que se muestra a continuación se muestran los resultados de la ejecución del algoritmo *divide y vencerás* con vectores `std::vector` de 10 elementos.



Eficiencia híbrida

Comparación de la eficiencia

En el siguiente gráfico se puede observar de forma visual qué algoritmo es más eficiente. Como era de esperar, el algoritmo clásico es el más lento de todos. Algo más curioso quizás es que el algoritmo que utiliza vectores *dinámicos* es más rápido que el que usa la clase `vector` de la STL.



Hemos utilizado una escala logarítmica para que se puedan ver bien las diferencias de tiempos.

Conclusiones

Como hemos visto, el mismo algoritmo puede programarse de forma más eficiente (y en muchas ocasiones, más simple) empleando la técnica de *divide y vencerás*. En el gráfico comparativo anterior se aprecia que el algoritmo clásico es casi 100 veces más lento que el algoritmo *divide y vencerás*, para los tamaños que hemos ejecutado.

Anexo

Características de los ordenadores donde se ha ejecutado

1. Apple MacBook Pro, Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz, 8GB RAM.
Compilador: clang-800.0.38
Sistema operativo: macOS Sierra
2. Dell XPS 13, Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 8GB RAM.
Compilador: g++ 6.3.1
Sistema operativo: Arch Linux