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*. Para el cálculo de la eficiencia consideraremos que el tamaño de los vectores n será una constante, ya que lo que nos importa en el estudio de este algoritmo es la variable k (número de vectores).

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*.
La idea principal del programa es ir mezclando cada vector con el siguiente, el vector resultante de esta mezcla se combina con el siguiente y así hasta haber mezclado todos los vectores en uno único. Para ello implementamos dos funciones: la primera mezcla dos vectores, y la segunda mezcla los k vectores mediante un bucle en el que vamos llamando a la función anterior.

```
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

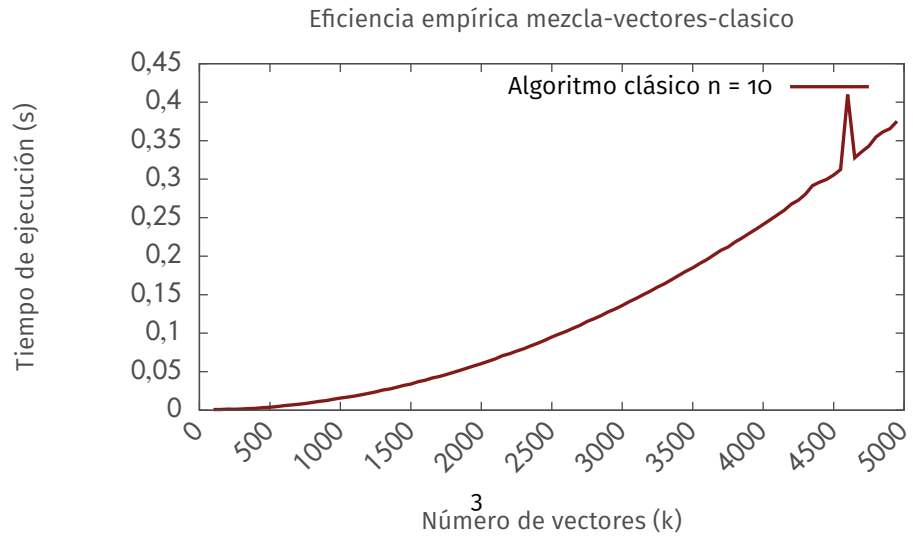
Para calcular la eficiencia teórica de este algoritmo, notaremos primero que solo debemos fijarnos en el bucle `for` de la función `mezcla_vectores`. Además, la función `swap` que intercambia dos punteros tiene eficiencia $O(1)$. Por otro lado, es fácil ver que la función `merge` tiene eficiencia $O(m)$, 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.

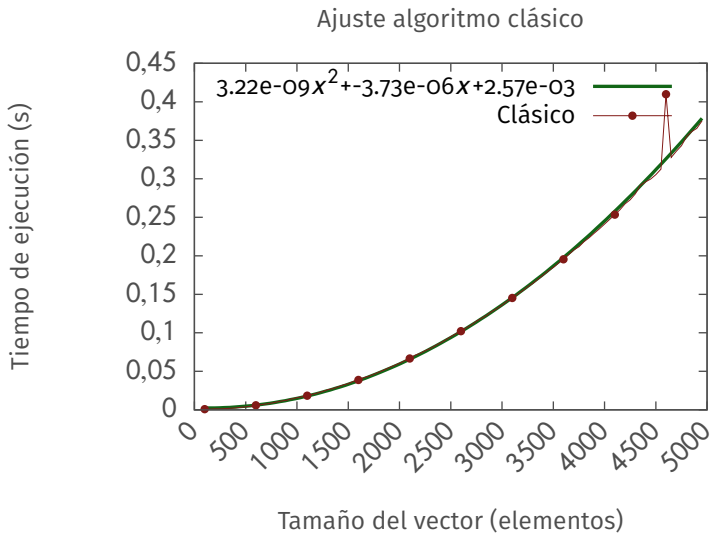
Eficiencia empírica

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



Eficiencia *híbrida*

En la gráfica a continuación se muestran representados mediante puntos los datos obtenidos como resultado de las distintas ejecuciones, y la función que ajusta dichos datos con sus coeficientes correspondientes. La función es $f(x) = a_0x^2 + a_1 * x + a_2$.



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 a $n - 1$ (ya que están todos los vectores ordenados) y en cada iteración del bucle comparamos el índice correspondiente 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    // Vector donde vamos a guardar el indice del valor que falta por meter de cada vector
6    int* v_indices = new int[k];
7    // Inicializamos al último indice de cada vector (los máximos)
8    for (int i = 0; i < k; ++i)
9      v_indices[i] = n - 1;
10   // El bucle dura hasta N = nk colocados; empezamos en N y acabamos en -1
11   int indice_colocar = N - 1;
12   while (indice_colocar >= 0)
13   {
14     // Este indice indica que vector es el que contiene el valor max.
```

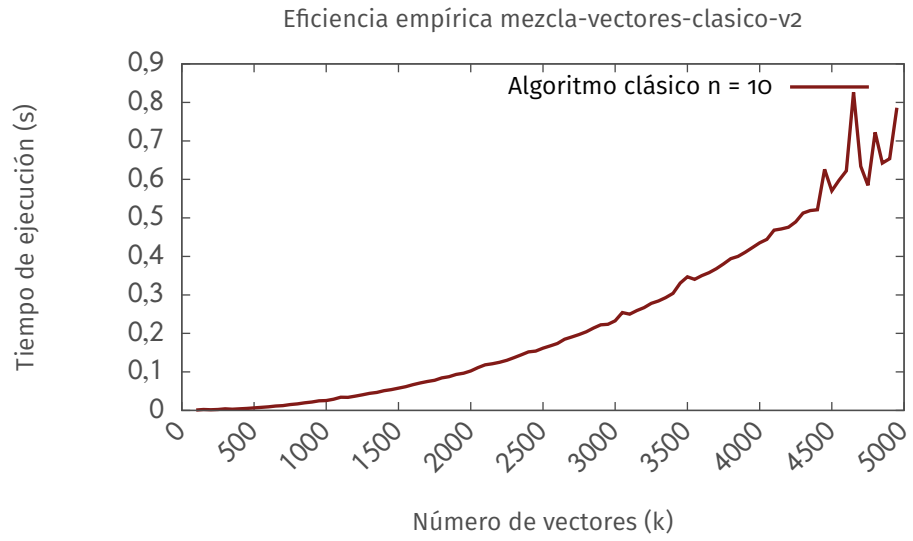
```
15      int indice_max = 0;
16      // Recorremos buscando el indice que contenga el valor mayor
17      for (int i = 0; i < k; ++i)
18        if (T[indice_max][v_indices[indice_max]] < T[i][v_indices[i]])
19          indice_max = i;
20      // Guardamos en res con el valor correspondiente
21      res[indice_colocar] = T[indice_max][v_indices[indice_max]];
22      // Decrementamos el indice corresponiente al vector que acabamos de usar
23      --v_indices[indice_max];
24      // Hemos colocado uno
25      --indice_colocar;
26    }
27    delete [] v_indices;
28  }
```

Eficiencia teórica

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

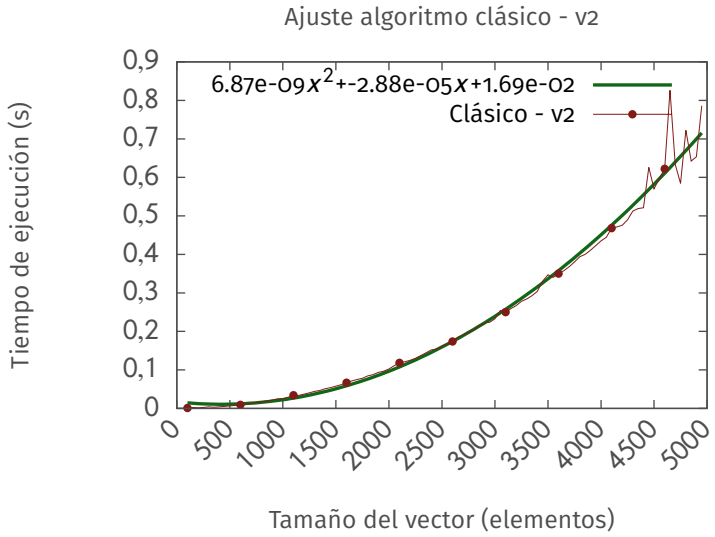
Eficiencia empírica

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

Hacemos el ajuste para una parábola: $f(x) = a_0x^2 + a_1 * x + a_2$



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*.

Para seguir la filosofía *divide y vencerás* en este problema, basta con dividir el número de vectores que mezclar en dos partes (con la mitad de elementos), cada una de las cuales se mezclará utilizando el mismo algoritmo. Como caso base tomamos $k = 1$, en el que devolvemos el mismo vector.

```
int* mezclaDV(int** T, int n, int start, int end) {
  int k = end - start + 1;  // Número de vectores

  // Caso base
  if (k == 1) {
    return T[start];
  }

  // Caso general
  else {
    int middle = (start + end) / 2;
    int n1 = middle - start + 1;
    int n2 = end - (middle + 1) + 1;

    // Divide
    int* izqda = mezclaDV(T, n, start, middle);
    int* dcha = mezclaDV(T, n, middle + 1, end);

    // Vencerás
    return merge(izqda, dcha, n * n1, n * n2);
  }
}
```

Eficiencia teórica

Escribimos la ecuación general recursiva en función de k:

$$\begin{cases} T(1) = 1 \\ T(k) = 2T\left(\frac{k}{2}\right) + n\frac{k}{2} \end{cases}$$

Para resolverlo, tomamos el cambio de variable $k = 2^m$:

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

Resolviendo esta ecuación en diferencias, tenemos que:

$$T(2^m) = 2^m c_1 + 2^{m-1} n m c_2$$

Teniendo ahora que: $k = 1 \iff 2^m = 1 \iff m = 0$ y $T(1) = 1$

$$T(1) = c1 = 1$$

Finalmente: $k = 2 \iff 2^m = 2 \iff m = 1$ y $T(2) = n$

$$T(2) = 2 + nc_2 = n \iff nc_2 = n - 2 \iff c_2 = \frac{n-2}{n}$$

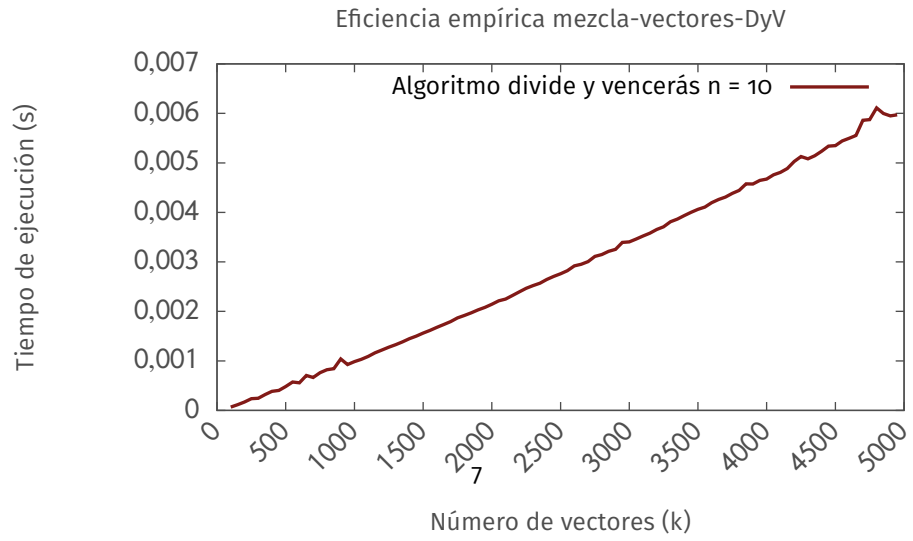
Obtenemos la solución:

$$T(k) = \frac{k}{2} n \log k$$

Concluimos por tanto que la eficiencia del algoritmo *divide y vencerás* es de $O(k \log k)$

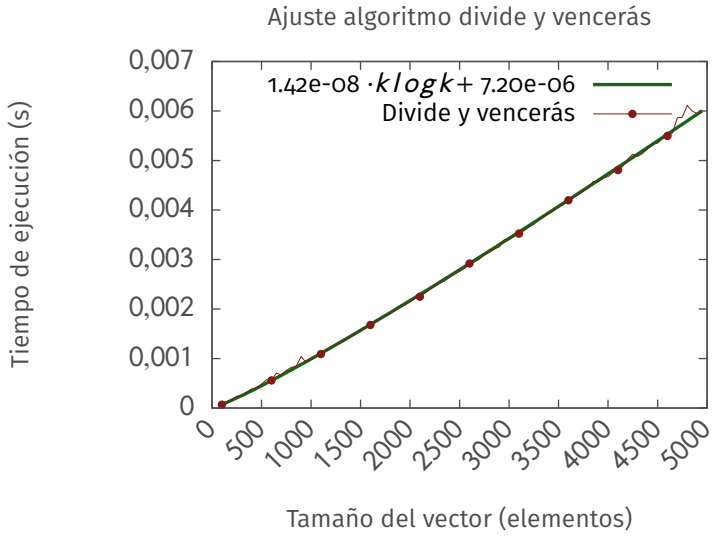
Eficiencia empírica

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



Eficiencia *híbrida*

Para la eficiencia híbrida realizamos un ajuste a la función: $f(x) = b_0 * 10 * x * log(x) + b1$



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*.
En este caso utilizamos un planteamiento similar al anterior, únicamente cambiando la estructura de datos utilizada.

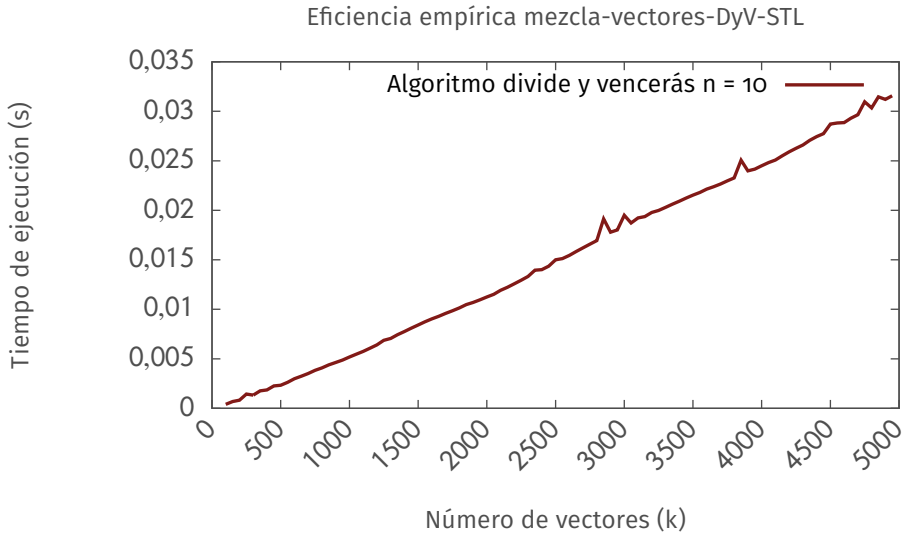
```
vector<int> mezclaDV(vector<vector<int>> vectores) {

    // Casos base
    if (vectores.size() < 1) {
        vector<int> sol;
        return sol;
    } else if (vectores.size() == 1) {
        return vectores[0];
    } else if (vectores.size() == 2) {
        return merge(vectores[0], vectores[1]);
    }

    vector<vector<int>>::iterator half = vectores.begin() + vectores.size() / 2;
    vector<vector<int>> firstHalf(vectores.begin(), half), secondHalf(half + 1, vectores.end());

    // Divide
    vector<int> s1 = mezclaDV(firstHalf);
    vector<int> s2 = mezclaDV(secondHalf);

    // Vencerás
    return merge(s1, s2);
}
```

Eficiencia teórica

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

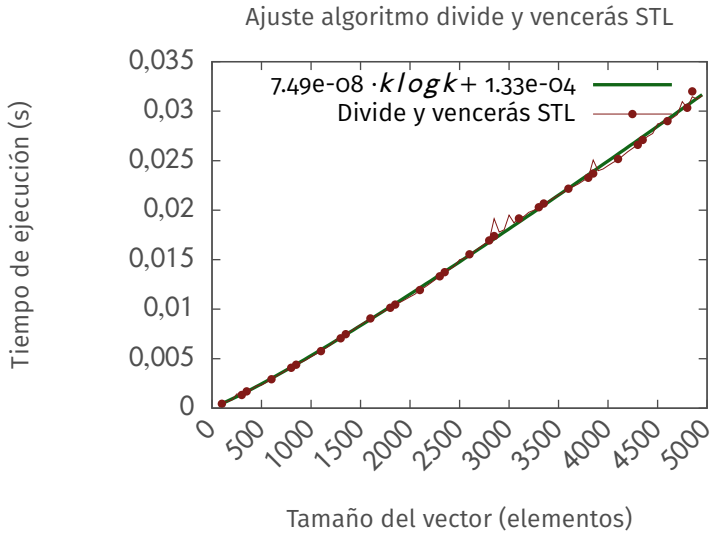
Eficiencia empírica

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



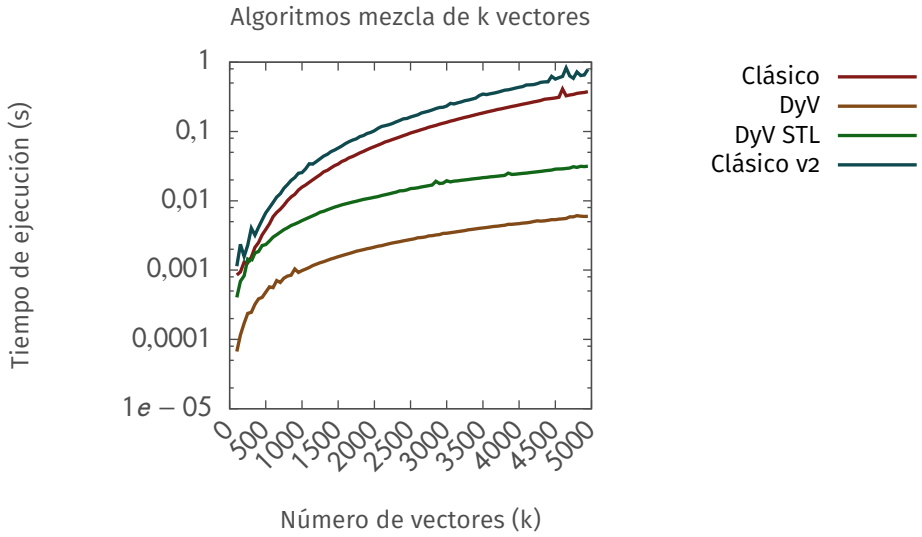
Eficiencia *híbrida*

La función a la que ajustamos los datos en esta ocasión es similar al caso anterior.



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