UNIVERSIDAD DE GRANADA

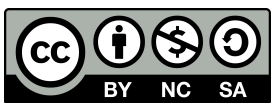
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 2: Divide y Vencerás

Doble Grado Ingeniería Informática y Matemáticas

Autores:
Jose Alberto Hoces Castro
Javier Gómez López
Moya Martín Castaño

Abril 2022



Índice

1. Introducción

El objetivo de esta práctica es utilizar la técnica "divide y vencerás" para resolver problemas de forma más eficiente que otras alternativas más sencillas o directas. Para ello, se plantean los siguientes dos problemas:

- **Ejercicio 1:** Este problema consiste en realizar la búsqueda de un elemento en un vector ordenado con n elementos.
- **Ejercicio 2:** Este problema consiste en dados k vectores de n elementos, todos ellos ordenados de menor a mayor, combinar todos los vectores en uno único ordenado.

2. Desarrollo

Para los análsis de algoritmos que nos pedirán más adelante, hemos realizado los siguientes pasos:

1. Un **análisis teórico** de los algoritmos usando las técnicas vistas en clase.
2. Un **análisis empírico** donde hemos ejecutado los algoritmos en nuestros ordenadores bajo las misma normas y condiciones. Hemos compilado usando la compilación `-Og`. Además hemos usado como *datasets* de pruebas generadores de datos aleatorios proporcionados por la profesora. Por otro lado, para automatizar el proceso, hemos generado unos *scripts* de generación de datos de prueba y de ejecución de nuestros programas. Hemos ejecutado cada algoritmo 15 veces en cada uno de los tamaños que han sido probados, y hemos hecho la media de ellos para reducir perturbaciones que puedan alterar el resultado.
3. Un **análisis híbrido** donde hemos tomado los datos de cada uno de los alumnos del grupo y hemos hallado la K (constante oculta). Para ello hemos usado gnuplot.

2.1. Ejercicio 1

El enunciado del problema es el siguiente: *Dado un vector ordenado (de forma no decreciente) de números enteros v, todos distintos, el objetivo es determinar si existe un índice i tal que $v[i] = i$ y encontrarlo en ese caso. Diseñar e implementar un algoritmo "divide y vencerás" que permita resolver el problema. ¿Cuál es la complejidad de ese algoritmo y la del algoritmo "obvio" para realizar esta tarea? Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.*

Supóngase ahora que los enteros no tienen por qué ser todos distintos (pueden repetirse). Determinar si el algoritmo anterior sigue siendo válido, y en caso negativo proponer uno que sí lo sea. ¿Sigue siendo preferible al algoritmo obvio?

2.1.1. Algoritmo de fuerza bruta

La manera obvia de resolver este ejercicio sería mediante un algoritmo secuencial, que vaya recorriendo el vector hasta encontrar el elemento buscado. Pasemos a realizar un análisis de este algoritmo:

1. **Análisis Teórico**

```
int buscarSecuencial(int v[], int n){
    for (size_t i = 0; i < n; i++) //O(n)
    {
        if (v[i] == i){ //O(1)
            return i; //O(1)
        }
    }

    return -1;//O(1)
}
```

Tal y como se ha indicado en los comentarios del código, todas las operaciones de asignación y comprobación de los `if` son $\mathcal{O}(1)$. Estas, a sus vez, se incluyen dentro de un bucle `for`. Dicho bucle es $\mathcal{O}(n)$, obteniendo así que la función `int buscarSecuencial` es $\mathcal{O}(n)$, es decir

$$T(n) \in \mathcal{O}(n)$$

2. **Análisis Empírico**

Tras ejecutar el algoritmo para 26 tamaños distintos; desde 1000000 hasta 20000000 dando saltos de 760000, hemos obtenido los siguientes resultados:

Algoritmo de fuerza bruta	
Elementos (n)	**Tiempo (s)**
1760000	0.0165694
2520000	0.0262689
3280000	0.0336055
4040000	0.0368924
4800000	0.0399273
5560000	0.0485439
6320000	0.0529679
7080000	0.0585823
7840000	0.0649594
8600000	0.0723527
9360000	0.0801981
10120000	0.0856522
10880000	0.0922361
11640000	0.0992702
12400000	0.105115
13160000	0.114969
13920000	0.118283
14680000	0.123955
15440000	0.132098
16200000	0.139156
16960000	0.146774
17720000	0.150614
18480000	0.157312
19240000	0.163214
20000000	0.169743

Cuadro 1: Experiencia empírica de la búsqueda a fuerza bruta

3. **Análisis Híbrido**

A continuación, aprovechamos los tiempos obtenidos en el análisis empírico para hallar la función de ajuste, para lo cual hemos hecho uso de gnuplot de la misma forma que en la anterior práctica. He aquí la gráfica correspondiente:

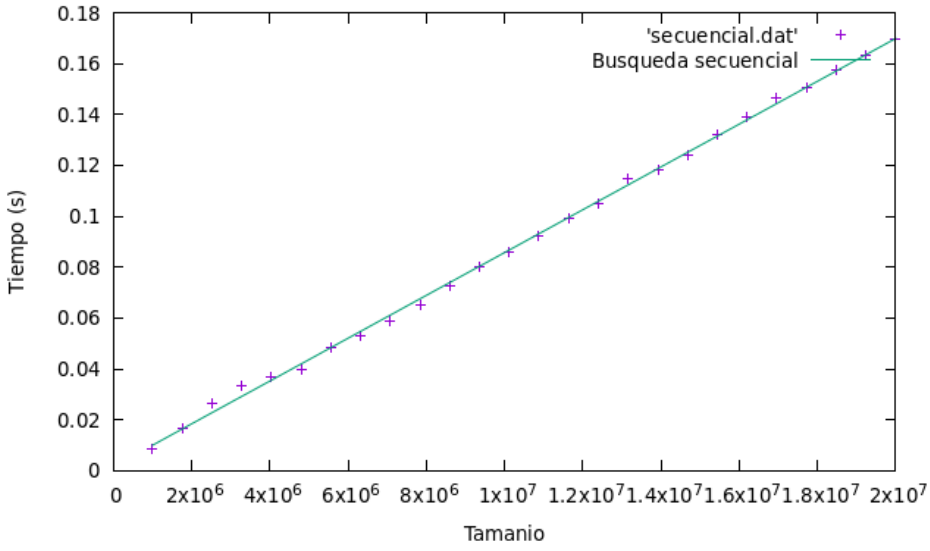


Figura 1: Gráfica con los tiempos de ejecución de la búsqueda a fuerza bruta

Y las constantes ocultas obtenidas son:

$T(n) = 8{,}41755 \cdot 10^{-9} n + 0{,}00153755$.

Y el coeficiente de determinación es:

Coef.determinación = 0.9992

Verificando así que nuestro análisis teórico es correcto.

2.1.2. Divide y Vencerás

La oportunidad para usar la técnica "Divide y Vencerás" ocurre en el momento en el que trabajamos con vectores ordenados (en nuestro caso de manera creciente). Esto nos permite usar un algoritmo que se basa en la técnica "Divide y Vencerás": la búsqueda binaria.

La búsqueda binaria consiste en ir comprobando los elementos del vector comenzado por su mitad. Al estar el vector ordenado, si el valor que hemos comprobado es mayor que la posición en la que se encuentra, podemos desechar la mitad superior de nuestro vector. En caso de ser menor que la posición en la que se encuentra,

podemos desechar la primera mitad del vector. Aplicando esto de manera recursiva, obtenemos un algoritmo de características muy buenas e interesantes. Pasemos a su análisis:

1. **Análisis Teórico**

```
int buscarBinaria(int *v, int inicio, int fin){
    if(fin >= inicio){ // O(1)
        int medio = inicio + (fin - inicio) / 2; // O(1)

        if(v[medio] == medio){ // O(1)
            return medio; // O(1)
        }

        if(v[medio] > medio){ // O(1)
            return buscarBinaria(v, inicio, medio - 1); // O(n/2)
        }

        //else
        return buscarBinaria(v, medio + 1, fin); // O(n/2)
    }

    return -1; // O(1)
}
```

Como podemos observar en los comentarios del código, todas las operaciones son $\mathcal{O}(1)$ y la llamada recursiva a la función es $\mathcal{O}\left(\frac{n}{2}\right)$. Una de las dos llamadas recursivas nunca se ejecuta, es decir, vamos descartando mitades del vector, por lo que la recurrencia que expresa el tiempo a partir del tamaño del vector viene dada por la siguiente expresión, de la cual vamos a deducir la eficiencia del algoritmo:

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

Para resolver esta recurrencia, tomamos $n = 2^k$, quedándose de la forma:

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

Cuyo polinomio característico es:

$$(x-1)^2$$

De donde se deduce que la solución de la recurrencia será de la forma $(c0 + c1 \cdot k) \cdot 1^k$, y al deshacer el cambio $n = 2^k$, obtenemos que:

$$T(n) = c0 + c1 \cdot \log_2(n)$$

Por lo que concluimos que:

$$T((n) \in \mathcal{O}(log_2(n))$$

2. **Análisis Empírico**

Tras ejecutar el algoritmo para 26 tamaños distintos; desde 1000000 hasta 20000000 dando saltos de 760000, hemos obtenido los siguientes resultados:

Divide y Vencerás sin repeticiones	
Elementos (n)	**Tiempo (s)**
1760000	0.000000402733
2520000	0.0000005118
3280000	0.000000472
4040000	0.000000538667
4800000	0.0000006558
5560000	0.0000006632
6320000	0.000000618467
7080000	0.0000005378
7840000	0.000000617267
8600000	0.000000618667
9360000	0.0000007254
10120000	0.000000638133
10880000	0.0000006072
11640000	0.00000071
12400000	0.000000569667
13160000	0.0000006822
13920000	0.000000631667
14680000	0.000000569333
15440000	0.000000697867
16200000	0.0000005758
16960000	0.00000069
17720000	0.000000623667
18480000	0.000000644133
19240000	0.0000007254
20000000	0.000000673533

Cuadro 2: Experiencia empírica de la búsqueda binaria

3. Análisis híbrido

El análisis híbrido nos permitirá comprobar que nuestro análisis teórico era correcto. Usando los datasets del anterior apartado, hemos usado gnuplot para graficar los puntos obtenidos junto con su función de ajuste. A continuación mostramos la gráfica con su respectiva función de ajuste:

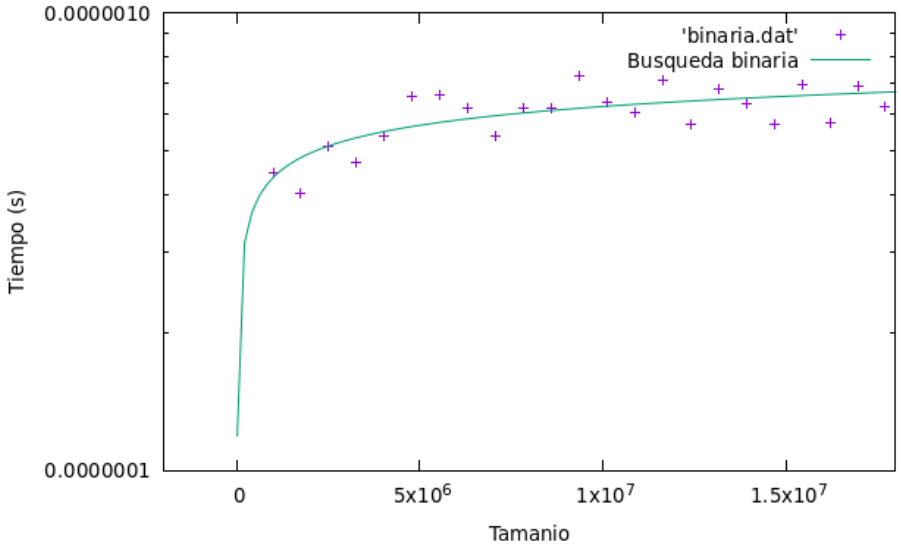


Figura 2: Gráfica con los tiempos de ejecución de la búsqueda binaria

Y las constantes ocultas son:

$T(n) = 5{,}63832 \cdot 10^{-8} \cdot \log_2(n) - 6{,}87177 \cdot 10^{-7}$.

Para terminar nuestro análisis de este algoritmo, terminaremos de confirmar que el ajuste logarítmico es el óptimo viendo el coeficiente de determinación que nos ha proporcionado gnuplot:

Coef.determinación = 0.999999954166

4. Comparación algoritmo de fuerza bruta vs Divide y Vencerás sin repeticiones

Como ya sabemos de la práctica anterior, el orden de eficiencia $\mathcal{O}(log_2(n))$ es mejor que $\mathcal{O}(n)$, por lo que nuestra implementación usando la técnica "Divide y Vencerás" mejora los tiempos de ejecución considerablemente. Para saber a partir de qué tamaño del vector nos renta usar más la búsqueda por fuerza bruta o la nueva implementación, hemos de calcular el **umbral**, que surge de igualar las expresiones halladas en el análisis híbrido de ambos algoritmos:

$$5{,}63832 \cdot 10^{-8} \cdot \log_2(n) - 6{,}87177 \cdot 10^{-7} = 8{,}41755 \cdot 10^{-9} n + 0{,}00153755$$

Sin embargo, no existe ningún natural tal que se cumpla esta igualdad, pues la función del algoritmo de fuerza bruta siempre va a estar por encima, luego el umbral es $n = 1$. Por último, mostramos la gráfica comparando las dos funciones de ajuste de ambos algoritmos:

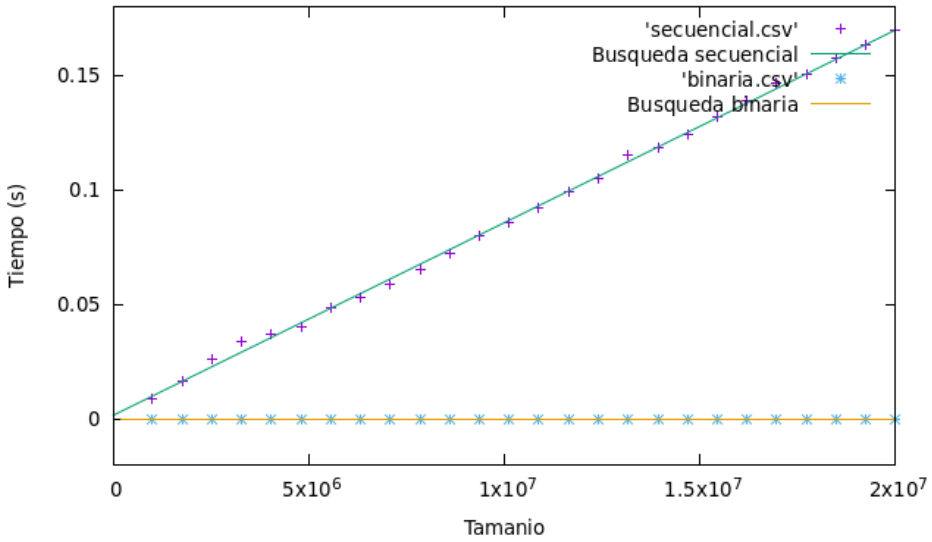


Figura 3: Gráfica comparativa: Fuerza Bruta vs DyV sin repeticiones

2.1.3. ¿Elementos repetidos?

Otra parte que se nos plantea en el ejercicio es la siguiente: ¿qué pasaría si en el elemento tuviésemos elementos repetidos? En este caso, la búsqueda binaria tal y como la programamos antes dejaría de ser efectiva. Aquí un ejemplo:

$$1 \quad 2 \quad 3 \quad 4 \quad 4 \quad 5 \quad 6 \quad 7$$

En este caso, el primer elemento que comprobaría nuestro algoritmo sería $v[3] = 4$. Entonces, desecharía los siguiente elementos a partir de $v[3]$. Pero vemos que en $v[4] = 4$, que es el elemento que deseamos. Para evitar este problema, hemos diseñado la siguiente solución:

```
int buscarBinaria(int v[], int inicio,int fin){
    int medio = (inicio + fin)/2; // O(1)
    int resultado = -1; // O(1)

    if(v[medio] == medio){ // O(1)
        return medio; // O(1)
    }
    else{
        if(inicio <= fin){ // O(1)
            resultado = buscarBinaria(v, inicio, medio - 1); // O(n/2)

            if(resultado == -1){
                resultado = buscarBinaria(v, medio + 1, fin); // O(n/2)
            }
        }
    }

    return resultado; // O(1)

}
```

1. **Análisis teórico**

Con este algoritmo, obtenemos la siguiente ecuación de recurrencia:

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

Resolviéndola mediante los métodos vistos en clase, tomamos $n = 2^k$ y obtenemos que el polinomio característico asociado a la recurrencia es:

$$(x-1)(x-2)$$

De donde se obtiene que el tiempo se expresa en función de k como:

$$T(2^k) = c0 + c1 * 2^k$$

Y deshaciendo el cambio obtenemos que $T(n) = c0 + c1 * n$, por lo que concluimos que:

$$T(n) \in \mathcal{O}(n)$$

Vemos que es el mismo orden de eficiencia que la búsqueda secuencial. Esto ocurre porque nuestro peor caso es en el que el elemento que buscamos está en $v[n-2]$. Puesto que vamos comprobando elementos siempre en las mitades inferiores hasta que encontramos lo deseado, para llegar a el elemento $v[n-2]$ tendríamos que hacer n iteraciones. Además, este algoritmo es incluso peor que la búsqueda secuencial, ya que en vectores muy grandes podemos llenar la pila de llamadas y hacer nuestro programa imposible de ejecutar. Esto también se ve en los datasets obtenidos en el análisis empírico y en la gráfica comparativa, que veremos a continuación en el análisis empírico e híbrido

2. **Análisis empírico**

Al igual que para el algoritmo de fuerza bruta y el anterior, hemos probado para 26 tamaños distintos (15 repeticiones por tamaño), desde 1000000 hasta 20000000, dando saltos de 760000, dando lugar a este dataset:

Divide y Vencerás con repeticiones	
Elementos (n)	**Tiempo (s)**
1760000	0.020179
2520000	0.0220417
3280000	0.0344659
4040000	0.0398963
4800000	0.0443348
5560000	0.0502432
6320000	0.0558109
7080000	0.0592223
7840000	0.0630519
8600000	0.0698851
9360000	0.0772074
10120000	0.0808893
10880000	0.0893751
11640000	0.0940162
12400000	0.0992901
13160000	0.103868
13920000	0.116623
14680000	0.118174
15440000	0.12476
16200000	0.131296
16960000	0.145325
17720000	0.162416
18480000	0.170681
19240000	0.177497
20000000	0.185571

Cuadro 3: Experiencia empírica de la búsqueda con repeticiones

3. **Análisis híbrido**

A partir del dataset anterior, hemos usado gnuplot para comprobar que la función de ajuste se corresponde con el orden de eficiencia hallado, corroborando así nuestro análisis. He aquí la gráfica:

Y las constantes ocultas son:

$T(n) = 8{,}71886 \cdot 10^{-9} \cdot n - 0{,}00140853$.

Para terminar nuestro análisis de este algoritmo, terminaremos de confirmar que el ajuste lineal es el óptimo viendo el coeficiente de determinación que nos ha proporcionado gnuplot:

Coef.determinación = 0.9941

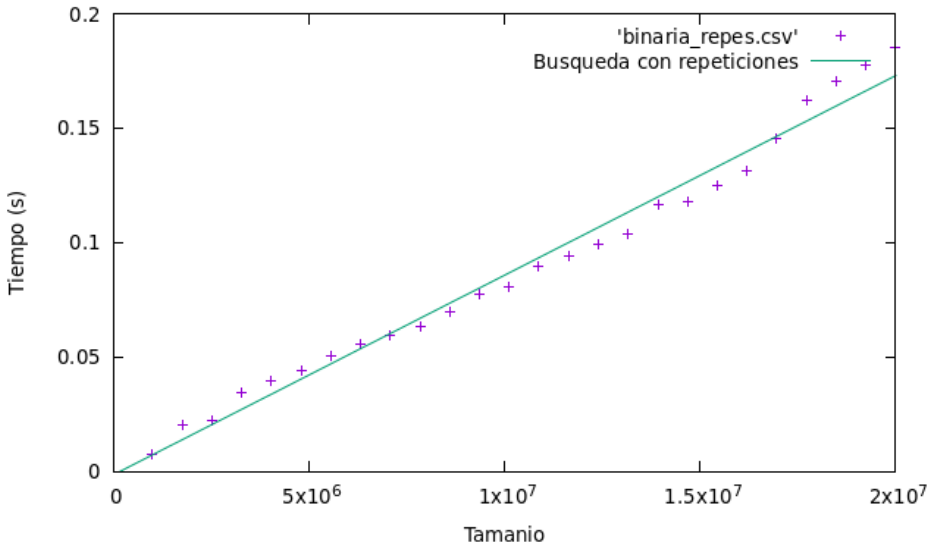


Figura 4: Gráfica con los tiempos de ejecución de la búsqueda con repeticiones

4. **Comparación algoritmo de fuerza bruta vs Divide y Vencerás con repeticiones**

Como hemos visto, tanto el algoritmo de fuerza bruta como el "Divide y Vencerás" para vectores ordenados con repeticiones presentan una eficiencia $\mathcal{O}(n)$. Además si observamos sus respectivas funciones de ajuste y las graficamos observamos que el algoritmo de fuerza bruta es más eficiente que el "Divide y Vencerás" implementado, pues la pendiente del de fuerza bruta es algo menor que la del otro algoritmo:

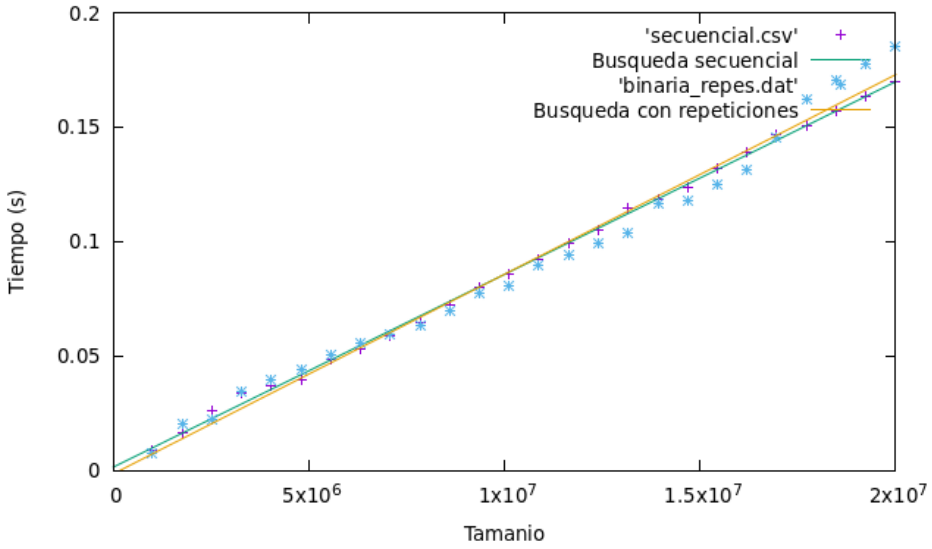


Figura 5: Gráfica comparativa: Fuerza Bruta vs DyV con repeticiones

Fuerza bruta $\longrightarrow T(n) = 8{,}41755 \cdot 10^{-9} n + 0{,}00153755$.

DyV con repeticiones $\longrightarrow T(n) = 8{,}71886 \cdot 10^{-9} \cdot n - 0{,}00140853$.

El estudio de este caso nos ha servido para obtener una de las conclusiones de nuestro trabajo, y es que la técnica "Divide y Vencerás" no siempre da resultado e incluso puede dar lugar a algoritmos que tarden más en dar la solución que lo que tarda el de fuerza bruta.

2.2. Ejercicio 2

El enunciado del problema es el siguiente: *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). Una posible alternativa consiste en, utilizando un algoritmo clásico, mezclar los dos primeros vectores, posteriormente mezclar*

el resultado con el tercero, y así sucesivamente.
¿Cuál sería el tiempo de ejecución del algoritmo?
Existe un algoritmo más eficiente?. Si es así diseñe, analice la eficiencia e implemente dicho algoritmo.
Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

2.2.1. Algoritmo de fuerza bruta

Como se comenta en el enunciado, el algortimo de fuerza bruta de este ejercicio es ir mezclando los dos primeros vectores, después mezclar el resultante con el tercero y así sucesivamente.

Para ello la forma simple definir un array del tamaño final que sabemos que será k·n. Inicializamos el primer vector en las priemras n posiciones del array y vamos iterando en dicho array cada elemento de cada nuevo vector e ir comprobando si es menor que alguno de los ya insertados previamente. Si el elemento es menor se realiza una traslación hacia la derecha desde la posición en la que ha resultado ser menor hasta el último elemento insertado en el array. Si no es menor, entonces se insertará en la siguiente posición al útlimo insertado. De esta forma se va expandiendo hasta completar las k*n posiciones del array.

1. **Análisis Teórico**

```
void mergeKArrays(int nElementos, int **arr, int nVectores, int * &v_resultante)
{

    int k = 0;
    bool encontrado;
    for(int i = 0; i < nElementos; i++){
      v_resultante[i] = arr[0][i];
    }

    for(int i = 1; i < nVectores; i++){          //O(k)
      for(int j = 0; j < nElementos; j++){       //O(n)
        encontrado = false;
        k = 0;
        while(k < nElementos * i + j && !encontrado){    //O(kn + n)
          if(v_resultante[k] > arr[i][j])
            encontrado = true;
          else
            k++;
        }

        if(encontrado){
          for(int l = nElementos*i+j-1; l >= k; l--){    //O(kn)
            v_resultante[l+1] = v_resultante[l];
          }
          v_resultante[k] = arr[i][j];
        }
        else{
          v_resultante[nElementos * i + j ] = arr[i][j];
        }
      }
    }

}
```

Tal y como se ven en los comentarios del código, el bucle principal está formado por dos bucles anidados cuyos tamaños son $\mathcal{O}(k)$ y $\mathcal{O}(n)$ respectivamente y dentro de ellos tenemos dos bucles en paralelo de tamaños $\mathcal{O}(kn+n)$ y $\mathcal{O}(kn)$. Luego la eficiencia teórica del algoritmo, según lo visto en clase, es evidente que es:

$$T((n) \in \mathcal{O}(k^2n^2)$$

2. **Análisis empírico** Tras ejecutar el algoritmo para 26 tamaños distintos; desde 1000000 hasta 20000000 dando saltos de 760000, hemos obtenido los siguientes resultados:

2.2.2. Divide y Vencerás

El algoritmo más eficiente para realizar este ejercicio es de la forma divide y vencerás. Lo primero que hacemos al igual que antes es definir una matriz de k filas y n columnas y un un array del tamaño final que sabemos que será k·n. Ahora lo que hacemos es ir creando en cada caso 2 arrays auxiliares de la matriz de tamaño $\frac{k}{2}$ y llamar a la función de nuevo para cada uno de arrays, actualizando en cada caso el valor de k a la mitad. De

esta forma tenemos una función recursiva que divide el array de entrada en 2 arrays auxiliares, se llama a sí misma en cada array y finalmente se mezclan los dos arrays resultantes. El caso base de la función recursiva es que k sea 1 y entonces se copia el array auxiliar en el array resultado de la función.

1. **Análisis Teórico**

```
void mergeKArrays(int n, int **arr, int n1,int n2, int * &array_resultante)
{
    //si solo hay un array
    if(n1==n2){                         //O(n)
        for(int i=0; i < n; i++)
        array_resultante[i]=arr[n1][i];
    }
    else{

      int nVect = n2-n1+1;
      int mitad = (n2+n1)/2;

      //Dimensiones arrays auxiliares
      int tam2 = nVect/2;
      int tam1 = nVect - tam2;

        //Arrays resultantes

        int *array1 = nullptr;
        reservarArray(n*(tam1), array1);
        int *array2 = nullptr;
        reservarArray(n*(tam2), array2);

        //divide el array en dos mitades
        mergeKArrays(n, arr,n1,mitad,array1);
        mergeKArrays(n, arr,mitad+1,n2,array2);

        //mezcla el array resultante
        merge2Arrays(array1,array2,n*tam1,n*tam2,array_resultante);      //O(kn)


        if (array1 != nullptr){
            delete[] array1;
        }
        if(array2 != nullptr){
            delete[] array2;
        }
    }

}
```

Tal como se ha comentado, es una función recursiva en la que como se ve en la imagen, el caso base es $\mathcal{O}(n)$ y la función utilizada para mezclar dos vectores es $\mathcal{O}(kn)$. La ecuación recurrente asociada a dicho algorirmo será entonces:

$$T(n) = \begin{cases} n & si \quad n = 1 \\ \\ 2T(\frac{n}{2}) + c2k \cdot n & si \quad n > 1, n = 2^z \end{cases}$$

Tras resolver la ecuación recurrente como hemos visto en clase (por expansión) obtenemos que:

$$T(n) = 2^z T(1) + z2$$

y como T(1) = n y n = $2^z entonces tenemos finalmente que$

$$T(n) = n^2 + c2 \cdot \log_2(n) \cdot k \cdot n$$

y podemos deducir entonces que si $M = max\left\{n^2, \log_2(n) \cdot k \cdot n\right\}$ nuestra función es

$$T(n) \in \mathcal{O}(M)$$