## horizontal line



Práctica 2: Problema de la comparación de las preferencias

2019

**─**

Realizado por:

Miguel Ángel Campos Cubillas

Alejandro Pinel Martínez

Guillermo Palomino Sánchez

Pablo Lombardero Ros

Nikita Stetskiy

# 

[**Problema: Comparación de preferencias**](#_nvhedu9rldrd) **2**

[**Objetivo**](#_3at9u9s4e0vp) **2**

[**Esquema de la solución**](#_4p7xi5bvhxdr) **2**

[**Algoritmo básico de cálculo de inversiones de un vector**](#_yyrhu7ml5bea) **2**

[**Algoritmo Divide y Vencerás de cálculo de inversiones en un vector**](#_iahla24jqe7l) **6**

[**Comparación entre Algoritmo Básico y Algoritmo Divide y Vencerás**](#_tgulstc614xq) **14**

[**Conclusión**](#_k9oo0skrqwd6) **15**

# 

# 

# 

# Problema: Comparación de preferencias

El problema que vamos a resolver tiene el siguiente enunciado:

|  |
| --- |
| *Muchos sitios web intentan comparar las preferencias de dos usuarios para realizar suge-*  *rencias a partir de las preferencias de usuarios con gustos similares a los nuestros. Dado un*  *ranking de n productos (p.ej. pelı́culas) mediante el cual los usuarios indicamos nuestras pre-*  *ferencias, un algoritmo puede medir la similitud de nuestras preferencias contando el número*  *de inversiones: dos productos i y j están “invertidos” en las preferencias de A y B si el usuario*  *A prefiere el producto i antes que el j, mientras que el usuario B prefiere el producto j antes*  *que el i. Esto es, cuantas menos inversiones existan entre dos rankings, más similares serán las*  *preferencias de los usuarios representados por esos rankings.*  *Por simplicidad podemos suponer que los productos se pueden identificar mediante enteros*  *1, . . . , n, y que uno de los rankings siempre es 1, . . . , n (si no fuese ası́ bastarı́a reenumerarlos) y*  *el otro es a 1 , a 2 , . . . , a n , de forma que dos productos i y j están invertidos si i < j pero a i > a j .De esta forma nuestra representación del problema será un vector de enteros v de tamaño n,*  *de forma que v[i] = a i , i = 1, . . . , n.*  *El objetivo es diseñar, analizar la eficiencia e implementar un algoritmo “divide y vencerás”*  *para medir la similitud entre dos rankings. Compararlo con el algoritmo obvio. Realizar también*  *un estudio empı́rico e hı́brido de la eficiencia de ambos algoritmos.* |

# Objetivo

El objetivo, resumido, es, dado un vector de enteros que contenga una permutación de los números naturales de 1 a n, contar el número de pares de enteros i y j que cumplan la condición i < j y v[i] > v[j].

# Esquema de la solución

Para resolver este problema, se recurrirá a un algoritmo básico: sencillo pero ineficiente, de O(n²); y a un algoritmo que utiliza la técnica divide y vencerás, más complejo y eficiente.

Se realizará un estudio teórico, empírico e híbrido de ambos algoritmos y se compararán.

# Algoritmo básico de cálculo de inversiones de un vector

El algoritmo básico que se ha utilizado para contrastar con el algoritmo de Divide y Vencerás es el contenido en el fichero fuente *algoritmo\_normal.cpp*. Se ha partido del archivo para generar el vector de ordenación ofrecido en DECSAI y se ha añadido lo necesario para que se puedan contabilizar las inversiones. Concretamente, la siguiente función es la encargada de todo el proceso:

|  |
| --- |
| int ranking(int v[], int tam){  int cont = 0;    **for** (int i=0; i<tam; i++){  **for** (int j=i+1; j<tam; j++){  **if**(v[i] > v[j])  cont ++;  }  }  **return** cont;  } |

Como se indica en el guión que uno de los dos vectores de preferencias ya está ordenado, basta con trabajar con el otro. Así, el bucle externo se encarga de ir iterando por cada uno de los elementos del vector, mientras que el interno recorre los elementos restantes hasta el final del array. En caso de que el elemento seleccionado por el primer bucle sea mayor que el recorrido por el bucle interno, se produce una inversión y, por consiguiente, se incrementa el contador.

Una vez se tiene aclarado el comportamiento de dicho algoritmo, se pasará a su análisis de eficiencia.

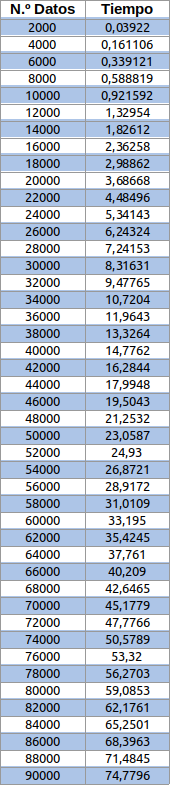
***Eficiencia Teórica***

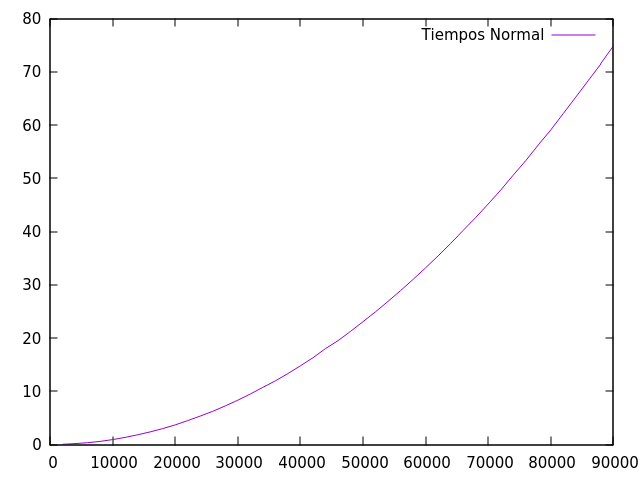
En cuanto a la **eficiencia teórica**, un primer vistazo permite observar que se trata de un algoritmo de eficiencia **O(n²)**, pues tiene un bucle anidado dentro de otro y los procesos que se realizan en él consumen tiempo constante.

***Eficiencia Empírica***

Pasando a la **eficiencia empírica**, se utiliza un script para ejecutar sucesivas veces el programa y para tamaños de entrada cada vez más grandes (dicho script se encuentra adjuntado a la memoria).

Los resultados son los siguientes:

Se analizan estos tiempos, como de costumbre, con la herramienta gnuplot, obteniendo la siguiente gráfica de tiempos de ejecución (eje Y) con respecto al tamaño del vector de preferencias (eje X):



Para profundizar, también realizamos el análisis híbrido de la eficiencia de este algoritmo. Como sabemos que es O(n²), partimos de la función f(x) = a0\*x\*x + a1\*x+ a2 y la ajustamos por medio de las tres constantes ocultas a0, a1 y a3 a la función empírica. De nuevo, lo hacemos por medio de gnuplot de forma idéntica a como se realizó en la práctica anterior, obteniendo los siguientes valores para las constantes:

Final set of parameters Asymptotic Standard Error

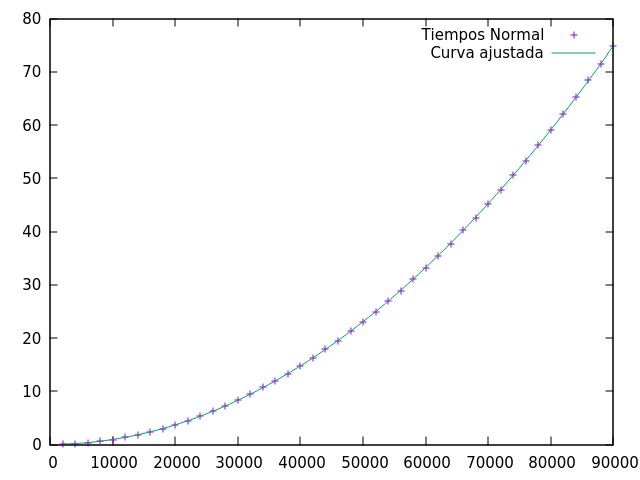
======================= ==========================

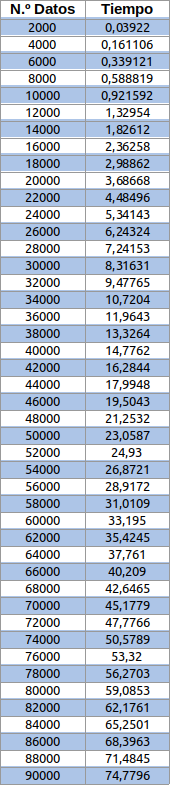
a0 = 9.25653e-09 +/- 1.106e-11 (0.1195%)

a1 = -2.12961e-06 +/- 1.05e-06 (49.29%)

a2 = 0.0325934 +/- 0.02094 (64.23%)

También representamos ambas funciones, empírica y teórica ajustada, en una misma gráfica:



La coincidencia precisa entre ambas funciones nos permite, además, cerciorarnos de la corrección del análisis teórico. 

# Algoritmo Divide y Vencerás de cálculo de inversiones en un vector

Divide y Vencerás es una técnica para diseñar algoritmos que consiste en descomponer el caso del problema que se quiere resolver en un número de subproblemas menores del mismo problema, resolviendo sucesiva e independientemente cada uno de ellos mediante un algoritmo que se supone que existe, al que en lo sucesivo se referirá como “básico”, y después combinando las soluciones así obtenidas de modo que se obtenga la solución del problema inicial.

Para aprovecharse de esta técnica, se utilizará una característica de nuestro problema: si tienes un segmento del vector ordenado de forma ascendente y, a continuación, un número menor a ese segmento, el número de inversiones provocadas por ese elemento es igual al número de elementos de ese segmento ordenado. En otras palabras, si se ordena el vector, el número de posiciones que hay que desplazar un elemento a la izquierda, es el número de inversiones que provoca ese elemento.

Por ejemplo, si se tienen estos datos:

|  |
| --- |
| 2 3 4 5 6 7 1 |

Se puede observar que el “1” final es menor que todo el segmento anterior, que se encuentra ordenado de forma ascendente.

Mediante el algoritmo básico, habría que comprobar todos los elementos con todos los siguientes. Sin embargo, si se sabe que todo el segmento está ordenado, basta con compararlo con el primer elemento y, cómo es menor que este, el número de inversiones es 6, la cantidad de elementos del segmento ordenado que estaban delante suyo.

Generalizando este razonamiento, se puede calcular el número de inversiones mientras ordenamos el vector si se aprovecha para ello segmentos del vector ordenados.

El algoritmo utilizado será parecido al mergesort, modificando el algoritmo de mezcla para que, cuando un elemento de la mitad derecha del vector pase por delante de un número “x” de elementos del vector de la izquierda, el número de inversiones se incremente en “x”.

Dicho algoritmo se compone de dos funciones, la primera de ellas se encarga de dividir el vector en mitades de forma recursiva, mientras que la segunda realiza la ordenación del vector y el cálculo de las inversiones.

Antes de ver el código, sin embargo, se analizará el pseudocódigo de dicho algoritmo, ya que al ser una visión más general sirve muy bien de base para luego comprender la implementación utilizada.

***Pseudocódigo***

|  |
| --- |
| Procedure ranking (vector, tamanio)  Begin  si tamanio <= 1  devolver 0  si no  divide\_vector\_en\_dos  inversiones = ranking(primera\_mitad\_vector, tamanio/2)  inversiones += ranking(segunda\_mitad\_vector, tamanio - tamanio/2)  inversiones += unir(vector, tamanio/2, tamanio)  End |

|  |
| --- |
| Procedure unir (vector, mitad, final)  Begin  inversiones = 0  izq = 0  der = mitad  var temp[final]  indice = 0  mientras izq < mitad y der < final  si der == mitad o vector[izq] <= vector[der]  temp[indice] = vector[izq]  incrementa izq  si izq == mitad  temp[indice] = vector[der]  incrementa der  si vector[der] < vector[izq]  temp[ìndice] = vector[der]  inversiones += mitad - izq  incrementa der  incrementa i  End |

En el pseudocódigo anterior se puede observar como la idea es ir dividiendo recursivamente el vector en dos partes iguales hasta llegar al caso base, el vector de un componente, que devuelve 0 inversiones. Una vez que se tiene las dos mitades del vector se procede a su unión, proceso por el que se van ordenando con un sistema de mezcla, con la única peculiaridad de que, cuando un elemento de la mitad derecha, se incrementarán las inversiones por cada elemento de la izquierda restante.

Una vez se ha analizado el algoritmo en pseudocódigo, se puede pasar a analizar su implementación en C++.

***Implementación***

En el caso de la implementación en C++ lo que se puede observar es que el algoritmo se divide en dos funciones, llamadas en este caso *ranking* y *unir*, donde la primera se encarga de dividir el vector en mitades de forma recursiva y de calcular el número total de inversiones, y la segunda compara los elementos de dichas mitades y calcula el número de inversiones en ese intervalo.

|  |
| --- |
| */\**  *Esta función une las dos partes del vector, ordenadas individualmente y calcula*  *el número de inversiones que se producen entre las dos partes.*  *\*/*  int unir(int \* vector, int mitad, int final, int \* temp) {  int izq = 0, der = mitad, i = 0;  int inversiones = 0;  **while** (izq < mitad || der < final) {  *//Si se ha llegado al final del vector derecho, o el elemento de la izquierda*  *//es menor, añadimos el elemento de la izquierda*  **if** (der == final || (izq < mitad && vector[izq] <= vector[der])) {  temp[i] = vector[izq];  i++;  izq++;  }  *//Si se ha llegado al final del vector izquierdo, o el elemento de la derecha*  *//es menor, añadimos el elemento de la derecha.*  **else** {  temp[i] r= vector[der];  i++;  der++;  *//Calculamos el número de inversiones (el elemento recién insertado con*  *// todos los elementos restantes en la izquierda)*  inversiones += mitad - izq;  }  }  **for** (int j = 0; j < final; j++)  vector[j] = temp[j];  **return** inversiones;  } |

|  |
| --- |
| */\**  *Función que calcula las inversiones de un vector. El vector es ordenado en el*  *proceso. Argumentos: puntero al vector, números de componentes de este y un*  *puntero a un vector auxiliar que se utilizará en el proceso. El vector auxiliar*  *se proporciona para no incluir en el algoritmo los costes de sobrecarga.*  *\*/*  int ranking(int \* vector, int n, int \* temp) {  *//Caso base, un elemento, 0 inversiones*  int inversiones = 0;  *//El resto se calcula como la suma de las inversiones de la derecha con las de*  *// la izquierda y con las generadas en la unión.*  **if** (n > 1) {  int mitad = n/2;  inversiones += ranking(vector, mitad, temp);  inversiones += ranking(vector + mitad, n - mitad, temp + mitad);  inversiones += unir(vector, mitad, n, temp);  }  **return** inversiones;  } |

***Eficiencia Teórica***

Primero se va a calcular la eficiencia teórica del algoritmo.

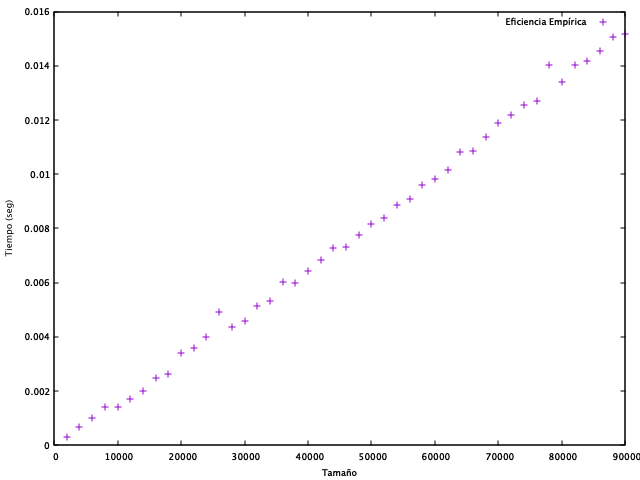
La función unir se puede observar que es de orden lineal, pues itera el bucle n veces. La función ranking es recursiva, luego se recurre a la fórmula maestra para conocer su orden de eficiencia. Realiza dos llamadas recursivas con un vector de la mitad de tamaño y luego llama a la función unir, de orden lineal, luego su eficiencia será:

**T(n)= 2\*T(n/2) + n**

**T(n)= O(n\*log(n))**

El algoritmo es, por tanto, de orden **nlog(n)**

***Eficiencia Empírica***

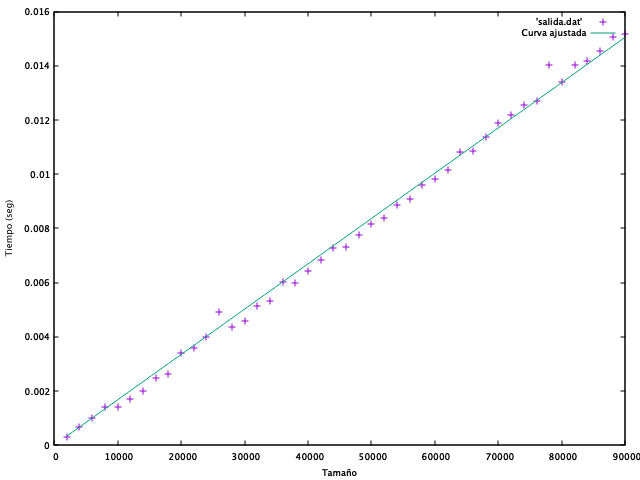
****



***Eficiencia Híbrida***

Gracias al cálculo de la eficiencia teórica, se ha obtenido que la eficiencia de este algoritmo es **O(nlog(n))**, partiendo de la función f(x) = a0\*x, se ajusta por medio de la constante a0 a la función empírica.

Como se puede observar en la gráfica que hay a continuación, salvo por alguna excepción, los datos obtenidos de forma empírica se ajustan muy bien a la función obtenida mediante el cálculo de la eficiencia teórica, por lo que se puede asegurar que el orden de este algoritmo es **O(nlog(n))**, tal y como se había calculado.



Asimismo, si se observa el error del ajuste, se puede ver que efectivamente estamos tratando con un algoritmo de orden **O(nlog(n))**

Final set of parameters Asymptotic Standard Error

======================= ==========================

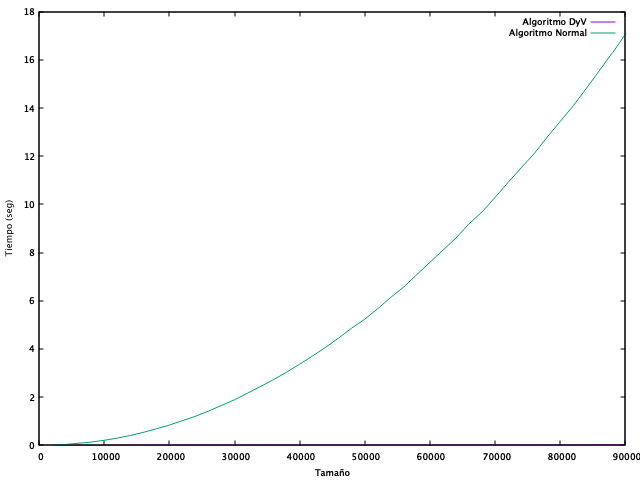
a0 = 1.67334e-07 +/- 7.819e-10 (0.4673%)

# Comparación entre Algoritmo Básico y Algoritmo Divide y Vencerás

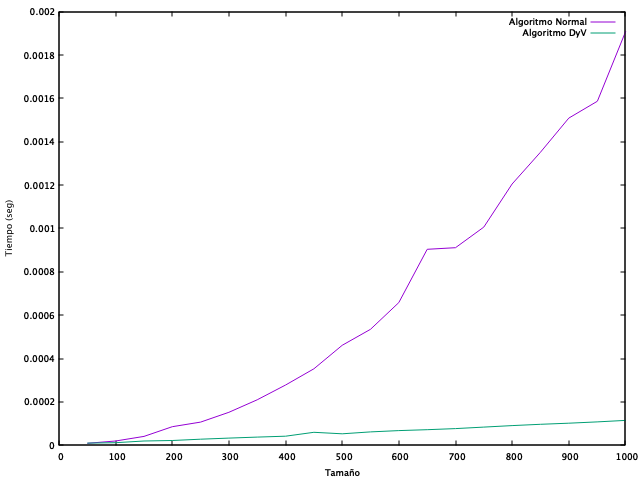
Finalmente, tras haber estudiado ambos algoritmos por separado, se procederá a compararlos el uno con el otro, para así poder llegar a una conclusión satisfactoria.

Habiendo analizado el comportamiento de ambos algoritmos para valores de 2000 a 90000, los resultados obtenidos son muy reveladores. Viendo y comparando el orden de ambos algoritmos ya se podía llegar a la conclusión de que la versión DyV iba a ser mucho más eficiente que la versión básica, pero no es si no hasta que se observa una comparación directa de los datos obtenidos empíricamente, que este hecho se hace mucho más evidente.

Como se puede observar en la gráfica obtenida la diferencia es extremadamente grande, hasta el punto de que la función obtenida con el algoritmo DyV se hace casi imperceptible.

****

No es hasta que se reduce el rango de valores, que se puede apreciar el crecimiento de ambas funciones de forma clara.

****

Estos hechos no hacen más que poner en entredicho el uso del algoritmo básico frente a su versión Divide y Vencerás, y a su vez ponen una reflexión interesante sobre la mesa, ¿merece la pena el tiempo extra invertido en el estudio del problema para poder hallar un algoritmo más eficiente?

# Conclusión

El estudio de este problema ha sido bastante interesante, ya que debido a nuestra corta experiencia como programadores, la mayoría de las veces preferimos hacer un trabajo rápido y que simplemente cumpla con su función en vez de preocuparnos por el tiempo de ejecución de dicho programa. Esto ahora mismo no es un problema, porque los trabajos que realizamos la eficiencia no suele ser crítica. Esto en el mundo real, no es así.

Y es por esto que a la pregunta, ¿merece la pena el tiempo extra invertido en el estudio del problema para poder hallar un algoritmo más eficiente? Nosotros hemos llegado a la conclusión de que sí que merece la pena. Este algoritmo nos ha permitido pasar de un orden cuadrático a uno nlog(n), lo cual es un salto inmenso en términos de eficiencia. Esto ha conllevado, eso sí, el desarrollo de un algoritmo utilizando una técnica más compleja, divide y vencerás, que requiere pensar de manera diferente a la acostumbrada, pero que nos permite mejorar los algoritmos de esta forma.

En definitiva, la técnica divide y vencerás puede ser difícil de aplicar, pues requiere manejar con soltura recursividad y otros conceptos no intuitivos, pero es imprescindible para el desarrollo de algoritmos eficientes.