Algorítmica - Practica 2: Divide y Vencerás

A. Herrera, A. Moya, I. Sevillano, J.L. Suarez

14 de abril de 2015

Contents

1 Organización de la práctica

La práctica 2 trata sobre el desarrollo de algoritmos siguiendo el paradigma divide y vencerás. El problema que nuestro grupo debe resolver es el 6. Hemos desarrollado además varios algoritmos para el problema opcional que se exponen en su correspondiente apartado.

Para cada problema se sigue la siguiente estructura:

- Enunciado del problema
- Resolución teórica del problema (con una subsección por algoritmo)
- Análisis empírico. Análisis de la eficiencia híbrida

En este último apartado se proporcionan gráficas con los resultados de los algoritmos y un análisis de la eficiencia híbrida para los mismos.

Los algoritmos se han ejecutado sobre un ordenador con las siguientes características:

- **Marca:** Toshiba
- **RAM:** 8 GB
- **Procesador:** Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

El código, los resultados de las ejecuciones, las gráficas y los pdf asociados se puede encontrar en GitHub.

2 Problema 6

2.1 Enunciado del problema

Sea un vector v de números con n componentes, todas distintas, de forma que existe un índice p (que no es ni el primero ni el último) tal que a la izquierda de p los números están ordenados de forma creciente y a la derecha de p están ordenados de forma decreciente, es decir:

$$\forall i, j \leq p \text{ con } i < j \Rightarrow v[i] < v[j] \text{ y } \forall i, j \geq p \text{ con } i < j \Rightarrow v[i] > v[j] \quad (1)$$

Lo que implica que el máximo se encuentra en p. Diseñe un algoritmo "divide y vencerás" que permita determinar p. ¿Cuál es la complejidad del algoritmo?. Compárelo con el algoritmo "trivial" para realizar esta tarea. Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

2.2 Resolución teórica del problema

Un ejemplo de vector unimodal es el dado en la Imagen 1, en la cual se señala el buscado elemento p. Se harán referencias a la imagen a lo largo del texto.

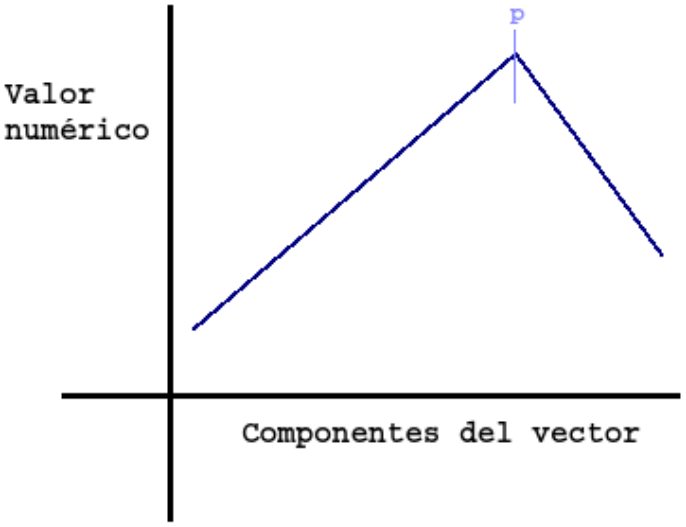

Imagen 1. Representación de un vector unimodal.

Daremos dos soluciones al problema, una trivial y otra basada en divide y vencerás. Como es de esperar, la primera tendrá un orden de eficiencia lineal mientras que con la segunda conseguiremos alcanzar el deseado orden $\log n$.

En primer lugar, nótese que dada la forma del vector, caracterizada por (1), el elemento p es el máximo del vector. Esto implica que p verifica (2):

$$v[p-1] < v[p] > v[p+1] \quad (2)$$

Y, además, es el único elemento del vector que verifica (2) pues en caso caso contrario no se mantendría el invariante (1) del vector. Es por tanto el único máximo local. Utilizaremos este hecho para el desarrollo de ambos algoritmos.

2.2.1 Solución trivial

La idea es sencilla: recorrer el vector hasta encontrar el punto p. Recordemos que la propiedad (2) caracteriza al punto p. Por tanto, en nuestro algoritmo trivial basta recorrer todas las componentes $2, 3, ..., n-1$ y encontrar la primera componente que verifica (2), que será p por unicidad.

Dado que nuestro recorrido es lineal de izquierda a derecha podemos obviar la primera comparación $v[p-1] < v[p]$ pues si no hemos encontrado todavía p es porque $v[p-2] < v[p-1] < v[p-1]$.

2.2.1.1 Pseudocódigo.

Se proporciona el pseudocódigo a continuación:

```
# Pseudocódigo del algoritmo trivial.
# v es el vector unimodal con n componentes.
for i in range(1, n-1):
    if v[i] > v[i+1]:
        p = i
        break
```

En el caso de la Imagen 1, el algoritmo recorrería el vector de izquierda a derecha hasta encontrar el elemento p, donde la pendiente se vuelve negativa.

2.2.1.2 Ejemplo.

Veamos un ejemplo del algoritmo. Consideramos el vector:

$$\text{v} = [1\ 3\ 5\ 7\ 8\ 6\ 4]$$

Empezamos el recorrido en la componente de índice 1. Esta es menor que $v[2] = 5$ luego continuamos la búsqueda. Repetimos este hecho hasta que llegamos a la componente $v[4] = 8$. Tenemos que $v[5] = 6 < v[4] = 8$ y por ello se tiene que $p = 4$.

2.2.2 Solución mediante divide y vencerás

Queremos obtener una solución con un orden de eficiencia $\theta(\log n)$. Para ello deberímos librarnos de la búsqueda lineal. Un problema similar y ampliamente conocido es el de búsqueda en un vector. La búsqueda lineal sirve en vectores desordenados. Pero en el caso de ser ordenados, podemos hacerlo mucho mejor con una búsqueda binaria, que sigue las directrices del paradigma de diseño de algoritmos divide y vencerás. Nos inspiraremos en la búsqueda binaria para el diseño de nuestro algoritmo.

En primer lugar, comprobamos si el elemento en la posición mitad del vector verifica (2). En tal caso, este elemento debe ser p, devolviendo su índice como resultado del algoritmo. En el caso de no ser p recurrimos a divide y vencerás. Para ello, tal y como sucede en la búsqueda binaria, obviaremos uno de los dos vectores mitades (izquierda y derecha) llamando a nuestro algoritmo recursivo sobre el otro subvector.

La cuestión es la siguiente: ¿sobre qué subvector aplicar la recursividad y por qué?.

Consideremos el vector unimodal de la Imagen 1. Los elementos a la izquierda de p están ordenados de menor a mayor y los que se encuentran a su derecha de mayor a menor. Encontramos la componente mitad del vector, llamémosla $m1$. El vector se encuentra dividido tal y como muestra la Imagen 2.

Es claro que la componente p se queda en el subvector derecha. Podemos ahora abstraer la razón de este hecho. Esta es:

$$v[m1-1] < v[m1] < v[m1+1]$$

Como $v[m1] < v[p]$ y nos encontramos en el subvector creciente, p debe estar a la derecha de $m1$ tal y como sucede en la imagen.

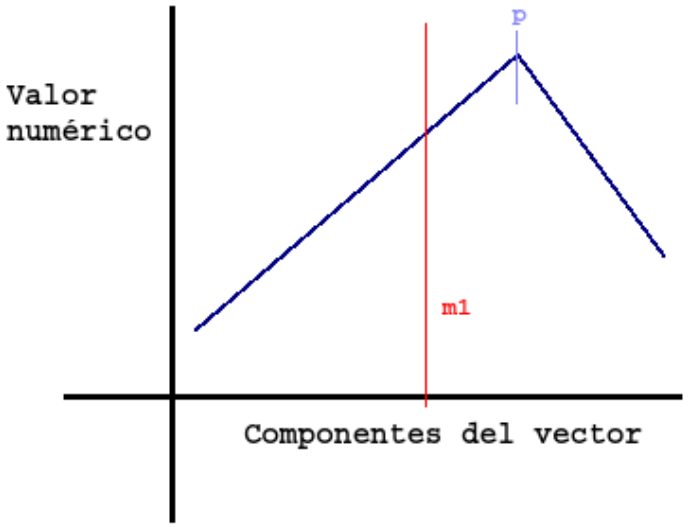

Imagen 2. Componente mitad del vector unimodal.

Utilizamos ahora recursividad sobre las componentes $\{m1+1, ..., n-1\}$. La Imagen 3 muestra esta situación. Obtenemos la componente mitad de este nuevo subvector, denotémosla $m2$, que tampoco verifica (2) y por ello no es p. Pero en este caso se tiene que $m2$ cumple:

$$v[m2-1] > v[m2] > v[m2+1]$$

Por tanto, estamos situados en la zona decreciente del vector v y por ello p se encuentra a la izquierda de $m2$ ($v[p] > v[m2]$). El subvector sobre el que se volvería a aplicar la recursividad se ha resaltado en gris. Se debe repetir el proceso hasta dar con el elemento p.

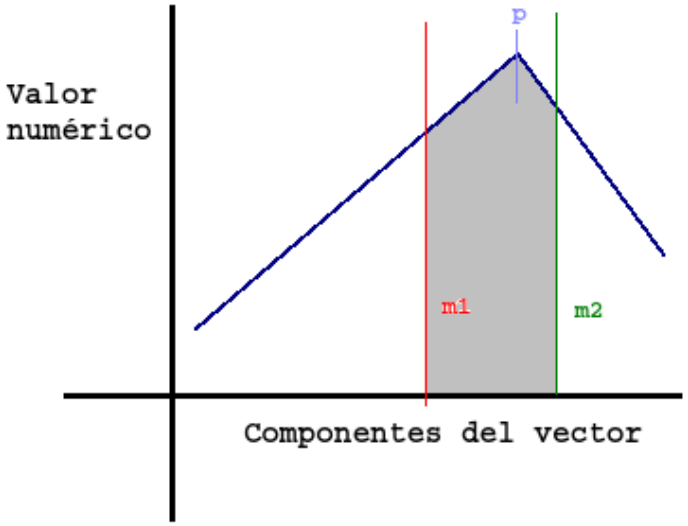

Imagen 3. Segunda iteración sobre el vector unimodal.

2.2.2.1 Pseudocódigo. Formalizando lo anterior, podemos dar el siguiente pseudocódigo:

```
# Algoritmo recursivo que resuelve el problema en O(log n).
# El vector dado como parámetro se presupone unimodal.
# Parametros:
#  - v      : Vector sobre el que se realiza la búsqueda.
#  - inicio : Posición de inicio del subvector a considerar.
#  - final  : Posición siguiente a la última del subvector.
def obtenerP(v, inicio, final):
    mitad = (inicio + final) / 2
    if mitad > inicio:
        if v[mitad-1] < v[mitad] and v[mitad] > v[mitad+1]:
            return mitad
        else if v[mitad-1] < v[mitad]:
            return obtenerP(v, mitad+1, final)
        else:
            return obtenerP(v, inicio, mitad)
    else:
        return inicio
```

2.2.2.2 Ejemplo. Veamos un ejemplo del funcionamiento del algoritmo. Consideramos el mismo vector que para el algoritmo lineal:

$$\text{v} = [1\ 3\ 5\ 7\ 8\ 6\ 4]$$

Este vector consta de 7 componentes. Inicialmente, `inicio = 0` y `final = 7`. Se tiene que la componente mitad tiene índice $3 = (0 + 7)\ /\ 3$. Sin embargo, $v[2] = 5 < v[3] = 7 < v[4] = 8$. Debemos pues aplicar la recursividad en el subvector derecha, `return obtenerP(v, mitad+1, final)`.

$$\text{v} = [-\ -\ -\ -\ 8\ 6\ 4]$$

En este caso, la componente mitad tiene índice 5 y valor 6. Verifica $v[4] = 8 < v[5] = 6 < v[7] = 4$. En este caso se aplica la recursividad sobre el vector izquierda:

$$\text{v} = [-\ -\ -\ -\ 8\ -\ -]$$

Este contiene un único elemento, el 8, cuyo índice se devuelve como p.

Si comparamos el ejemplo con el del algoritmo lineal, aunque sea más antinatural el algoritmo recursivo realiza menos iteraciones (solo 3) en comparación con las 4 realizadas por el lineal. En problemas grandes la diferencia es incluso mayor como se verá en el siguiente apartado.

2.2.2.3 Eficiencia teórica. En cada llamada recursiva del algoritmo se hace un trabajo constante para calcular el punto medio y decidir en cual de los tres posibles casos estamos. En el peor de ellos, nunca se encuentra p y siempre se debe hacer una llamada recursiva a un vector de tamaño mitad. Si denotamos $T(n)$ a función con la eficiencia del algoritmo, en el peor caso esta verifica:

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

Tenemos una ecuación en diferencias lineal de fácil solución. Tomando $n = 2^k$:

$$T(n) = T(2^k) = T(2^{k-1}) + \theta(1) = \ldots = T(1) + (k-1)\theta(1) \in \theta(k) = \theta(\log_2 n)$$

Efectivamente, como ya se había adelantado, el algoritmo en el peor caso es $\theta(\log_2 n)$.

2.3 Análisis empírico. Análisis de la eficiencia híbrida

Para analizar de forma empírica el comportamiento de ambos algoritmos hemos recurrido al pograma `generar_unimodal.cpp` proporcionado para la práctica que genera vectores unimodales, es decir, verifican (1), de un tamaño dado como parámetro. Para su uso hemos desarrollado un script `generar_test_cases.sh` que compila y utiliza el programa anterior para crear 19 vectores unimodales cuyo tamaño varía de 50000 en 50000 desde 100000 hasta 1000000.

Mediante otro script llamado `comparar_algoritmos.sh` se ejecutan ambos algoritmos sobre todos los vectores generados, obteniendo el tiempo de ejecución para cada uno de ellos. Los resultados en un archivo para cada algoritmo compuesto de una columna con el tamaño de los vectores y otra con el tiempo de ejecución. Posteriormente, el script llama a `plot_resultados.sh`, que utiliza GNU-PLOT para generar la Imagen 4.

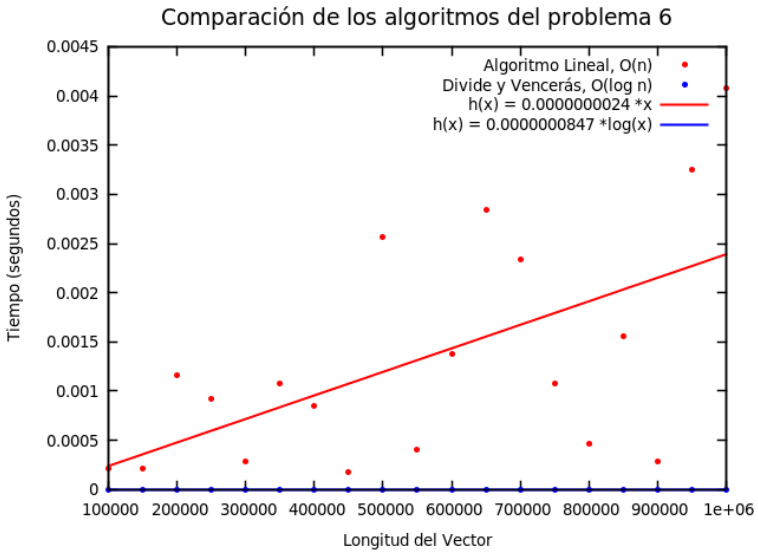

Imagen 4. Resultado de las ejecuciones de ambos algoritmos.

Como vemos, hay un grave problema: una ejecución en un vector de gran tamaño puede obtener tiempo 0. ¿Por qué se produce esto?. Si nos fijamos en ambos algoritmos, en el momento en el que se encuentra p su ejecución para. ¡Si se tiene suerte el algoritmo podría terminar en la primera iteración! Es decir, en el mejor caso ambos algoritmos son $\theta(1)$. Para remediarlo, debemos realizar la media de varias ejecuciones sobre vectores aleatorios en cada una de ellas.

La Imagen 5 se ha obtenido como la media de 1000 ejecuciones para cada tamaño de vector sobre diferentes vectores aleatorios. Podemos ver que el algoritmo lineal mantiene ese comportamiento en media y la gran diferencia con el algoritmo recursivo que parece constante en comparación.

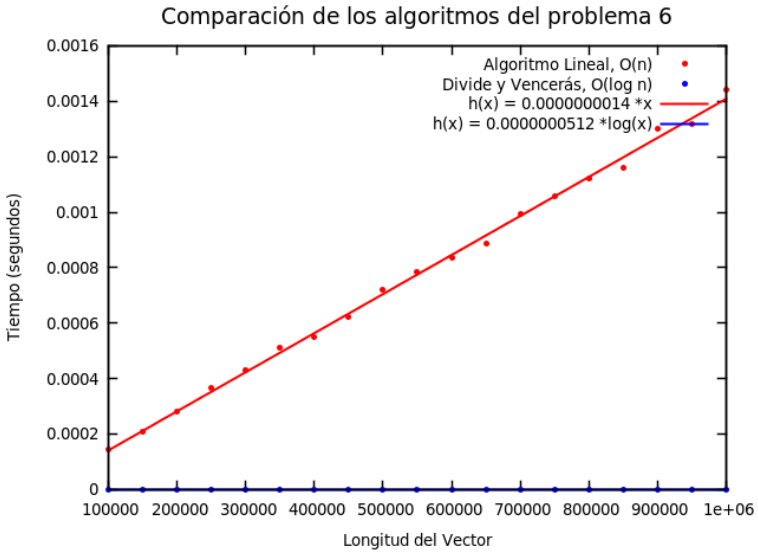

Imagen 5. Media de ejecuciones para ambos algoritmos.

La Imagen 6 muestra solamente el algoritmo recursivo para ver que efectivamente tiene un comportamiento logarítmico. Sin embargo, al ser tan rápido a pesar de ser datos medios son subceptibles a una gran varianza, lo que explica su extraña distribución.

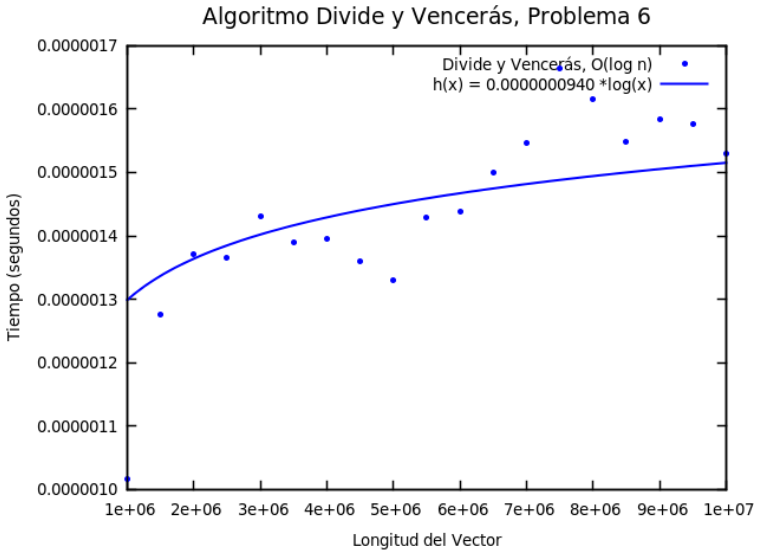

Imagen 6. Media de ejecuciones para el algoritmo logarítmico.

3 Opcional: Problema 2

3.1 Enunciado del problema

Muchos sitios web intentan comparar las preferencias de dos usuarios para realizar sugerencias a partir de las preferencias de usuarios con gustos similares a los nuestros. Dado un ranking de n productos (p.ej. películas) mediante el cual los usuarios indicamos nuestras preferencias, 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 $1, \ldots, n$, y que uno de los rankings siempre es $1, \ldots, n$ (si no fuese así bastaría reenumerarlos) y el otro es $a_1, a_2, \ldots, a_n$, 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 $v[i] = ai, \forall i = 1, \ldots, n$. 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 de "fuerza bruta" obvio. Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

3.2 Resolución teórica del problema

Como se ha ido indicando en el enunciado del problema, podemos abstraernos en el problema propuesto para reducirlo a un problema equivalente, que es el de contar inversiones en un vector. Decimos que hay una inversión en un vector v, si para un par de indices i, j del vector verificando $i < j$, se tiene que $v[i] > v[j]$.

Se puede pensar fácilmente en una forma sencilla de resolver este problema "a la fuerza bruta", sin más que recorrer todas las posibles parejas $(i, j) : i < j$ e incrementar el número de inversiones cada vez que se nos presente la condición $v[i] > v[j]$. Analizaremos esta versión junto con otras basadas en ciertos algoritmos de ordenación, que nos permiten aprovechar las mismas iteraciones del algoritmo para incrementar las inversiones sin un exceso computacional apreciable. Finalmente, concluiremos de esta forma con un algoritmo divide y vencerás basado en el algoritmo mergesort.

3.2.1 Solución trivial

Ya hemos mencionado en el apartado anterior la idea para resolver trivialmente nuestro problema. Dado un vector v de n elementos con índices $0, ..., n-1$, la idea trivial es la siguiente:

- Para i desde 0 hasta $n-1$
  - Para j desde $i+1$ hasta $n-1$
    - Si $v[i] > v[j]$
      - inversiones++
- Devolver inversiones.

Claramente, estamos hablando de un algoritmo cuadrático. Concretamente, la expresión condicional del interior de los bucles es de orden constante, y el número de iteraciones en función del tamaño n del vector es:

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-i-1) = \frac{n(n-1)}{2}$$

Vemos así que, claramente, el algoritmo trivial es de un orden de eficiencia $\theta(n^2)$. A continuación se muestra una versión sencilla de este algoritmo en código C++:

```
# Código para el algoritmo trivial:
# v es un array de tamaño n
# se devuelven las inversiones (inv)
inv = 0;
for(int i = 0; i < n; i++)
    for(int j = i+1; j < n; j++)
        if(v[i] > v[j])
            inv++;

return inv;
```

3.2.2 Más soluciones posibles del mismo orden que el trivial (Idea de la ordenación)

Antes de comenzar esta sección, se tiene en cuenta que hemos implementado tres algorítmos más, por lo que sería muy pesado poner en todos ellos pseudocódigos, sabiendo además que estos algorítmos son básicamente los de ordenación modificados. Así pues, se notará en que momento lo modificamos, pero no el desarrollo completo de la explicación.

Para realizar este mismo problema, aunque no se pedía en el ejercicio hemos pensado en más formas de solucionar el algorítmo, en concreto recurriendo a la ordenación del vector. Hemos encontrado que hay algunos de ellos que podemos contabilizar el número de inversiones.

Burbuja:Un ejemplo de este tipo de algorítmos es el burbuja. Como el algorítmo funciona moviendo un elemento a su posicion siguiente(o anterior) si este elemento está invertido con respecto a esa posición, sabemos que el numero de inversiones que tiene el vector resultante es una menos que el vector original. Así, si cada vez que intercambiamos dos elementos incrementamos un contador, al ordenar completamente el vector, este contador nos indica el número de inversiones que se han realizado, ya que el vector ordenado tiene cero inversiones.

Inserción:Otro algorítmo de ordenación que conserva la idea de que el orden se tiene en cuenta, y con ello las inversiones, es el algorítmo de inserción. En el,en cada paso se pasa el elemento mínimo al primer elemento no ordenado. Nótese que no podemos cambiarlo por el elemento en esa posición, si no que debemos pasarlo paso a paso para que no cambie el orden de los elementos intermedios. Así, al pasar un elemento a la posición del primer elemento no ordenado, el numero de inversiones que realiza tal elemento es su posición menos la posición del primer elemento no ordenado, que es el número de elementos que "pasa por encima", o el numero de inversiones que realiza. Lo volvemos a almacenar en un contador y cuando el vector queda ordenado, en ese contador quedará el numero de inversiones hechas.

La idea de ordenar el vector parece que nos ha sido productiva, cambiando así de concepto para contar inversiones. En concreto, la idea de contar varias inversiones a la vez(con el algorítmo de inserción) es una idea que no podíamos contemplar con la solución trivial. La pregunta inmediata sería si otros algorítmos más rápidos, como los $O(nlog(n))$ también nos servirían para esta tarea en especial.

3.2.3 Solución divide y vencerás: modificación del Quicksort y del Mergesort.

El algorítmo en media más rápido a nivel de ordenación que querríamos implementar es el quicksort, que usa el método de divide y vencerás. Ya de primeras nos damos cuenta que la idea de los intercambios indiscriminados que se toma con el método usual de utilizar dos punteros, uno al principio y otro al final nos "desordena" la idea de contar inversiones, por que no sabemos si los elementos que pasa por encima son elementos invertidos con respecto a el mismo o estaba en una posición sin inversiones, y el numero de inversiones aumenta. Ese paso se subsana utilizando otra forma de modificar el vector, que sería insertando uno de los elementos antes que el otro, y así respetamos el orden de esos elementos, sumando o restando uno, dependiendo de si hemos saltado un elemento que esaba invertido o no.

Por simplicidad, en vez de utilizar este tipo de partición del vector, utilizamos otra en la que ambos punteros empiezan al principio. Decimos por simplicidad porque solo se tiene en cuenta que hay que intercambiar un elemento y no dos, respetando así la idea del algorítmo, que en el caso anterior se basaba en la idea de cambiar dos elementos, y en este solo uno. El resto del algoŕitmo se comporta de igual manera, salvo que recojemos de ordenar cada parte el número de inversiones realizada.

El método utilizado en codigo c++ para dividir el vector en dos subvectores y un pivote medio sería el siguiente:

```
unsigned long long dividir_qs(vector <int> &T, int inicial, int final, int & k){
  int pivote;
  int aux;
  int pp;
  unsigned long long contador = 0;

  pivote = T[inicial];
  k = pp = inicial+1;

  while(pp < final){
    if(T[pp] < pivote){
      aux = T[pp];
      for(int i = pp; i > k;i--){
        T[i] = T[i-1];
      }
      T[k] = aux;
      contador += (pp-k);
      k++;
    }
    pp++;
  }

  aux = T[inicial];
  for(int i = inicial; i < k-1;i++){
    T[i] = T[i+1];
  }
  T[k-1] = aux;
  contador += k-inicial-1;

  return contador;
}
```

Al utilizar este método se nos ocurre pensar que tambien puede ser del mismo orden que el quicksort, sin embargo el método para partir el vector usa la inserción, por lo que tememos que sea cuadrático. En la siguiente gráfica se compara este mètodo con los otros algorítmos expuestos anteriormente:

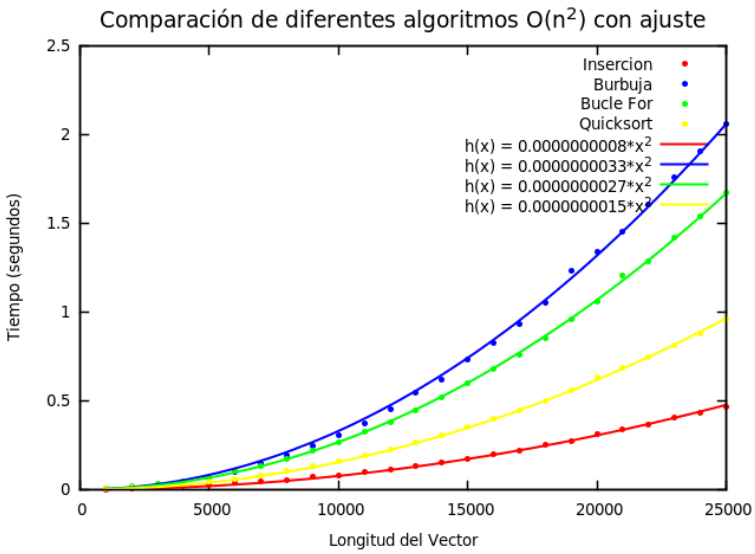

Imagen 1. Gráficos que comparan la eficiencia de los algorítmos expuestos hasta ahora.

A priori nos puede pasar que nos sorprenda tal comparación. ¿Cómo el algorítmo de inserción utilizado puede ser más rápido que el quicksort? Esto es facil de explicar, ya que el quicksort era tan rápido por el intercambio, no por la inserción que lo hace lento, de los elementos con respecto al pivote. Podemos concluir que estamos haciendo una inserción por pasos cada vez, una mezcla entre burbuja e inserción, ya que movemos los elementos solo lo que pueden ser movidos hacia la izquierda del vector por el método de inserción. Por tanto debe tener una eficiencia, como hemos visto, no mayor que el burbuja pero si mayor que el inserción.

Una vez tenido este chasco, se nos ocurre utilizar otro algorítmo logarítmico, en este caso el MergeSort. Aunque es más lento que el quicksort a la hora de ordenar, veamos si puede mantener su condición de algoritmo "n logaritmico":

3.2.3.1 Solución final. Mergesort modificado. Vamos a plantear la solución divide y vencerás partiendo de un ejemplo sencillo. Sea $v = [4, 2, 1, 3]$ un vector de tamaño $n = 4$. Para usar el paradigma divide y vencerás partimos nuestro vector por la mitad, obteniendo los subvectores izquierdo $[4, 2]$ y derecho $[1, 3]$. Ahora partimos de que conocemos la solución de este subproblema, es decir, conocemos las inversiones de cada subvector, que son 1 en el izquierdo, y 0 en el derecho. Ahora tenemos que juntar de nuevo los vectores y a partir de sus respectivas soluciones concluir con la solución final. ¿Cómo hacemos esto?

Supongamos que hubiéramos recuperado los subvectores ordenados tras la división del problema. De esta forma quedaríamos con los subvectores [2,4] y [1,3]. Esta ordenación no nos afectaría al resultado puesto que estamos partiendo de que ya hemos obtenido las inversiones de cada subvector, y como permanecen las mismas componentes, las inversiones existentes entre ambos subvectores van a seguir siendo las mismas. Una vez razonado esto, veamos cómo podemos obtener de forma sencilla las inversiones entre ambos vectores. Además, como hemos usado en la hipótesis de división la ordenación, deberíamos juntar de forma ordenada los subvectores para no romper con la técnica divide y vencerás. De esta forma, podemos seguir un razonamiento muy simple: creamos un nuevo vector donde almacenaremos el ordenado de los subvectores,recorremos nuestros dos subvectores, con un índice al inicio de cada subvector y realizamos los siguientes pasos:

- Si el elemento que marca el índice del subvector izquierdo es menor o igual que el del subvector derecho, no tenemos inversión. De hecho, la condición de que los subvectores estén ordenados nos permite afirmar que no tenemos ninguna inversión del elemento izquierdo con todo el restante subvector derecho, por lo que podemos incrementar el índice izquierdo sin aumentar las inversiones con total tranquilidad. El elemento izquierdo, obviamente, es el que añadimos al vector ordenado.
- Si el elemento que marca el indice del subvector izquierdo es mayor que el del subvector derecho, vamos a tener inversiones. La ordenación de los subvectores nos permite afirmar que todo el subvector restante

izquierdo restante por recorrer va a tener una inversión con el elemento del subvector derecho. De esta forma, añadimos tantas inversiones como elementos queden por recorrer en el subvector izquierdo. Añadiremos en este caso el elemento derecho al array ordenado e incrementaremos el índice derecho.

Esto lo hacemos para cada índice, hasta haber recorrido los subvectores completos. Hay que tener en cuenta que cuando un vector se haya recorrido completo, los elementos restantes del otro se añadirán directamente sin inversiones adicionales. Nótese que tal y como se ha descrito, esta parte de nuestro algoritmo es lineal respecto a la suma de los tamaños de los subvectores, que es justamente el tamaño del vector inicial, en nuestro ejemplo, $n = 4$. También debemos percatarnos ahora que lo que acabamos de proponer no es ni más ni menos que el algoritmo de ordenación Merge Sort (ordenación por mezcla) ya conocido, al que se añade simplemente una pequeña modificación para contar las inversiones. Tras el algoritmo se nos devolverá el vector ordenado mediante las mezclas de los subvectores correspondientes. Como caso base para las divisiones tendremos, como en el propio Merge Sort, los vectores de un elemento.

Volviendo a nuestro ejemplo, ya podemos finalizarlo concluyendo que tendremos 4 inversiones: una es la obtenida de los subcasos izquierdo y derecho, y las restantes de la mezcla (al comparar el 2 de [2,4] con el 1 de [1,3] tenemos inversiones para el 2 y el 4; la comparación de 2 con 3 no da inversión, y finalmente comparamos 4 y 3 que nos da la última).

3.2.3.1.1 Pseudocódigo y eficiencia. Una vez descrita nuestra solución, planteamos el algoritmo y su eficiencia. Tenemos dos partes claramente diferenciadas en el algoritmo: la parte que se encarga de dividir y la parte que se encarga de mezclar. Sus pseudocódigos se muestran a continuación:

```
#Función de mezcla de los subarrays:
def mergeCount(array, begin, middle, end):
    sorted_array = [ ]
    sol = 0; j = middle; i = begin

    while i < middle:
        if j < end:
            if array[i] <= array[j]:
                sorted_array.append(array[i])
                i += 1
            else:
                sorted_array.append(array[j])
                sol += (middle - i)
                j += 1
        else:
            sorted_array.append(array[i])
            i += 1

    for i in range(j, end):
        sorted_array.append(array[i])

    for i in range(0, end-begin):
        array[i+begin] = sorted_array[i]

    return sol


#Función que cuenta todas las inversiones:
def countInversions(array, begin, end):
    middle = (begin + end) // 2
```

```
    sol = 0

    if middle - begin > 1:
        sol += countInversions(array, begin, middle)
    if end - middle > 1:
        sol += countInversions(array, middle, end)
    if (begin < middle and middle < end):
        sol += mergeCount(array, begin, middle, end)

    return sol
```

Fijándonos en el algoritmo podemos observar con facilidad la relación de recurrencia que se satisface. Si denotamos $T(n)$ a la función con la eficiencia del algoritmo, y teniendo en cuenta como ya se dijo la linealidad del algoritmo de mezcla, tenemos que en el peor caso (cuando se entra en todas las sentencias condicionales) $T(n)$ verifica:

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

Tenemos una ecuación en diferencias, que tras el cambio de variable $n = 2^k$ podemos resolver con facilidad:

$$T(n) = T(2^k) = 2T(2^{k-1}) + 2^k \iff T(2^k) = a2^k + bk2^k \iff T(n) = an + bn\log n \in O(n\log n)$$

De esta forma, gracias a la estrategia divide y vencerás hemos conseguido mejorar el orden de eficiencia cuadrático de la solución más inmediata a este problema hasta un orden $n\log n$, lo cual es un gran avance respecto al coste computacional.

La eficiencia híbrida que se obtiene se ve representada en la siguiente gráfica:

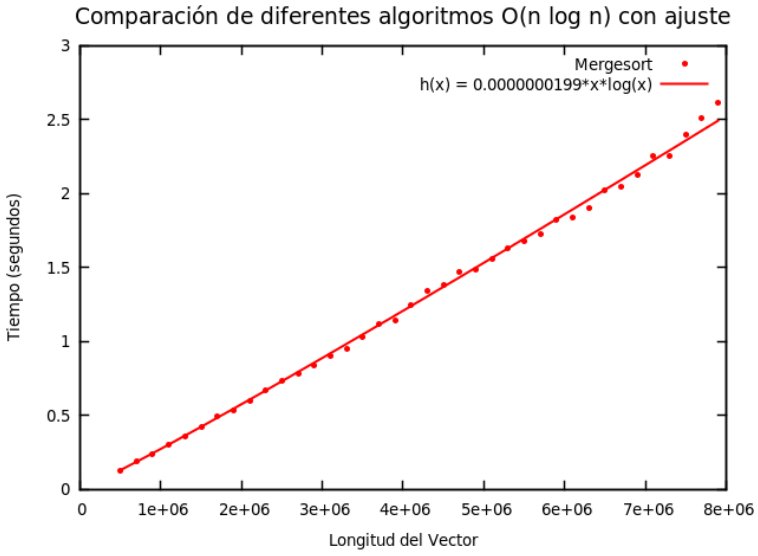

Imagen 2. Gráfico y ajuste del algorítmo MergeSort modificado.

En la siguiente gráfica se pone de manifiesto la relación de todos estos algoritmos mostrados, y la superioridad del Mergesort:

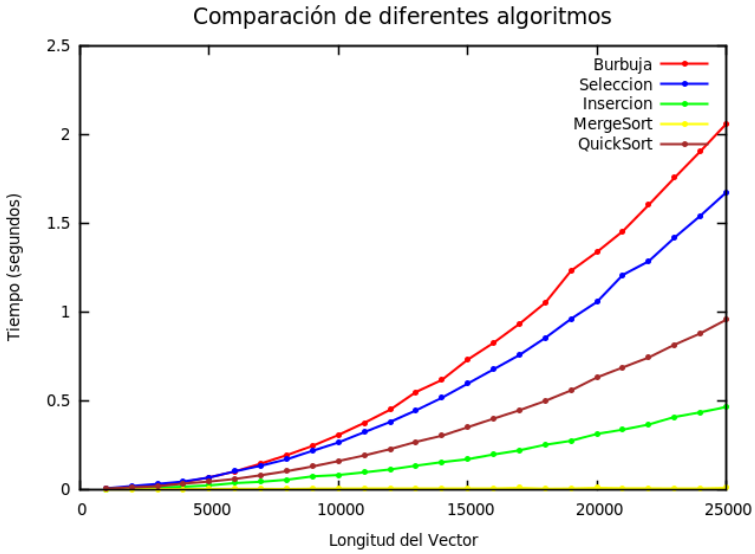

Imagen 3. Gráficos que comparan la eficiencia de los algorítmos expuestos hasta ahora.

3.3 Nota

Hemos desarrollado un algoritmo basado en Quicksort que si es $\theta(n \log n)$ en media. El problema que teníamos era el algoritmo de particionamiento para el pivote, al tener que preservar el orden de los demás elementos era preciso hacer inserciones en lugar de intercambios. Pero si decidimos hacer un particionamiento con un vector auxiliar (ya no es in-place) podemos copiar en tal vector los elementos menores que el pivote y posteriormente los mayores manteniendo el orden. Volvemos a copiar en el vector original los elemtnos con el pivote en su posición. Este particionamiento preserva el orden en $\theta(n)$. Sin embargo, es más lento que el Mergesort pues se recorre el subvector 3 veces durante la partición.

Se explicará más detalladamente en la presentación.