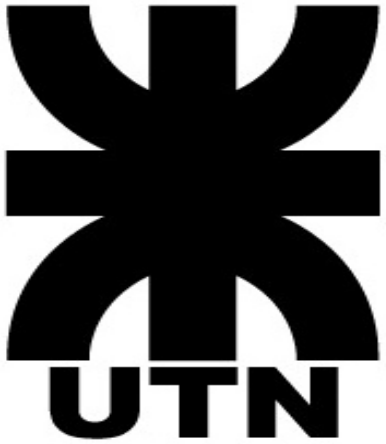


Análisis de eficiencia de algoritmos de ordenamiento

Universidad Tecnológica Nacional
Facultad Regional La Plata

Diciembre, 2012

CHIELLI, José Ignacio
D'UVA, Iván
JACZNIK, Rodrigo Ezequiel
MIGOTTO, Juan Fabrizio

Introducción

El trabajo se desarrollará sobre algoritmos de ordenamiento de vectores. Estos algoritmos toman la secuencia de elementos que componen al vector y los reordenan de alguna manera que queda especificada al momento de la implementación de cada algoritmo. Esto es muy útil por muchos motivos, ya sea para facilitar la lectura de la información, para contar información repetida, para comparar apariciones comunes de información entre 2 vectores, etc. Existen varios tipos de clasificaciones de esta clase de algoritmos:

- Según su estabilidad
- Según el lugar donde realicen el ordenamiento (Ordenamiento interno y externo).
- Según el tiempo que tardan en realizar el ordenamiento dadas entradas ordenadas o inversamente ordenadas. (Ordenamiento natural o no natural).

Hay varios tipos de algoritmos de ordenamiento que caben en distintas clasificaciones y usan distintos métodos.

Para nuestra investigación hemos tomado los algoritmos más utilizados actualmente para el ordenamiento en memoria principal, estos algoritmos desde los más simples a los más complejos han venido siendo utilizados en los últimos años. Algunos de los cuales fueron evolucionando para mejorar su funcionalidad. Son utilizados en la programación estructurada desde los comienzos, con la utilización de vectores, cualquiera sea el contenido que este almacene, desde una variable integer (números enteros), hasta registros de gran tamaño y con un número grande de campos.

Gracias a la aparición de estas formas de almacenar datos, surgió la necesidad de ordenar estos vectores de determinadas formas para poder trabajar mejor con ellos, nos permite búsquedas más eficientes, la posible eliminación de elementos repetidos, y muchas alternativas que nos hacen el trabajo más fácil y simple.

Tema: Algoritmos de ordenamiento y eficiencia

El análisis de eficiencia de cada algoritmo es interesante para analizar ciertos factores a tener en cuenta a la hora de implementar uno u otro. Estos factores son los que componen la *complejidad computacional*: el uso de memoria y otros recursos; el tiempo que se tarda en aplicar el algoritmo frente a distintos tipos de vectores con distinta longitud; la cantidad de pasos necesarios; y las decisiones que se deben tomar al momento de ejecutar el algoritmo.
El análisis de los algoritmos es importante porque el uso accidental o no intencionado de un algoritmo ineficiente puede impactar significativamente en la performance del sistema. En aplicaciones sensibles al tiempo, un algoritmo que tarde mucho en ejecutarse, puede dar resultados fuera de tiempo o inservibles. Un algoritmo ineficiente puede también terminar requiriendo energía computacional y/o almacenamiento poco favorables.

Planteamiento del problema de investigación

Delimitación semántica

Algoritmo: conjunto preescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución.

Algoritmo de ordenamiento: hace referencia a un algoritmo de ordenamiento interno, existen dos tipos de algoritmos de ordenamiento: los internos y los externos. Los primeros hacen referencia al ordenamiento en memoria principal y los segundos a ordenamiento en una memoria secundaria. Para nuestro caso de investigación vamos a utilizar los internos.

Complejidad computacional: este término tiene relación con el uso de memoria y otros recursos; el tiempo que se tarda en aplicar el algoritmo frente a distintos tipos de vectores con distinta longitud; la cantidad de pasos necesarios; y las decisiones que se deben tomar al momento de ejecutar el algoritmo.

Eficiencia: en este trabajo, para hablar de eficiencia nos referiremos a que un algoritmo es más eficiente que otro porque consume menos recursos y, sobre todo, porque tarda menos tiempo en ejecutarse.

Estabilidad: un método de ordenamiento es estable cuando mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Los algoritmos de ordenamiento inestable pueden cambiar el orden relativo de registros con claves iguales, pero los algoritmos estables nunca lo hacen. Si los elementos son indistinguibles, la estabilidad no interesa.

Estructura de control-Si: es una estructura condicional para usar en el diseño de algoritmos. En ella, se evalúa una expresión lógica y dependiendo del resultado, se ejecutan unas sentencias u otras. El cuerpo de la estructura sería:

```
si condición entonces:
    Sentencia
sino
    Otras sentencias
fin si
```

Ejemplo:

```
si a = 10 entonces:
    a = a + b
```

```
sino
    a = a-b
fin si
```

Estructura de control-Repetir para: estructura repetitiva no condicional para utilizar en el diseño de algoritmos. Es un bucle que se repite una cantidad fija de veces y al menos se ejecuta una vez. El cuerpo en forma general de esta estructura es:

```
Repetir para condición1 hasta condición2:
    Cuerpo del bucle
fin repetir para
```

En donde las condiciones pueden ser variables que se igualan a enteros, variables que se igualan a otras variables enteras, etc. En el cuerpo se encuentran las acciones que se quieran repetir. Ejemplo:

```
j = 0
Repetir para i = 1 hasta i = 20:
    j = j + i
fin repetir para
```

Estructura de control-Repetir mientras: estructura repetitiva condicional para utilizar en el diseño de algoritmos. Se trata de un bucle que se repite mientras una condición sea verdadera. Esto puede llevar a bucles infinitos si la condición nunca se vuelve falsa. El cuerpo de la estructura sería:

```
Repetir mientras condición:
    Cuerpo del bucle
fin repetir mientras
```

La condición puede ser cualquiera, hasta una variable booleana con valor verdadero o falso. El cuerpo se repetirá hasta que la condición deje de cumplirse, o nunca terminará. Ejemplo:

```
i = verdadero
Repetir mientras i = verdadero:
    a = escribir(“introducir código”)
    leer(a)
    si a = 123 entonces:
       i = falso
```

```
    fin si
fin repetir mientras
```

Pila de ejecución: se refiere a una sección de memoria que se utiliza para almacenar información acerca de las funciones de un programa. Los tamaños y los detalles técnicos de la pila varían dependiendo de lenguaje de programación, del compilador, del sistema operativo y del tipo de procesador.

Recurso: nos referimos a uso de la CPU y de la memoria.

Vector: es un arreglo unidimensional (de una sola dimensión), que constituye un tipo de datos estructurado para los lenguajes de programación. Esta compuesto de un número determinado de elementos (tamaño fijo), que son homogéneos (del mismo tipo). El tamaño del vector generalmente es conocido en tiempo de compilación. Sin embargo, dependiendo del lenguaje de programación utilizado, los vectores pueden ser estáticos o dinámicos, dependiendo de si tienen una cantidad fija o variable de memoria. Cuando nos referimos a vectores en este trabajo, estaremos hablando de vectores estáticos y generalmente ya completos.
Los vectores se pueden presentar de 3 maneras, según el orden que se le quiera dar.
Tomaremos un ejemplo de orden para explicar las distintas situaciones, suponiendo que se quiere un vector de enteros con los números del 1 al 5 ordenados de izquierda a derecha. Serán ordenados, si cumplen con el orden especificado, o sea:

1 2 3 4 5

Serán inversamente ordenados si cumplen con el orden, pero exactamente al revés:

5 4 3 2 1

Serán desordenados en caso contrario, por ejemplo:

3 1 4 5 2

Oraciones tópicas

Los propósitos de esta investigación son:

- OT1: Saber cuál algoritmo es el más eficiente en el caso de un vector ordenado.
- OT2: Saber cuál algoritmo es el más eficiente en el caso de un vector ordenado inversamente.
- OT3: Saber cuál algoritmo es el más eficiente en el caso de un vector completamente desordenado.
- OT4: Conocer la complejidad computacional de cada algoritmo mediante análisis del código.

Problema: ¿Qué algoritmo de ordenamiento resulta más eficiente en cada caso?

Objetivos

- Implementar los algoritmos en el lenguaje de programación C.
- Proponer 3 condiciones iniciales para los vectores a ordenar. Ordenados, con orden inverso y desordenados de manera aleatoria.
- Obtener los resultados experimentalmente.
- Obtener los resultados analiticos.
- Comparar los resultados obtenidos para cada algoritmo.
- Determinar qué algoritmo es más eficiente para cada caso.

Se tomarán algunos algoritmos de ordenamiento interno. Compararemos su desempeño para realizar un ranking de eficiencia según el escenario del que se trate.

Los resultados se van a adquirir por métodos analiticos y experimentales. Vamos a analizar analiticamente cada caso evaluando un peor y un mejor caso, obteniendo así un promedio de respuesta posible, para esto usaremos notación O. Por otro lado, de manera experimental, vamos a implementar los algoritmos y ejecutarlos en una máquina y así poder medir su complejidad computacional, principalmente en función del tiempo que tardan en ejecurtarse, dado que hoy en día el tiempo de procesamiento es uno de los recursos más importantes que se deben tener en cuenta.

Marco teórico

Teoría de la Complejidad computacional

Esta teoría es una rama de la teoría de la computación. Estudia la manera de clasificar problemas de acuerdo con la dificultad inherente de resolverlos. Los problemas se resuelven mediante algoritmos, que tendrán un peor o mejor caso dependiendo de la dificultad del problema que resuelvan. Esta dificultad depende de los factores que componen la complejidad computacional.

En el caso de los algoritmos de ordenamiento de vectores, la complejidad temporal es el número de pasos que lleva ordenar los elementos del vector. Si se toma un vector con *n* elementos, y se puede resolver en n^2 pasos, la complejidad es n^2. El número de pasos no se refiere solamente al código, sino que depende también de los demás factores de complejidad. Para no tener que hablar del costo exacto del algoritmo, se usa la notación O, que es definida a continuación.

Complejidad temporal

La complejidad temporal de un algoritmo cuantifica la cantidad de tiempo que lleva ejecutar el algoritmo como función de una entrada determinada. Comúnmente, es representada con la *notación O*. Expresada de esta manera, la complejidad temporal se dice que se describe

asintóticamente, o sea, con el tamaño de la entrada tendiendo a infinito. Por ejemplo, si el tiempo que requiere un algoritmo para ejecutarse con una entrada de tamaño *n* es $5n^3 + 3n$, entonces la complejidad temporal es $O(n^3)$.

La complejidad temporal es comúnmente estimada contando los números de operaciones elementales que ejecuta el algoritmo, que tienen cantidades fijas de tiempo para realizarse. El tiempo y el número de operaciones elementales difieren por más de un factor constante.

La *notación O* es usada como una representación del escenario del peor caso. Interesa como una función $f(x)$ puede crecer hacia $g(x)$, de manera que $f(x) = O(g(x))$.
Como la performance en el tiempo de un algoritmo puede variar con diferentes entradas del mismo tamaño, se usa comúnmente el peor caso de complejidad del algoritmo, denotado como $T(n)$, que se define como el máximo tiempo que tarda en ejecutarse el algoritmo ante cualquier entrada de tamaño n. Las complejidades temporales se clasifican con la función $T(n)$. Por ejemplo, un algoritmo con $T(n) = O(n)$ se denomina algoritmo de tiempo lineal, y un algoritmo con $T(n) = O(n^2)$, se denomina algoritmo de tiempo exponencial.

Notación O

El análisis de algoritmos estima el consumo de recursos de un algoritmo y esto nos permite comparar los costos relativos de dos o más algoritmos para resolver el mismo problema.

En estos términos, es importante el concepto de *razón de crecimiento*, que es la razón a la cual el costo de un algoritmo crece conforme el tamaño de la entrada crece. Nos permite comparar el tiempo de ejecución de dos algoritmos sin realmente escribir dos programas y ejecutarlas en la misma máquina.

Usualmente se mide el tiempo de ejecución de un algoritmo, y el almacenamiento primario y secundario que consume dependiendo de una determinada entrada. Es muy importante el número de operaciones básicas requeridas por el algoritmo para procesar una entrada de cierto tamaño.

Una razón de crecimiento de *cn* se le llama *razón de crecimiento lineal*. Si la razón de crecimiento tiene el factor n^2, se dice que tiene una *razón de crecimiento cuadrático*. Si el tiempo es del orden 2^n se dice que tiene una *razón de crecimiento exponencial*. Si el tiempo es del orden $\log(n)$ se dice que tiene una *razón de crecimiento logarítmico*.

Para algunos algoritmos, diferentes entradas para un tamaño dado pueden requerir diferentes cantidades de tiempo. Esto lleva a que cada algoritmo tenga un caso de *peor rendimiento*, otro de *mejor rendimiento* y otro de *rendimiento promedio o típico*.

Si analizamos la razón de crecimiento $f(n)$ en una computadora (10.000 operaciones en una hora) que resuelve un problema de tamaño n, y otra computadora 10 veces más rápida (100.000 operaciones en una hora), que resuelve un problema de tamaño n'.

$f(n)$	n **(para 10000 op)**	n' **(para 100000 op)**	**Cambio**	n'/n
$10n$	1000	10000	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	----

Entre $10n$, $20n$, y $2n^2$, notamos que $2n^2$, supera eventualmente a $10n$ y $20n$, difiriendo solamente en un valor mayor de n, donde ocurre el corte.

Por esto, cuando queremos una estimación del tiempo de ejecución u otros requerimientos de recursos del algoritmo, se suelen ignorar las constantes. Esto simplifica el análisis y nos mantiene pensando en la razón de crecimiento. A esto se le llama *análisis asintótico* del algoritmo. Esto es el estudio de un algoritmo conforme el tamaño de entrada "se vuelve grande" o alcanza un límite (en el sentido del cálculo).

Cota superior

La cota superior de un algoritmo indica una cota o la máxima razón de crecimiento que un algoritmo puede tener. Generalmente hay que especificar si es para el mejor, peor o caso promedio.

Por ejemplo, podemos decir: "tal algoritmo tiene una cota superior a su razón de crecimiento de n^2 en el caso promedio". Se adopta una notación especial llamada *notación O*, por ejemplo $O(f(n))$, para indicar que la cota superior del algoritmo es $f(n)$. En términos precisos, si $T(n)$ representa el tiempo de ejecución de un algoritmo, y $f(n)$ es alguna expresión para su cota superior, $T(n)$ está en el conjunto $O(f(n))$, si existen dos constantes positivas c y n_0 tales que $|T(n)| \leq c|f(n)$ para todo $n > n_0$.

Cota inferior

La cota inferior de un algoritmo indica la mínima cantidad de recursos que un algoritmo necesita para alguna clase de entrada. Se denota con la letra griega omega Ω. $T(n)$ está en el conjunto $\Omega(g(n))$, si existen dos constantes positivas c y n_0 tales que $|T(n)| \geq c|f(n)$ para todo $n > n_0$.

Cotas iguales

Si las cotas superior e inferior son la misma, se indica con la *notación theta Θ*.

Algoritmos de ordenamiento

Ordenamiento de Burbuja (*Bubblesort*)

Algoritmo de ordenamiento estable que utiliza el método de intercambio directo. Compara cada elemento con el siguiente y, de ser necesario, los intercambia, realizando esta tarea hasta que no se necesiten más intercambios.

```
1    procedimiento burbuja( A : vector de enteros )
2        variables
3            i, j, temporal: entero
4
5        repetir para i = longitud(A) - 1 hasta i > 0:
6            repetir para j = 1 hasta  j <= i:
7                si A[j-1] > A[j] entonces:
8                    temporal = A[j-1]
9                    A[j-1] = A[j]
10                    vector[j] = temporal
11               fin si
12            fin repetir para
13       fin repetir para
14   fin procedimiento
```

Del análisis del algoritmo, se pueden concluir ciertas observaciones interesantes, teniendo en cuenta que las asignaciones (líneas 8, 9 y 10) y comparaciones simples (línea 7) no tienen complejidad computacional comparado con los bucles:

- Existen dos bucles de repetición incondicional (**repetir para)** anidados (línea 5 y 6) y, sabiendo que se repetirán n veces, siendo n la cantidad de elementos del vector, la complejidad del algoritmo es de $O(n^2)$.
  El primer bucle se repetirá $n-1$ veces, pero se podría considerar que un elemento no tendría relevancia en un número muy grande. Luego el segundo bucle se repetir n veces. Por lo tanto la complejidad computacional será $n * n = n^2$ o $O(n^2)$.
- Se puede ver que necesariamente se realizarán n^2 comparaciones y también,

posiblemente, muchos intercambios dependiendo del caso.

- Por último, podemos concluir que el peor escenario para que este algoritmo se ejecute, sería en el caso de que el vector esté ordenado inversamente o desordenado, ya que se realizarían más intercambios que en otro caso. De cualquier manera, en cualquier escenario que se ejecute, se recorrerá todo el vector haciendo comparaciones, por lo que la complejidad es siempre la misma.
  Así también, podríamos decir que el mejor escenario para que este algoritmo se ejecute, sería el caso de que el vector esté ordenado, porque no se realizarían intercambios, y por lo tanto, se ejecutaría en menos tiempo.

Ordenamiento por Inserción (*Insertion sort*)

Algoritmo de ordenamiento estable que utiliza el método de inserción. Toma un elemento que es comparado con los demás elementos del vector que están ordenados. Esto se realiza hasta que se encuentre lugar para el elemento o hasta que el vector termine. Luego, el elemento se inserta debiendo desplazar los demás elementos.

```
1    procedimiento insercion( A : vector de enteros )
2        variables
3            i, j, indice: entero
4
5       repetir para i = 1 hasta longitud(A):
6            indice = A[i]
7            j = i - 1
8            repetir mientras j >= 0 y A[j] > indice:
9                A[j+1] = A[j]
10               j = j - 1
11           fin repetir mientras
12           A[j+1] = indice
13       fin repetir para
14   fin procedimiento
```

Del análisis del algoritmo, se pueden concluir ciertas observaciones interesantes, teniendo en cuenta que las asignaciones (líneas 6, 7, 9, 10 y 12) y comparaciones simples (línea 8) no tienen complejidad computacional comparado con los bucles, y además que en este algoritmo hay un clara diferenciación entre el mejor y el peor caso por la inclusión de un bucle condicional (línea 8). Entonces se puede decir:

- En el mejor de los casos, si el vector está previamente ordenado, se ejecutará solamente el bucle incondicional n veces, siendo n la cantidad de elementos del vector. Se puede decir que este caso tiene una complejidad computacional $O(n)$.
- En el peor de los casos, el vector ordenado inversamente, se ejecutarán ambos bucles n veces y por lo tanto se puede decir que este caso tiene una complejidad computacional $O(n^2)$.
- Se puede decir que en el caso promedio (vector desordenado) la complejidad es $O(n^2/2)$

. en realidad debería decir que el costo en el caso promedio se ve reflejado por la función $O(n^2/2 + n/2)$. Si se considera una cantidad de elementos grande el termino $n/2$ se puede no considerar y la complejidad se puede escribir como $O(n^2/2)$.

Ordenamiento por Selección (*Selection sort*)

Algoritmo de ordenamiento inestable que utiliza el método de selección. Primero busca el "mínimo" o "máximo" elemento entre una posición *i* y el final del vector. Luego se intercambia el elemento encontrado con el de la posición *i*. Esto se hace comenzando desde el primer elemento.

```
1       procedimiento seleccion( A : vector de enteros )
2               variables
3               minimo, i, j, temporal: entero
4
5               repetir para i = 1 hasta n - 1:
6                       minimo = i
7                       repetir para j=i+1 hasta n:
8                               si A[j] < A[minimo] entonces:
9                               minimo = j
10                      fin si
11                      fin repetir para
12              temporal = A[i]
13              A[i] = A[minimo]
14              A[minimo] = temporal
15              fin repetir para
16       fin procedimiento
```

Del análisis del algoritmo, se pueden concluir ciertas observaciones interesantes, teniendo en cuenta que las asignaciones (líneas 6, 9, 12, 13 y 14) y comparaciones simples (línea 8) no tienen complejidad computacional comparado con los bucles:

- Como en el caso del ordenamiento burbuja, aquí también se repetirán los bucles de repetición incondicional n veces, siendo n la cantidada de elementos del vector. Por lo tanto, también tendrá siempre una complejidad de $O(n^2)$.
- En cada ejecución del primer bucle, realiza sólo un intercambio, por lo tanto se realizarán $n-1$ intercambios necesariamente. Por esta razón, habrá poca diferencia de rendimiento en la ejecución del algoritmo en cualquier caso posible.
- En cualquier caso es un algoritmo más eficiente que BubbleSort porque realiza una menor cantidad de comparaciones, esta diferencia se hace más notable cuando el vector crece en cantidad de elementos.

Ordenamiento rápido (*Quicksort*)

Algoritmo de ordenamiento inestable que utiliza el método de partición. Esta basado en la técnica divide y vencerás recursivamente. Se toma un elemento *x* de una posición cualquiera del vector. Se trata de ubicar a un pivote *x* en la posición correcta del vector, de tal forma que

todos los elementos que se encuentran a su izquierda sean menores o iguales a *x* y todos los elementos que se encuentren a su derecha sean mayores o iguales a *x*. Se repiten los pasos anteriores pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición correcta de *x* en el vector.

```
1       procedimiento elegirPivote(A: vector de enteros, inicio_A: entro, long_A: entero)
2               variables
3               i, pivote, valor_pivote, temporal: entero
4
5               pivote = inicio_A;
6               valor_pivote = A[pivote];
7               repetir para i = inicio_A + 1 hasta i <= long_A:
8               si A[i] < valor_pivote entonces:
9                       pivote = pivote + 1
10                      temporal = A[i]
11                      A[i] = A[pivote]
12                      A[pivote] = temporal
13                      fin si
14              fin repetir para
15              temporal = A[inicio_A]
16              A[inicio_A] = A[pivote]
17              A[pivote] = temporal
18
19              retorna pivote
20      fin procedimiento
21
22
23      procedimiento quickSort(A : vector de enteros, inicio_A : entro)
24              variables
25                  pivote: entero
26
27              si inicio_A < longitud(A)) entonces:
28                      pivote = elegirPivote(array, inicio_A, longitud(A))
29                      quickSort(array, inicio_A, pivote - 1)
30                      quickSort(array, pivote + 1, longitud(A))
31              fin si
32      fin procedimiento
```

Del análisis del algoritmo, se pueden concluir ciertas observaciones interesantes, teniendo en cuenta que las asignaciones (líneas 5, 6, 9, 10, 11, 12, 15, 16 y 17) y comparaciones simples (línea 8 y 27) no tienen complejidad computacional comparado con los bucles:

- La complejidad es distinta a las de los otros algoritmos. Para que sea más entendible, hemos dividido el algoritmo en dos procedimientos. El procedimiento *elegirPivote* tiene una complejidad de $O(n)$, ya que la estructura de repetición incondicional debe repetirse n veces, siendo n la cantidad de elementos del vector. Por otro lado, la complejidad del

procedimiento *quickSort* tiene que ver con la subdivisión del vector original a partir del pivote. Entonces quedarán dos subdivisiones con la mitad de los elementos cada una. Cada subdivisión representa dos llamadas recursivas para elegir un pivote. Por lo tanto la complejidad sería $2 * f(\frac{n}{2})$, siendo f el procedimiento ***elegirPivote***
Para obtener la complejidad total, debemos sumar las dos complejidades: $n + 2f(\frac{n}{2})$.
Como el algoritmo es de naturaleza recursiva, esto se resolvería como: $c\frac{n}{2} + nlog_2 n$.
Entonces, la complejidad de un caso promedio sería $O(n\log_2 n)$. En este análisis tomamos el vector como si estuviera desordenado.

- Si el vector estuviera ordenado, cada repetición generaría una sola subdivisión, ya que todos los elementos serían menores que el pivote. Entonces, si tenemos n elementos en el vector, la complejidad sería de $O(n^2)$. Este sería el peor escenario en el que se ejecuta el algoritmo.
- El mayor inconveniente de este algoritmo es el desbordamiento de la pila que ocurre por la sucesivas llamadas recursivas. Cada llamada recursiva genera una nueva entrada en la pila de ejecución y como el algoritmo es ejecutado en una computadora con memoria limitada también es limitada la cantidad de llamadas recursivas.

Las observaciones anteriores se pueden resumir en la siguiente tabla

	Ordenado	Ordenado Inverso	Desordenado
Burbuja	$O(n^2)$	$O(n^2)$	$O(n^2)$
Inserción	$O(n)$	$O(n^2)$	$O(n^2/2)$
QuickSort	$O(n^2)$	$O((n\log_2 n + n^2)/2)$	$O(n\log_2 n)$
Seleccion*	$O(n^2)$	$O(n^2)$	$O(n^2)$

* A pesar de que la complejidad entre el método Burbuja y método de selección es igual, por lo explicado anteriormente, la selección es más eficiente.

Conclusiones

- Tomando en cuenta el escenario con el vector ordenado, todos los algoritmos tienen la misma complejidad (n^2), salvo por el de inserción (n). El tiempo de ejecución de este algoritmo en este escenario, a priori, sería mucho menor que el de cualquiera de los otros algoritmos. Más aún cuando aumenta la cantidad de elementos en el vector es considerable.
- Tomando en cuenta el escenario con el vector desordenado, la menor complejidad se ve

en el método *quicksort*. Por lo tanto, a priori podríamos decir que su ejecución sería más rápida que usando los demás en este escenario.

- Tomando en cuenta el escenario con el vector ordenado inversamente, todos los algoritmos tienen la misma complejidad salvo el *quicksort*. En principio, este algoritmo sería mucho más eficiente que los demás, pero con la salvedad que explicamos anteriormente, no serviría para un vector con muchos elementos. Descartado el *quicksort* nos quedan 3 opciones. Como ya hemos aclarado, en cualquier caso, el método de selección mejora al método burbuja. Pero comparando el método de inserción con el de selección, no podemos saber cual de los dos se ejecutaría más eficientemente.

Hipótesis

- Teniendo un vector ordenado, mientras crece la cantidad de elementos, el algoritmo que presenta mayor eficiencia es el de ordenamiento por inserción.
- Teniendo un vector desordenado, mientras crece la cantidad de elementos, el algoritmo que presenta mayor eficiencia es el de ordenamiento rápido (Quicksort).

Experiencia

Para realizar la verificación, hemos armado una experiencia. Hemos tomado los 4 algoritmos analizados en el trabajo (selección, burbuja, inserción y quicksort) y los 3 escenarios posibles (vector ordenado, vector desordenado y vector inversamente ordenado). Hemos ejecutado los algoritmos en esos 3 escenarios, tomando vectores de 1000, 5000, 10000, 100000, 300000 y 500000 elementos. Vale hacer las siguientes aclaraciones sobre las características del experimento:

Lenguaje de programación utilizado

Utilizaremos el lenguaje de programación C para este trabajo. De cualquier manera, este es un detalle menor, porque el tipo de lenguaje en el que está implementado el algoritmo no influye en la respuesta relativa. Por ejemplo, si el Algoritmo *X* es el que menos tiempo tarda en ejecutarse usando el Lenguaje *L*, si implementamos el mismo *X* en un Lenguaje *L2*, seguirá siendo el más rápido.
Elegimos este lenguaje porque es compilado, y utilizando un algoritmo de ordenamiento en el mismo vector varias veces, siempre va a resultar la misma respuesta.

Especificaciones de la máquina

Como nuestro problema trata de la eficiencia de cada algoritmo, la capacidad de procesamiento de la máquina a utilizar no importa demasiado. Si utilizáramos una máquina con menor capacidad, solamente habríamos de reducir la complejidad del problema (por ejemplo, probando vectores de menor tamaño) y el resultado, en cuanto al algoritmo que resulte más o menos eficiente, sería el mismo.

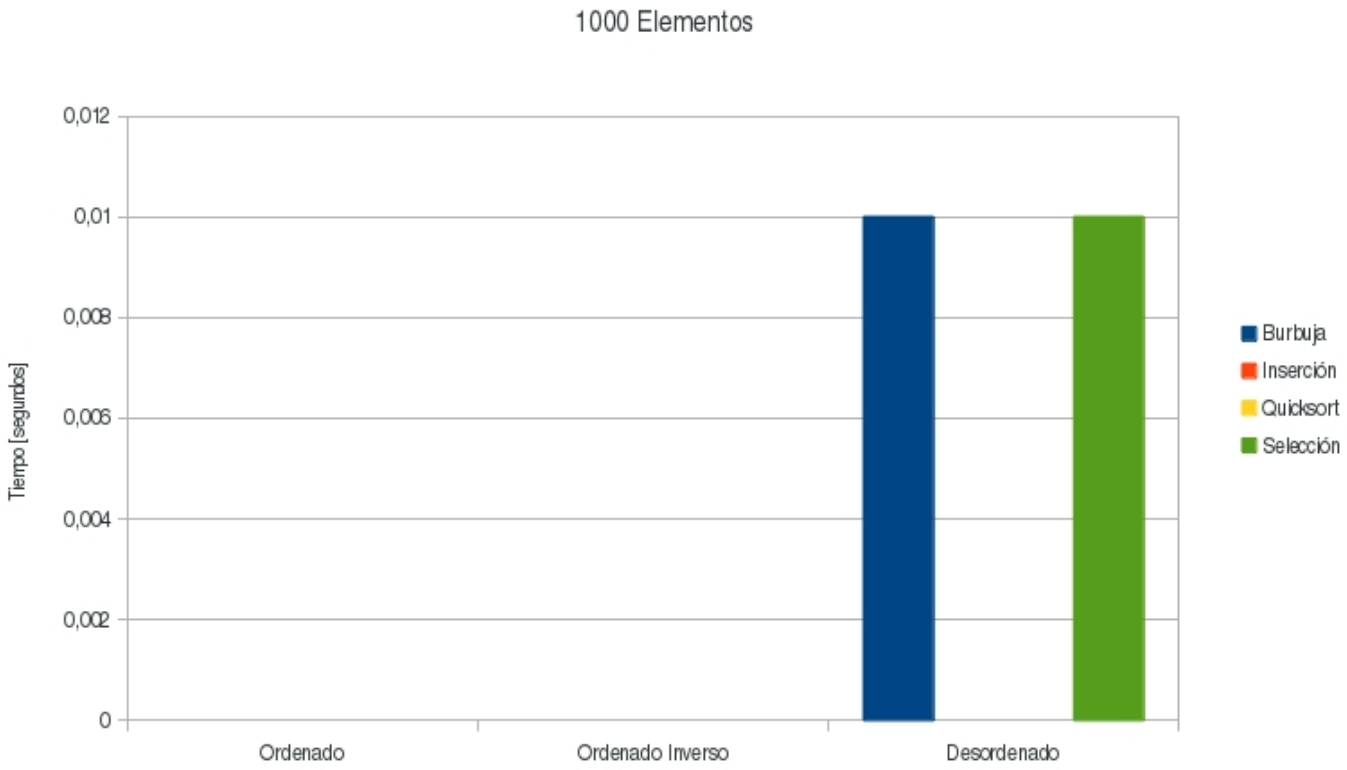
Repositorio del código fuente:

https://github.com/RodrigoJ/Metodologia/blob/master/main.c

Resultados

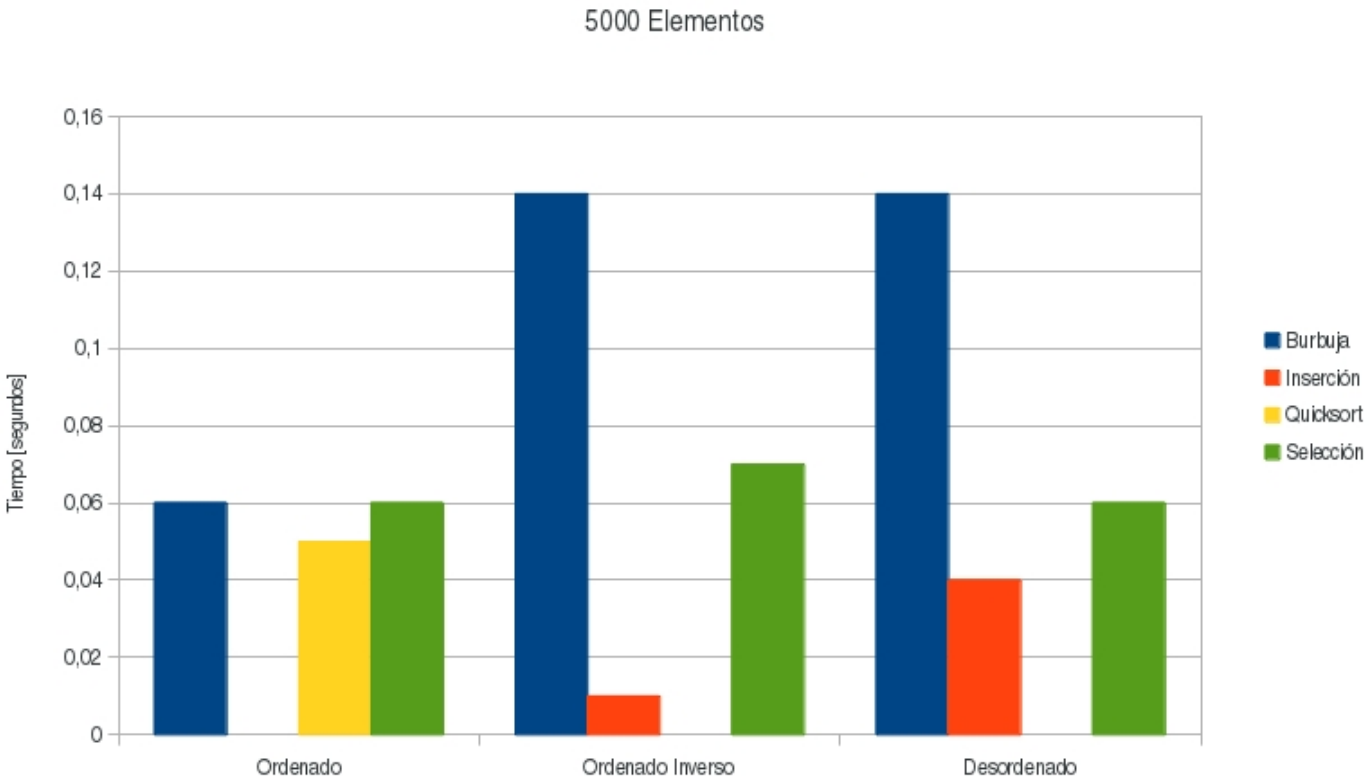
Vector de 1.000 elementos

1000 Elementos	Burbuja	Inserción	Quicksort	Selección
Ordenado	0	0	0	0
Ordenado Inv.	0	0	0	0
Desordenado	0,01	0	0	0,01



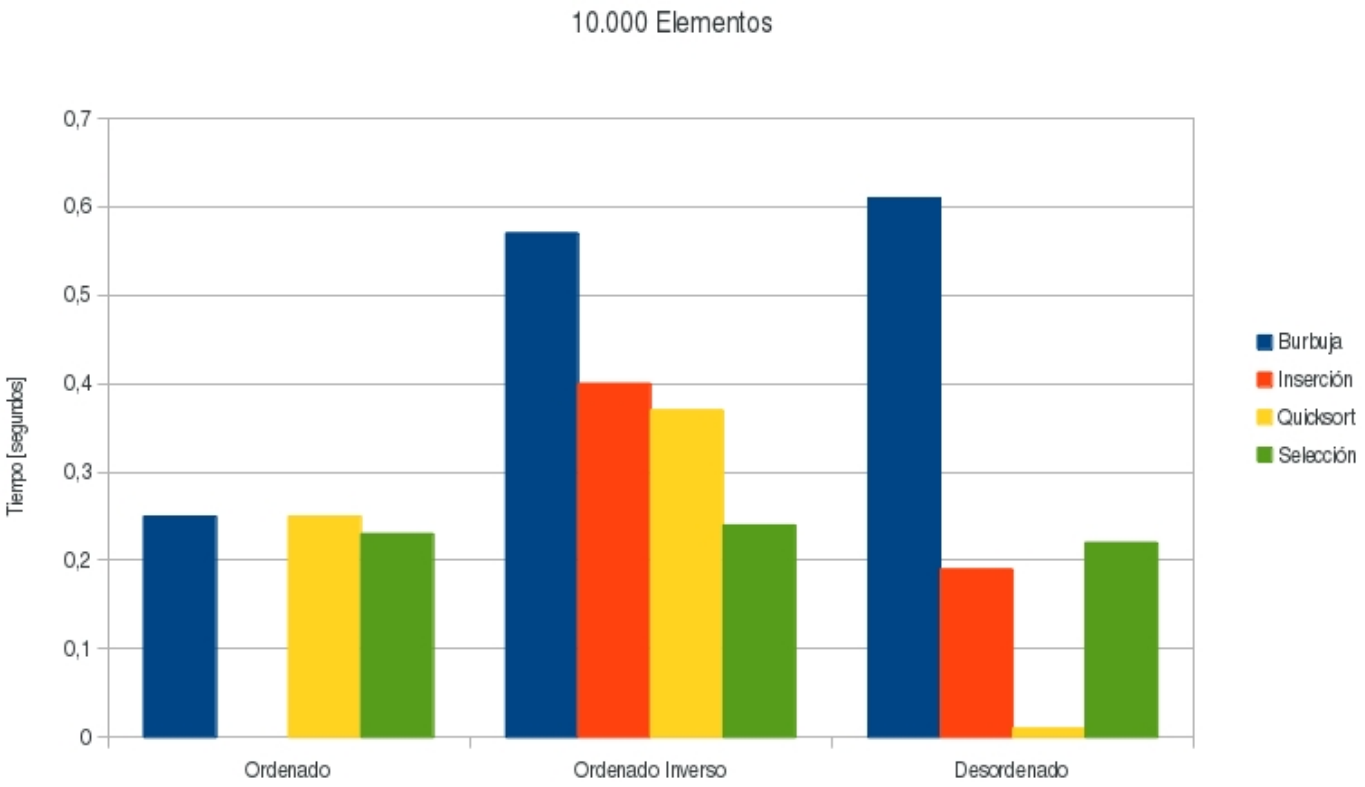
Vector de 5.000 elementos

5000 Elementos	Burbuja	Inserción	Quicksort	Selección
Ordenado	0,06	0	0,05	0,06
Ordenado Inv.	0,14	0,01	0,9	0,07
Desordenado	0,14	0,04	0	0,06



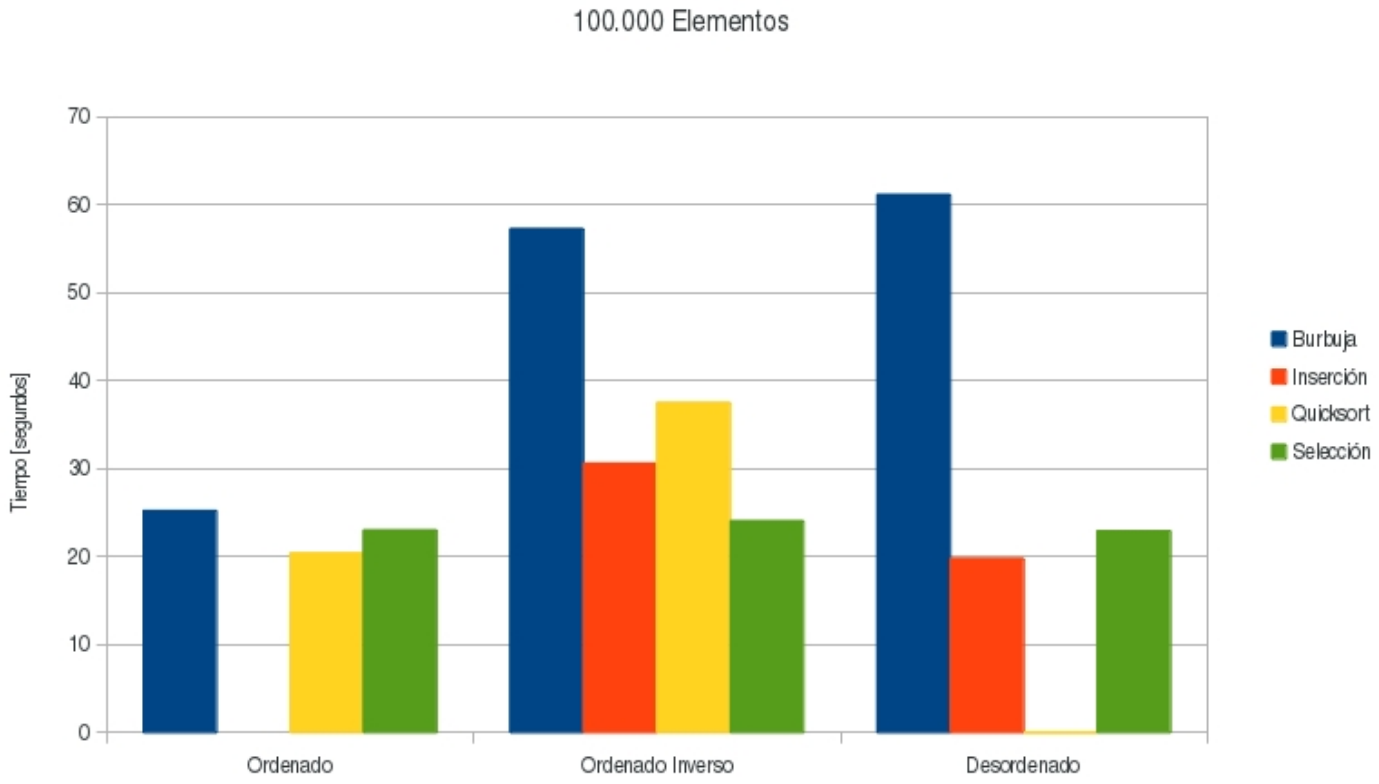
Vector de 10.000 elementos

10.000 Elementos	Burbuja	Inserción	Quicksort	Selección
Ordenado	0,25	0	0,25	0,23
Ordenado Inv.	0,57	0,4	0,37	0,24
Desordenado	0,61	0,19	0,01	0,22



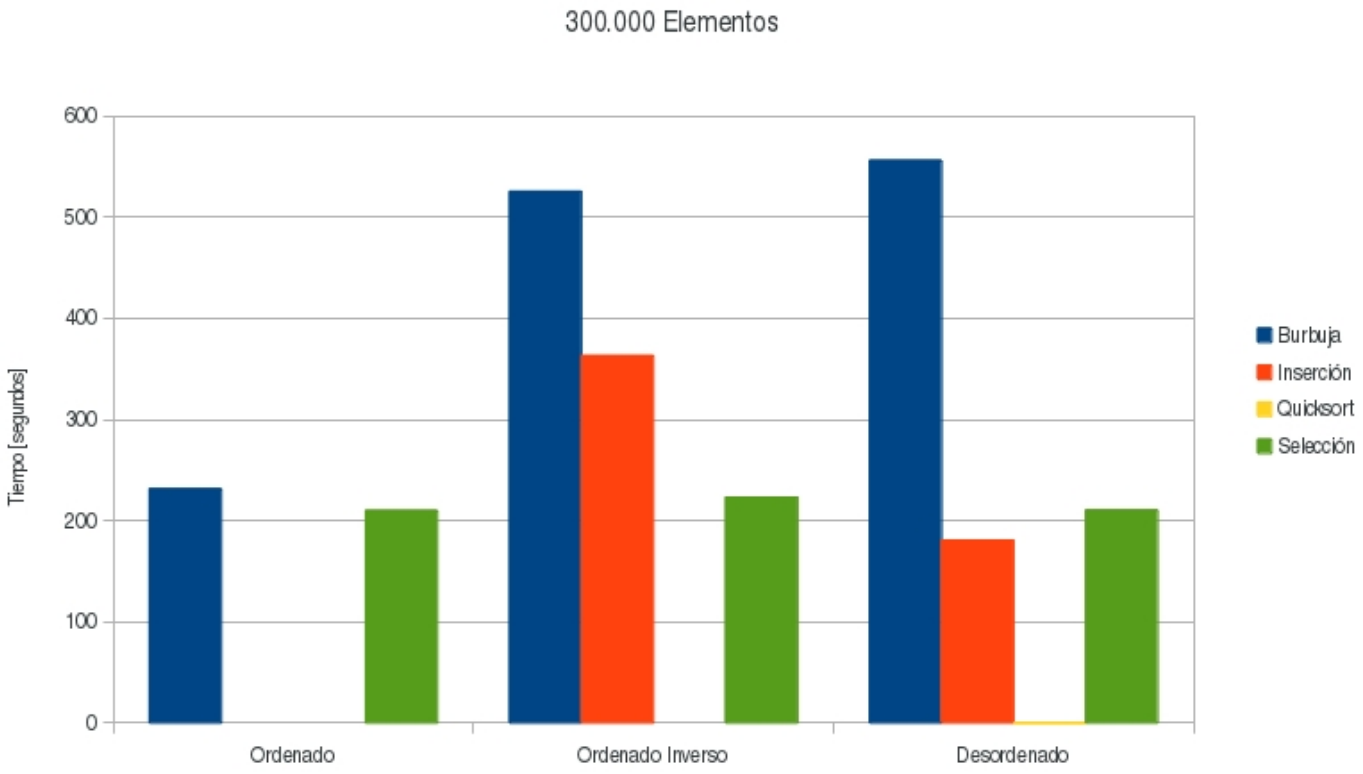
Vector de 100.000 elementos

100.000 Elementos	Burbuja	Inserción	Quicksort	Selección
Ordenado	25,23	0	20,39	23,01
Ordenado Inv.	57,24	30,61	37,51	24,05
Desordenado	61,13	19,77	0,03	22,93



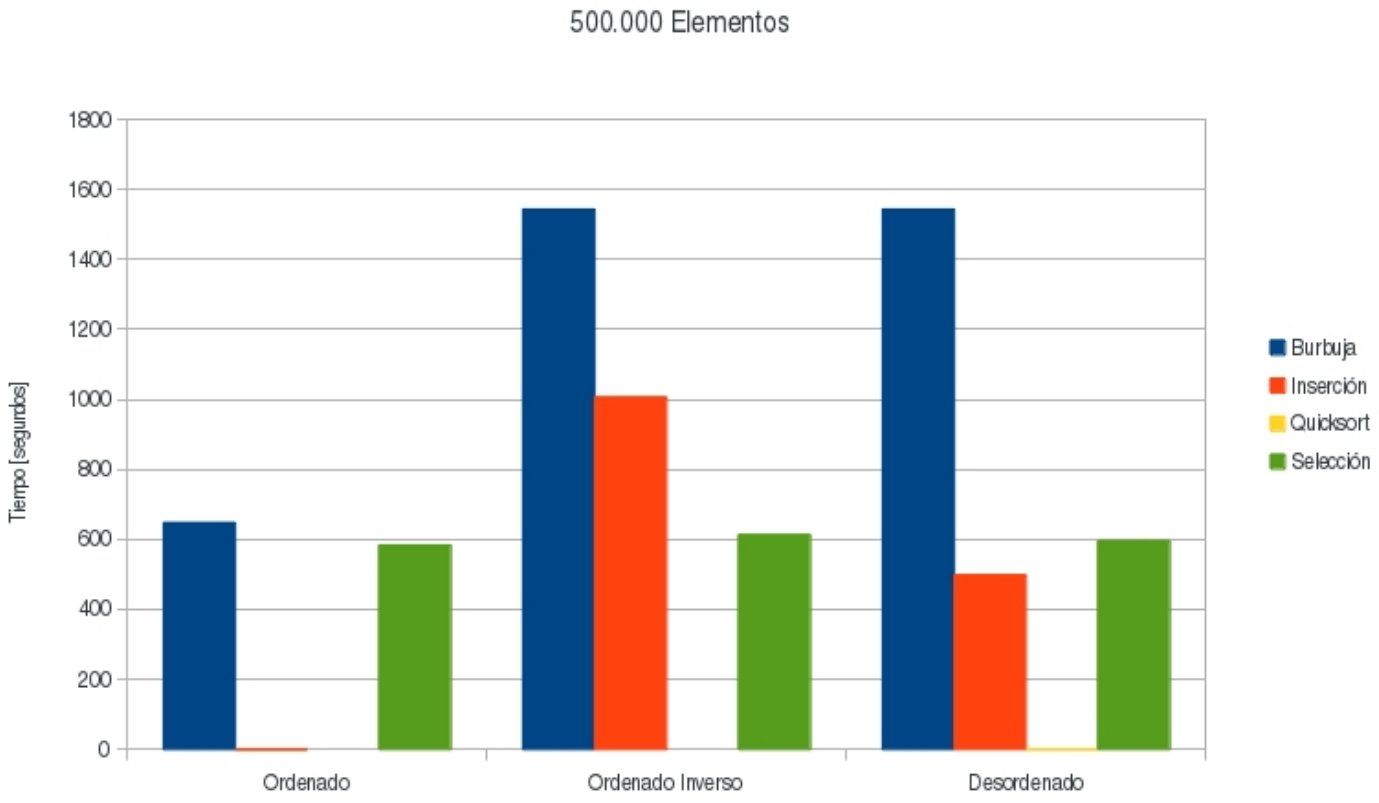
Vector de 300.000 elementos

300.000 Elementos	Burbuja	Inserción	Quicksort	Selección
Ordenado	231,5	0	VS	210,03
Ordenado Inv.	525,56	363,19	VS	223,31
Desordenado	556,29	180,85	0.08	210,51



Vector de 500.000 elementos

500.000 Elementos	Burbuja	Inserción	Quicksort	Selección
Ordenado	647,5	0,01	VS	583,91
Ordenado Inv.	1543,37	1006,24	VS	613,33
Desordenado	1543,37	498,15	0,15	595,83



Observaciones

- **Aclaración - Casos excepcionales**: Si el vector está ordenado o inversamente ordenado, con 300 y 500 mil elementos, el algoritmo *quicksort* termina inesperadamente con un desbordamiento de la pila de ejecución. Esto sucede por la naturaleza recursiva del algoritmo y los recursos escasos de memoria principal. Si se requiere ordenar esta cantidad de elementos en estas condiciones iniciales, se deberá modificar el algoritmo para que este sea iterativo o aumentar la memoria principal.
- En el caso de que el vector esté inicialmente desordenado, el algoritmo *quicksort* presenta una excelente eficiencia, consumiendo solo 0.15 segundos para ordenar 500000 elementos, por ejemplo. **Esto confirma nuestra hipótesis sobre vectores desordenados.**
- A medida que la cantidad de elementos del vector aumenta, si este está inversamente ordenado, el algoritmo *selection sort* se comporta de mejor manera que los demás. Sin embargo, si los elementos no son tantos, el algoritmo *insertion sort* es más eficiente.
- Si el vector está ordenado, el algoritmo más eficiente siempre será el *insertion sort*. **Esto confirma nuestra hipótesis sobre vectores ordenados.**
- En cualquier caso que tomemos, el algoritmo *bubblesort* siempre es el menos eficiente. En ciertos casos su eficiencia se iguala a la de algún otro. Si tenemos 1000 elementos y

el vector está desordenado, es igual de eficiente que el algoritmo *selection sort*; si tenemos 5000 elementos y el vector está ordenado, pasará lo mismo; por último si tenemos 10000 elementos y el vector está ordenado, su eficiencia se iguala a la del algoritmo *quicksort*.

- Cuando son pocos elementos, la diferencia en el uso de los algoritmos es casi imperceptible. Por lo tanto, no importaría demasiado que algoritmo se usa.
- El algoritmo *selection sort* se comporta prácticamente igual en todos los casos.

Este experimento sirve para confirmar las hipótesis que hemos planteado, y además, surge otra: ***Teniendo un vector inversamente ordenado, mientras crece la cantidad de elementos, el algoritmo que presenta mayor eficiencia es el de ordenamiento por selección.***

Bibliografía

Knuth, Donald: ***The Art of Computer Programming*** (2da Edicion, 1998)

Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford . ***Introduction to Algorithms*** (2da edicion, 2001)

Gregg, David. Biggar Paul. ***Sorting in the Presence of Branch Prediction and caches*** (2005)