Presentación Algoritmos de Ordenación

Seminario Doble Grado

Francisco Gallego Salido

Universidad de Granada

7 de mayo de 2016

Todo lo necesario está en github.com/fgallegosalido.

El programa de la visualización de algoritmos podéis descargarlo aquí.

Índice

General

¿Qué es un algoritmo de ordenación?

Un algoritmo de ordenación es un conjunto de pasos que, dada una determinada lista de valores, devuelve la correspondiente lista en orden (de menor a mayor por lo general).

¿Por qué son importantes?

Todo sistema informático o aplicación requiere de ordenar algún tipo de datos, ya sean palabras, números o cualquier otra cosa.

Objetivo

¿Qué vamos a estudiar?

Nuestro objetivo en este seminario será el de explicar qué algoritmos de ordenación existen, cuales son mejores, cuales se deben usar y cuales no, calcular su eficiencia, etc.

También aprenderemos a resolver alguna ecuación en recurrencia para poder calcular la eficiencia de algoritmos recursivos.

Todo esto irá acompañado de gráficas (que vosotros mismos podréis generar) para corroborar todo lo aprendido.

Empecemos pues...

¿Qué es la eficiencia?

Definición

La eficiencia de un algoritmo es básicamente la velocidad del mismo a medida que el número de datos crece. Para representar la eficiencia usaremos la notación O-grande $\mathcal{O}(f)$, donde f es el orden de eficiencia.

En las siguientes diapositivas explicaremos en qué consiste esta notación y qué propiedades tiene.

¿Para qué sirve?

La eficiencia nos servirá para poder decidirnos a la hora de elegir algoritmos e implementarlos.

Podríamos decir que, computacionalmente hablando, siempre vamos a intentar elegir aquellos algoritmos cuyo orden de eficiencia sea menor, es decir, la función de eficiencia crece más lentamente.

Ejemplo

Supongamos que el algoritmo 1 tiene orden $\mathcal{O}(n^2)$ y el algoritmo 2 $\mathcal{O}(2^n)$. En este caso eligiremos el algoritmo 1, pues su orden de eficiencia es menor (después explicaremos por qué).

Representación

Como ya hemos dicho anteriormante, el orden de una función (en nuestro caso lo usaremos para el orden de eficiencia) se expresa mediante la notación O-grande o $\mathcal{O}(f)$, donde f es una función que determina el orden.

Ejemplos

$\mathcal{O}(n)$, $\mathcal{O}(2^n)$, $\mathcal{O}(n^4)$, $\mathcal{O}(\log_2(n))$, $\mathcal{O}(n\log_2(n))$ ...

Función de orden

Las funciones de orden no son como las funciones que conocemos, sino más bien una simplificación de las mismas, donde solo nos interesa cómo crecen. Vamos a poner un ejemplo:

Ejemplo

Si tenemos que $f(x) = 3x^4 + 5x^2 + x + 4$, entonces el orden de esta función se corresponde con el término dominante, es decir, el que crece más rápido, sin constantes, es decir, el orden de $f(x)$ es $\mathcal{O}(n^4)$.

¿Qué funciones tienen mayor orden?

Como es natural, es necesario poder ordenar dichas funciones para así decidir cuál es el término que crece más rápido.

Definición

Dadas dos funciones $f(x)$ y $g(x)$, diremos que $\mathcal{O}(g) < \mathcal{O}(f)$ si $\nexists r \in \mathbb{R}$ tal que $f(x) < r \cdot g(x)\ \forall x \in \mathbb{R}$

Ejemplos

Vamos a poner un ejemplo.

Ejemplo

$f(x) = 2^x$, $g(x) = 3^x$. Es evidente que $\mathcal{O}(g) < \mathcal{O}(f)$ porque $3^x < r \cdot 2^x \Rightarrow (\frac{3}{2})^x < r$, pero

$$\lim_{x \to +\infty} (\tfrac{3}{2})^x = \infty$$

luego no existe ninguna constante r que acote ese límite.

Cuidado

Si tenemos $f(x) = x^2$ y $g(x) = 1000000x^2$, $\mathcal{O}(g) = \mathcal{O}(f)$ por muy grandes que sean las constantes de $g(x)$.

Operaciones

Una vez conocida la notación, vamos a ver las dos operaciones básicas:

- Suma: $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(\max(f, g))$
- Producto: $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$

Ejemplos

$\mathcal{O}(n) + \mathcal{O}(\log_2(n)) = \mathcal{O}(n)$
$\mathcal{O}(n) \cdot \mathcal{O}(2^n) = \mathcal{O}(n \cdot 2^n)$

Ya tenemos todo lo necesario para calcular algunas eficiencias.

Código Sin Bucles

Consideremos este código:

```
#include <iostream>
int main(){
  int a, b=0, aux;
  std::cin >> a;
  aux = a;
  a = b;
  b = aux;

  std::cout << a << b << std::endl;
}
```

Solución Código Sin Bucles

Cada instrucción se considera $\mathcal{O}(1)$:

```
#include <iostream>
int main(){
  int a, b=0, aux;   //------------------------------>O(1)
  std::cin >> a;   //------------------------------>O(1)
  aux = a;   //------------------------------>O(1)
  a = b;   //------------------------------>O(1)
  b = aux;   //------------------------------>O(1)

  std::cout << a << b << std::endl;   //-->O(1)
}
```

Sumamos todos los órdenes y listo, luego el orden de este código sería $\mathcal{O}(1)$.

Código Con Bucles

Consideremos este otro código:

```
#include <iostream>
int main(){
  int n;
  std::cin >> n;
  int* v = new int[n];

  for (int i=0; i<n; ++i){
    v[i] = i;
  }
}
```

Solución Código Con Bucles

La eficiencia de un bucle es la eficiencia de lo que hay dentro por el número de iteraciones del bucle:

```
#include <iostream>
int main(){
  int n;  //------------------------------------>O(1)
  std::cin >> n;  //---------------------------->O(1)
  int* v = new int[n];  //--------------------->O(1)

  for (int i=0; i<n; ++i){//        |\
    v[i] = i;  //-------------->O(1) |----->O(1)*n = O(n)
  }//                                |/
}
```

Sumamos todos los órdenes y listo, luego el orden de este código sería $\mathcal{O}(n)$.

Código Con Recursividad

Consideremos este otro código:

```
int Bar(int* array, int size, int ini, int fin){
  if(n>1){
    Bar(array, size-1, ini, fin-1);  //Llamada a todos
        menos al último
    Bar(array, size-1, ini+1, fin);  //Llamada a todos
        menos al primero
  }
}
```

Solución Código Con Recursividad

La eficiencia para la recursividad ha de resolverse con una ecuación recurrente:

```
int Bar(int* array, int size, int ini, int fin){
  if(n>1){
    Bar(array, size-1, ini, fin-1);   //------->O(?)
    Bar(array, size-1, ini+1, fin);   //------->O(?)
  }
}
```

Para plantear esta ecuación, solo nos vamos a centrar en las dos llamadas a la funciones Bar.

Ecuación Recurrente

Suponiendo n el tamaño del vector que se pasa y $T(n)$ la función que determina la eficiencia, podemos plantearla de la siguiente manera:

Ecuación

$$T(n) = \begin{cases} 1 & if \quad n = 1 \\ 2T(n-1) & if \quad n > 1 \end{cases}$$

La técnica que vamos a seguir para este caso va a ser expandir la ecuación e ir deduciendo cosas. Hay otros métodos según la ecuación a resolver, pero por el momento nos conformamos con este.

Solución Ecuación Recurrente

Vamos expandiendo la ecuación:

$$T(n) = 2T(n-1) = 2 \cdot (2T(n-2)) = 2^2 \cdot T(n-2) =$$
$$= 2^2 \cdot (2T(n-3)) = 2^3 \cdot T(n-3) = \ldots$$

Como podemos ver, al expandir aparece cierto patrón, que en general se tiene $T(n) = 2^i \cdot T(n-i)$. Vamos a tomar $i = n-1$:

$$T(n) = 2^{n-1} \cdot T(n-(n-1)) = 2^{n-1} \cdot T(1) = 2^{n-1}$$

Luego $T(n) = 2^{n-1}$, lo que implica que el orden de dicho algoritmo es $\mathcal{O}(2^n)$ (no $\mathcal{O}(2^{n-1})$).

Categorías

Y empezamos con la parte que realmente nos interesa, los algoritmos de ordenación y sus técnicas. Los vamos a dividir como sigue:

- Algoritmos cuadráticos
  - Burbuja
  - Selección
  - Inserción
- Algoritmos Divide y Vencerás
  - Mergesort
  - Quicksort
  - Bitonicsort

Categorías

- Algoritmos de Ordenación por Conjuntos
  - RadixsortLSD
  - RadixsortMSD
  - Countingsort
- Algoritmos Varios
  - Heapsort
  - Timsort
  - Slowsort
  - Permutationsort
  - Bogosort

Introducción

¿Cuáles son?

Lo algoritmos cuadráticos son aquellos cuya eficiencia es $\mathcal{O}(n^2)$.

¿Qué utilidad tienen?

Realmente estos algoritmos tienen poca utilidad práctica, pues son extremadamente lentos comparados con otros que veremos más adelante. Aún así alguno se comporta muy bien con muy pocos elementos o con vectores casi ordenados.

Burbuja

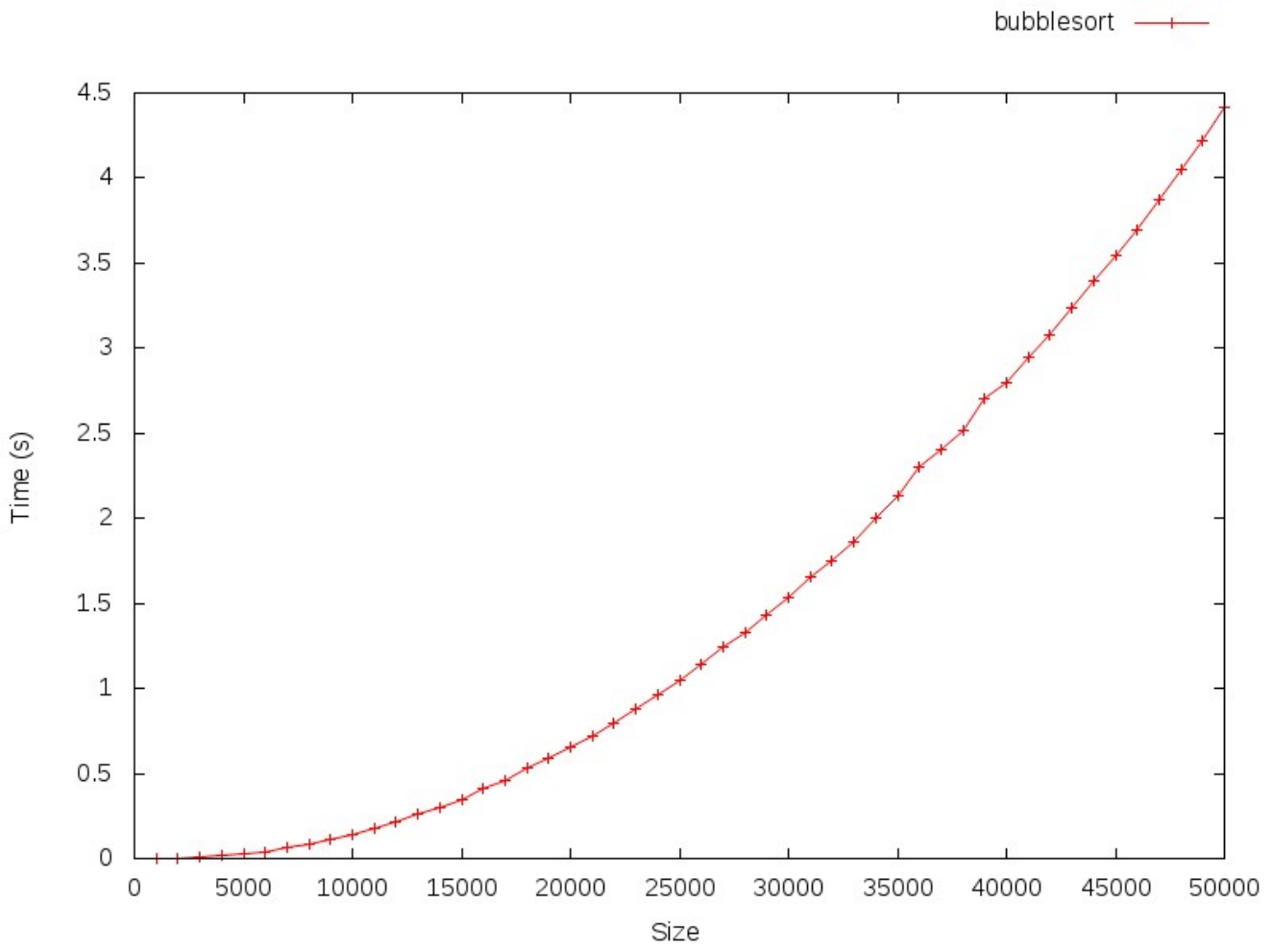
Este algoritmo consiste básicamente en ir trasladando el máximo de los elementos que aún no están ordenados al principio de los que sí están ordenados.

Podríamos decir que este es el peor de los algoritmos cuadráticos y que solo es útil para fines académicos.

Eficiencia

En el mejor caso, cuando el primer elemento es el mayor de todos y los demás están ordenados, es $\mathcal{O}(n)$; en el caso peor y promedio es $\mathcal{O}(n^2)$

Gráfica


bubblesort
Time (s)
Size
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000

Selección

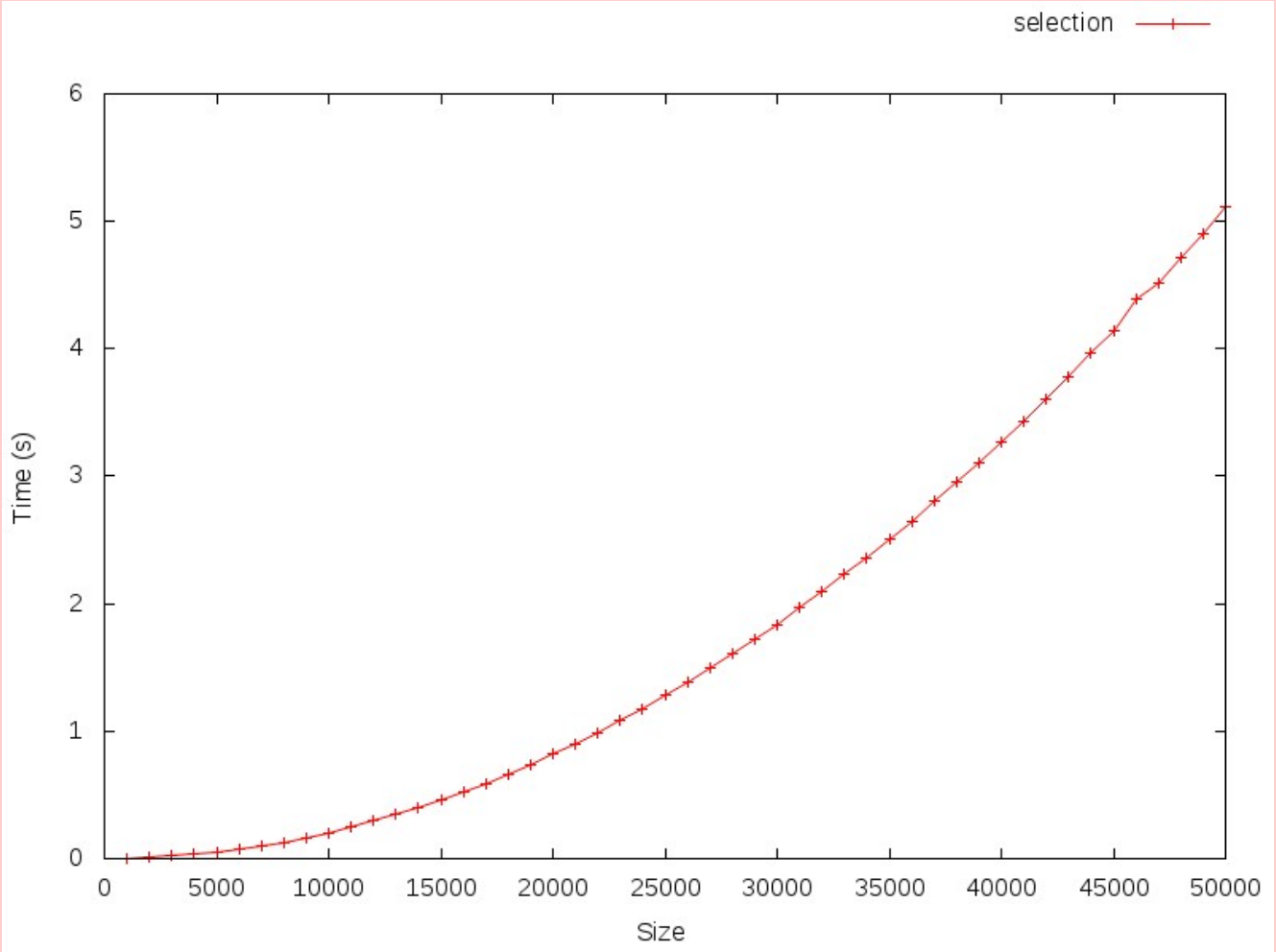
Este algoritmo es parecido al de Burbuja pero al revés, es decir, busca el mínimo de los que no están ordenados y lo inserta al final de los ordenados.

Tampoco es una maravilla comparado con el Burbuja (de hecho en las gráficas es peor que el burbuja).

Eficiencia

En el caso mejor, peor y promedio es $\mathcal{O}(n^2)$.

Gráfica


selection
Time (s)
Size
0
1
2
3
4
5
6
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000

Inserción

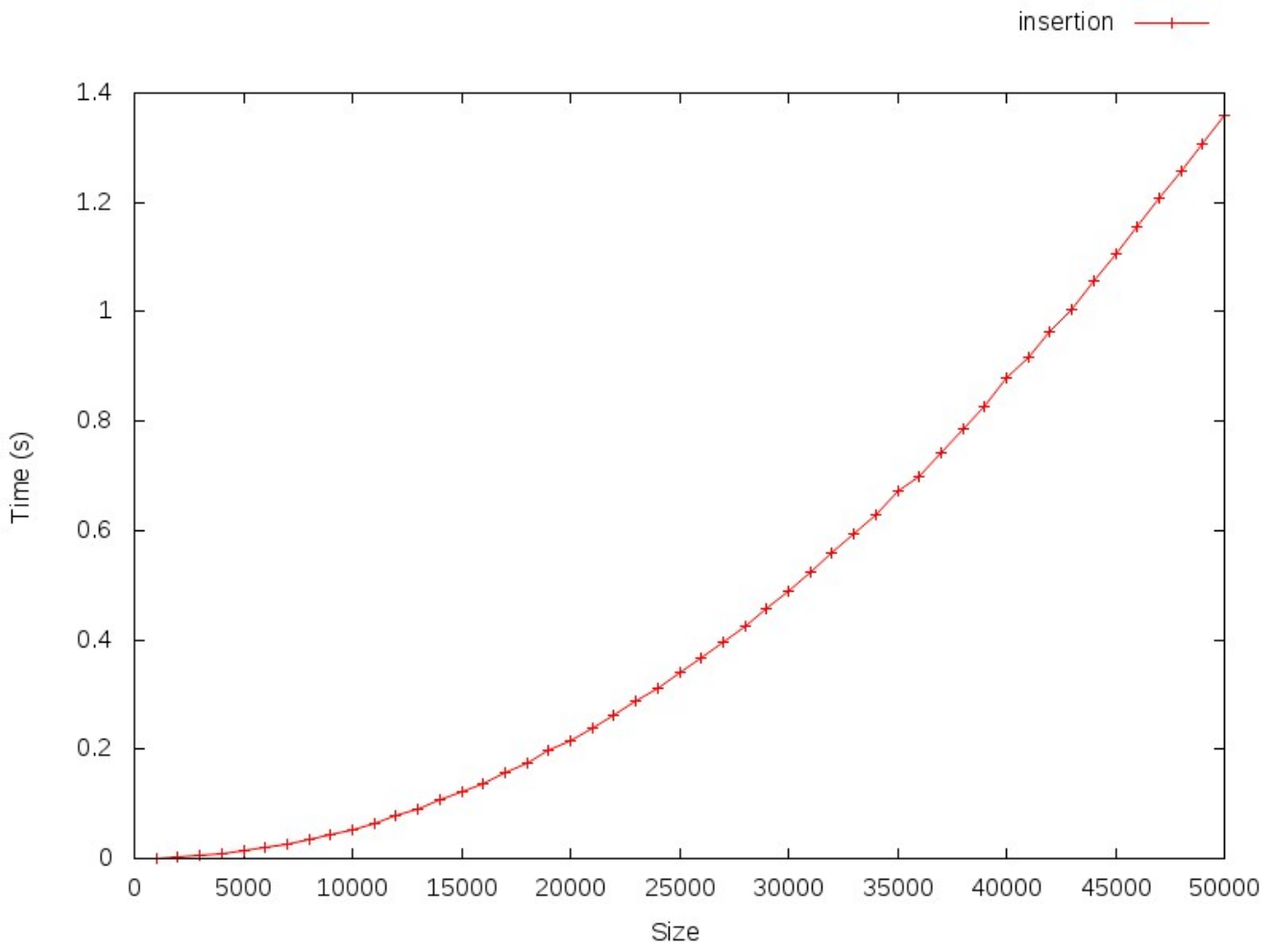
Este algoritmo consiste en ir recorriendo el vector de los elementos que no están ordenados e insertarlo en su correspondiente sitio en los ordenados.

Este algoritmo, al contrario que los otros dos cuadráticos, sí que se usa en algunas situaciones. Por ejemplo, con pocos elementos tiene mejores resultados que algunos de los más rápidos como el Quicksort, por lo que a partir de cierto umbral se usa este.

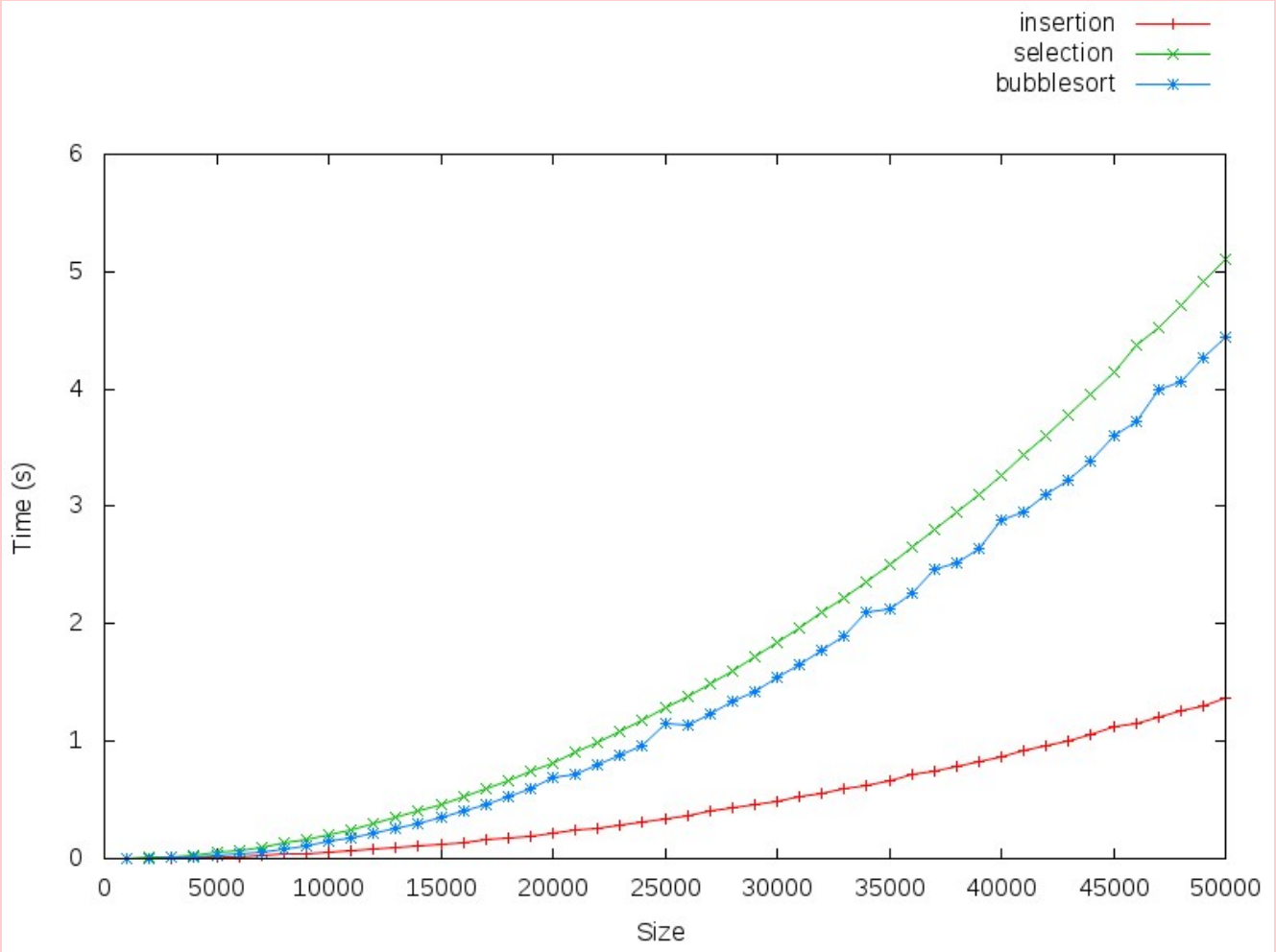
Eficiencia

En el caso mejor es $\mathcal{O}(1)$, y en el peor y promedio es $\mathcal{O}(n^2)$, aunque en algunas situaciones concretas tiende a tener una eficiencia $\mathcal{O}(n)$.

Gráfica


insertion
Time (s)
Size
0
0.2
0.4
0.6
0.8
1
1.2
1.4
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000

Comparación Cuadráticos



Introducción

¿Cuáles son?

Estos algoritmos son aquellos que siguen la filosofía de "Divide y Vencerás", la cual explicaremos a continuación. Suelen tener eficiencia más o menos próxima a $\mathcal{O}(n \cdot \log_2(n))$.

¿Qué utilidad tienen?

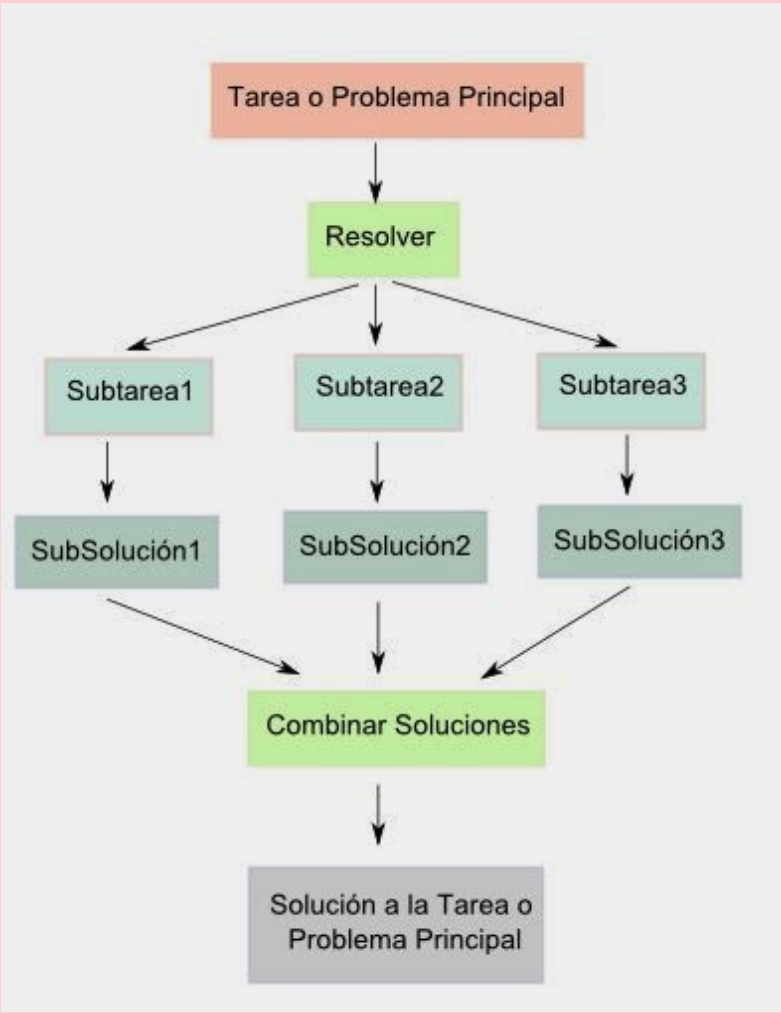
Son los algoritmos que se utilizan en los sistemas reales, pues son muy rápidos y pueden trabajar con cualquier tipo de datos, ya sean palabras, números, letras o cualquier cosa que pueda ser comparada.

Filosofía

¿En qué consiste?

Esta técnica consiste en dividir el problema total en problemas más pequeños y fáciles de resolver.

Una vez se han resuelto estos problemas, se unen las soluciones para generar la solución al problema total. Podemos verlo en el esquema de la imagen.



Mergesort

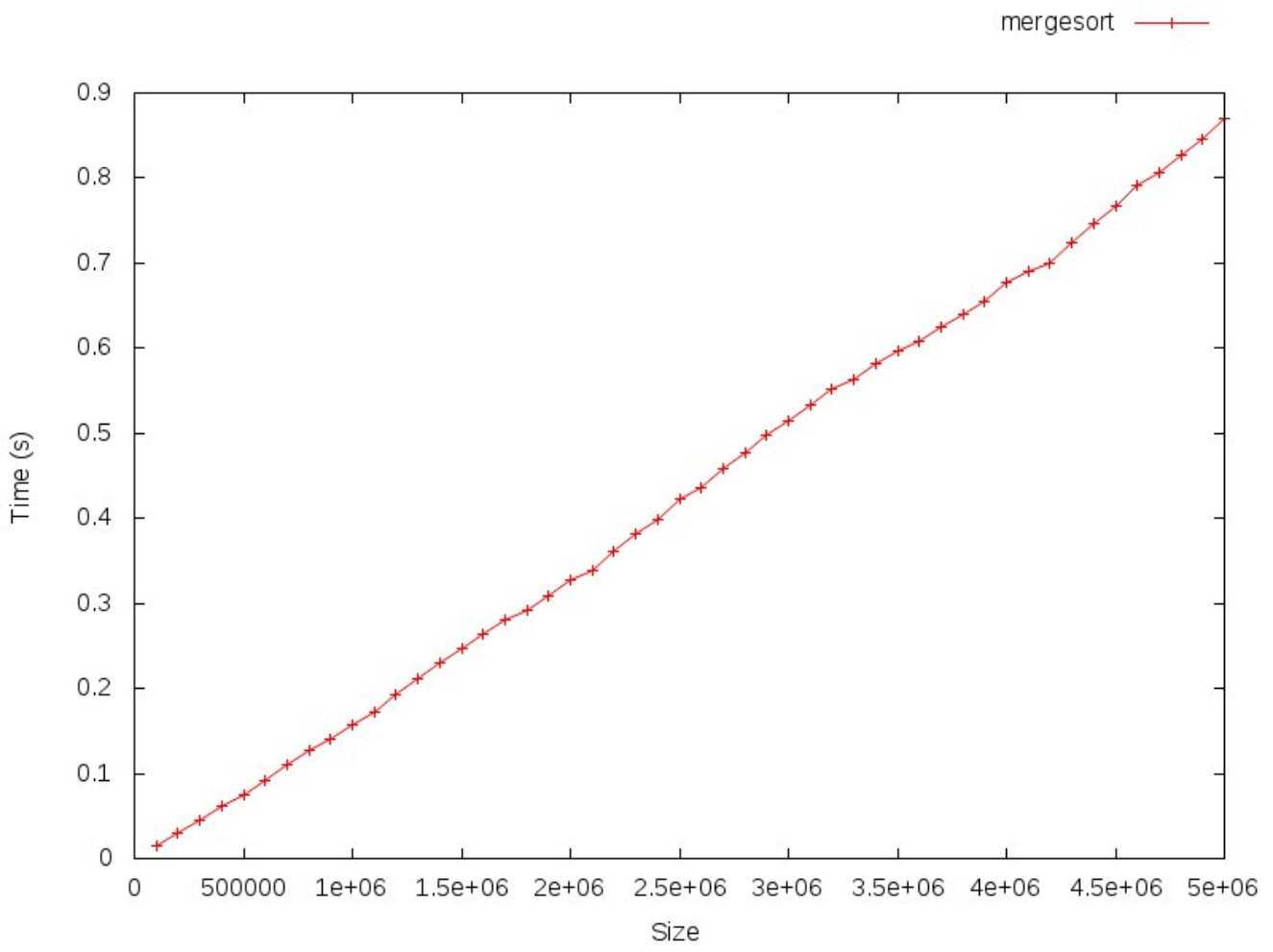
El primer algoritmo que vamos a tratar va a ser el Mergesort. Este algoritmo divide el vector en dos partes de igual tamaño. Esto lo hace recursivamente. Podemos decir que esta es la parte de división del trabajo.

Una vez llegado el caso base, se van uniendo las mitades (estas ya ordenadas) con una función de fusión (la cual ahora explicaré) de manera que la parte resultante queda ordenada.

Eficiencia

La eficiencia de este algoritmo es siempre $\mathcal{O}(n \cdot \log_2(n))$, aunque con pocos elementos tal vez convenga más usar algún otro algoritmo de ordenación (como el inserción).

Gráfica


mergesort
Time (s)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
4.5e+06
5e+06
Size

Quicksort

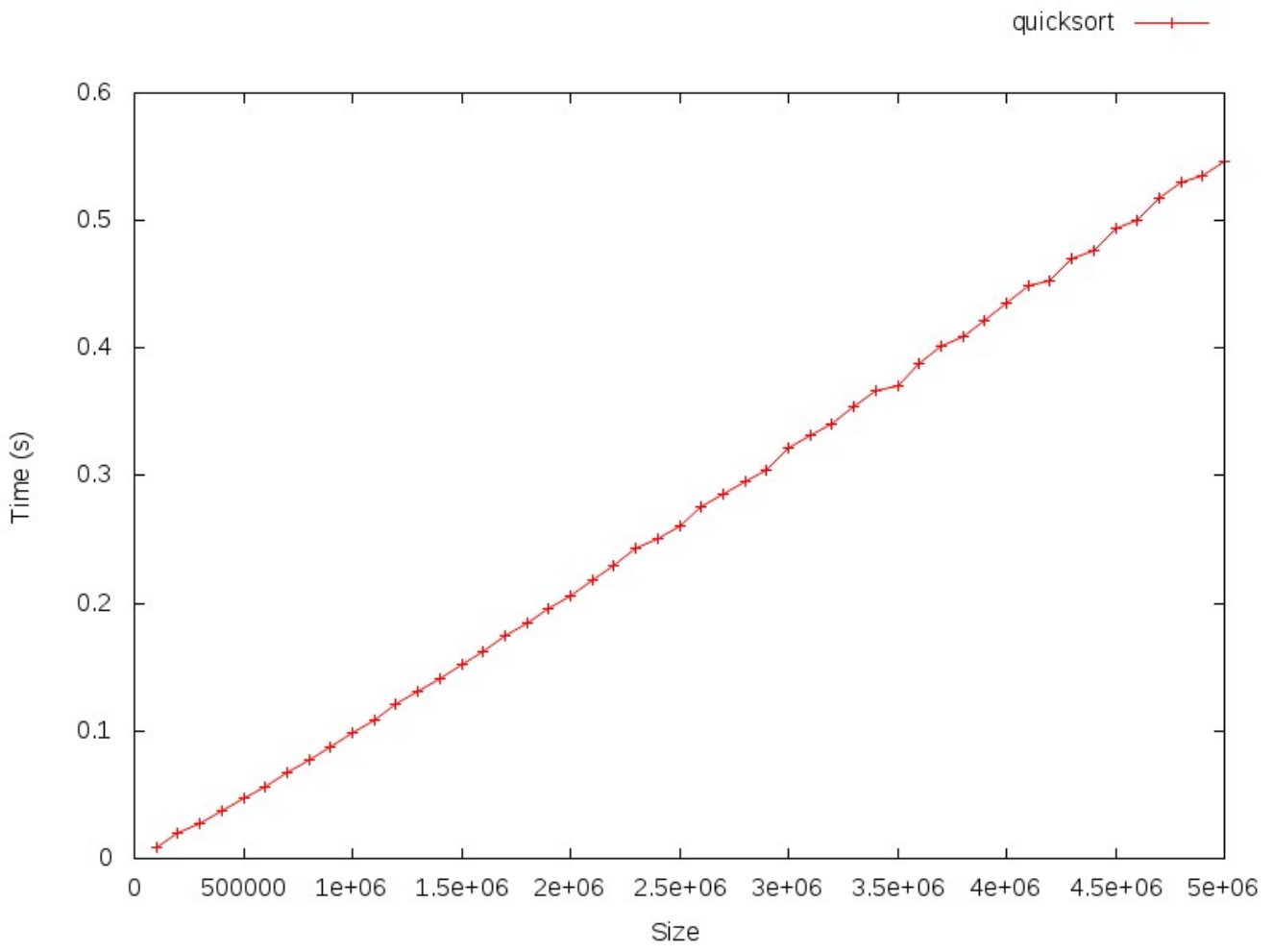
Llegamos al que posiblemente sea el algoritmo más usado de todos. Este algoritmo sigue estos pasos:

- Selecciona un pivote, esto es, un elemento al azar del vector.
- Cambia elementos de manera que a la izquierda del pivote todos son menores que el pivote, y a la derecha mayores.
- Aplica Quicksort sobre ambas particiones del vector

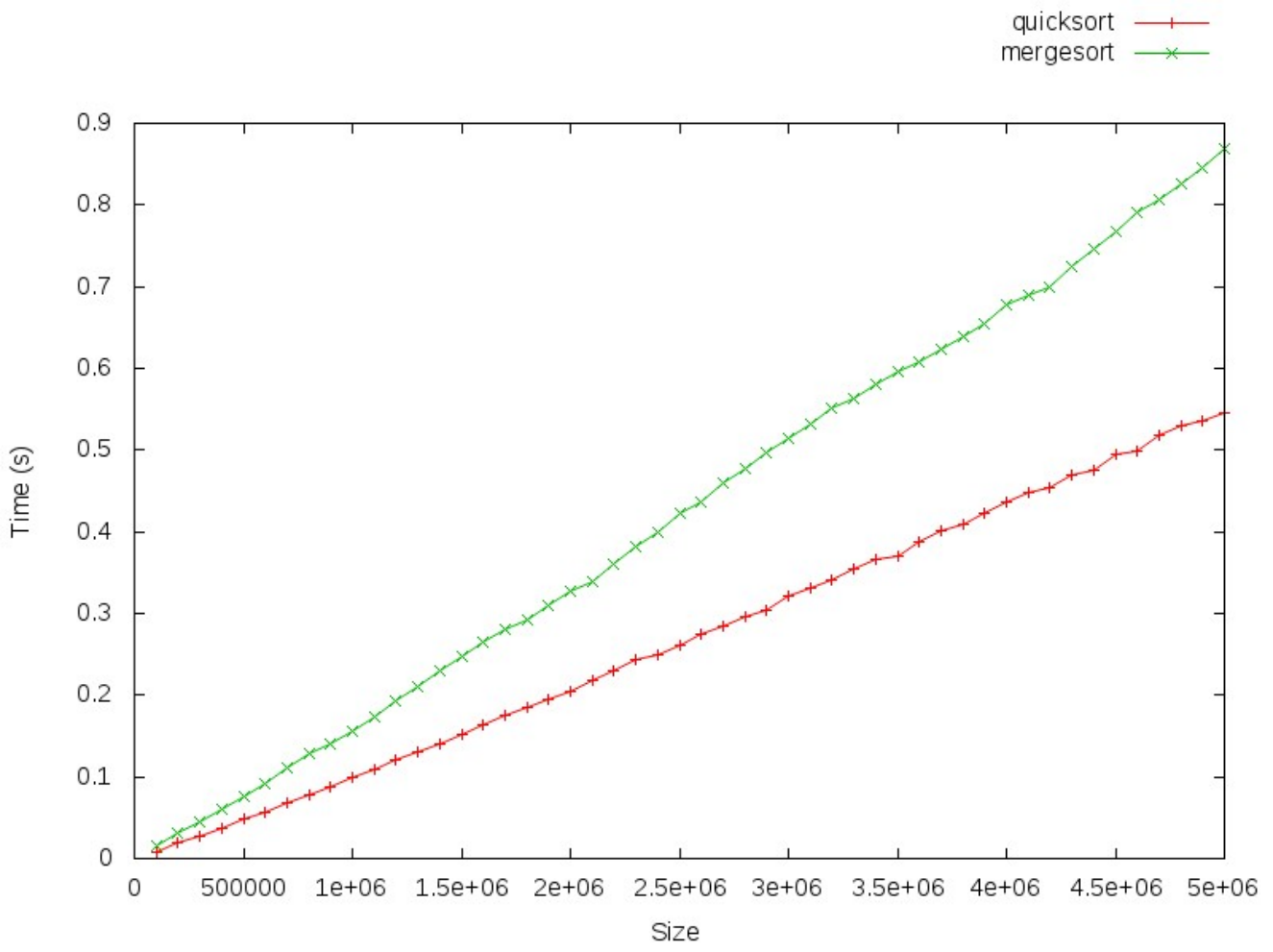
Eficiencia

Si hay que ponerle un pero al algoritmo es lo aleatorio de la partición. En el mejor caso y en el promedio tiene eficiencia $\mathcal{O}(n \cdot \log_2(n))$, pero en el peor caso nos vamos a la horrible eficiencia de $\mathcal{O}(n^2)$, que se dará cuando siempre que se elija un pivote, sea el menor o mayor del vector.

Gráfica


quicksort
Time (s)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
4.5e+06
5e+06
Size

Comparación Mergesort-Quicksort



Bitonicsort

Ahora hablaremos de un algoritmo un tanto desconocido, pero que sin embargo es muy usado en computación paralela debido a la facilidad de paralelizar las divisiones.

Es parecido al Mergesort, pero este algoritmo hace la fusión de una manera que permite la paralelización. Consiste en:

- Hace la división en mitades, igual que el mergesort.
- Cada mitad se ordena diferente: una creciente y otra decreciente, formando lo que se conoce como una secuencia bitónica.
- Se aplica la función de fusión, al igual que con el mergesort, con la particularidad de que esta sí que puede ser paralelizada.

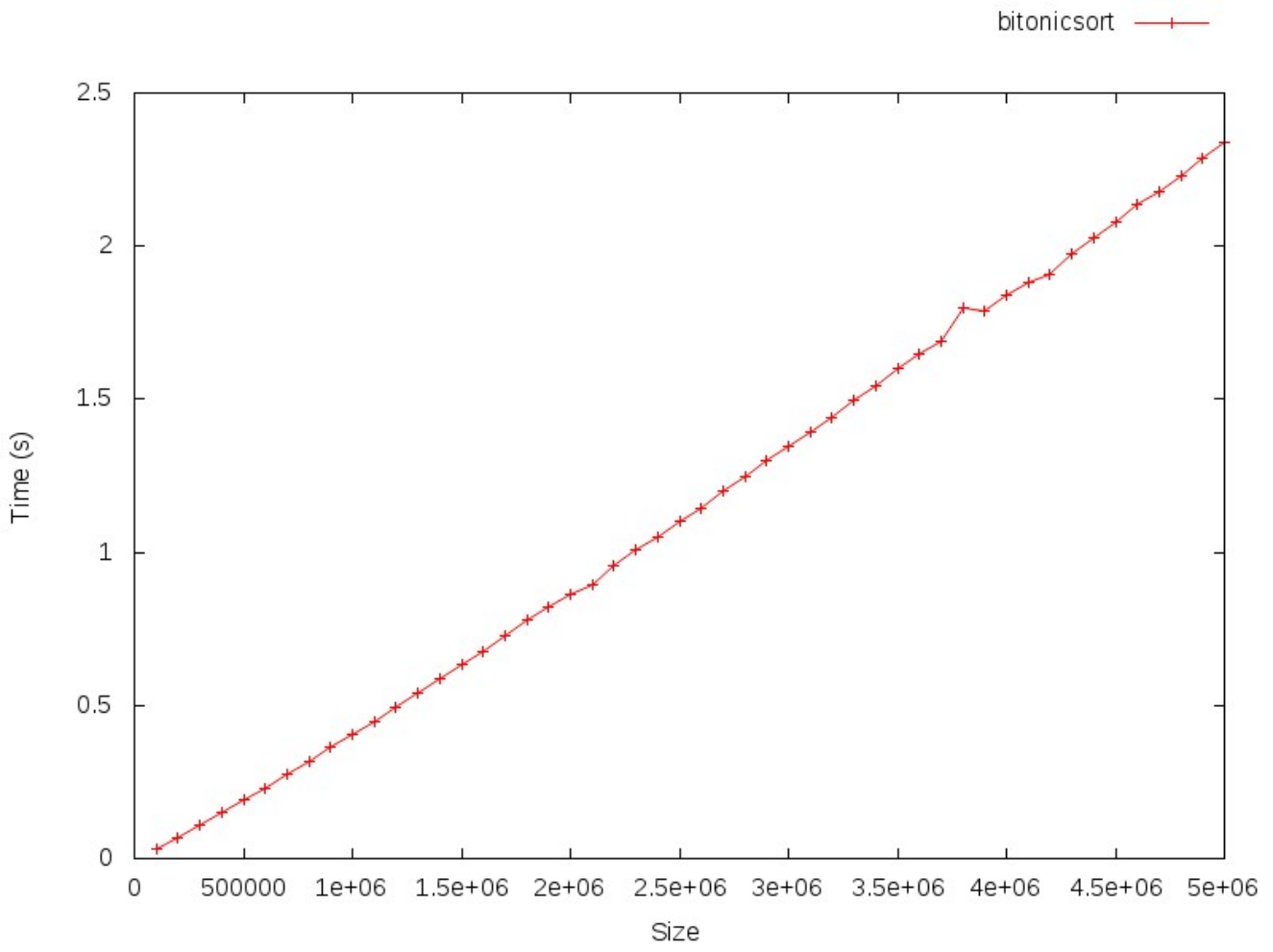
Bitonicsort

Puesto que es un algoritmo algo difícil de comprender, se explicará la idea para $n = 2^k$ datos (para un número arbitrario de datos se usa una ligera modificación de esta idea, pero más pulida).

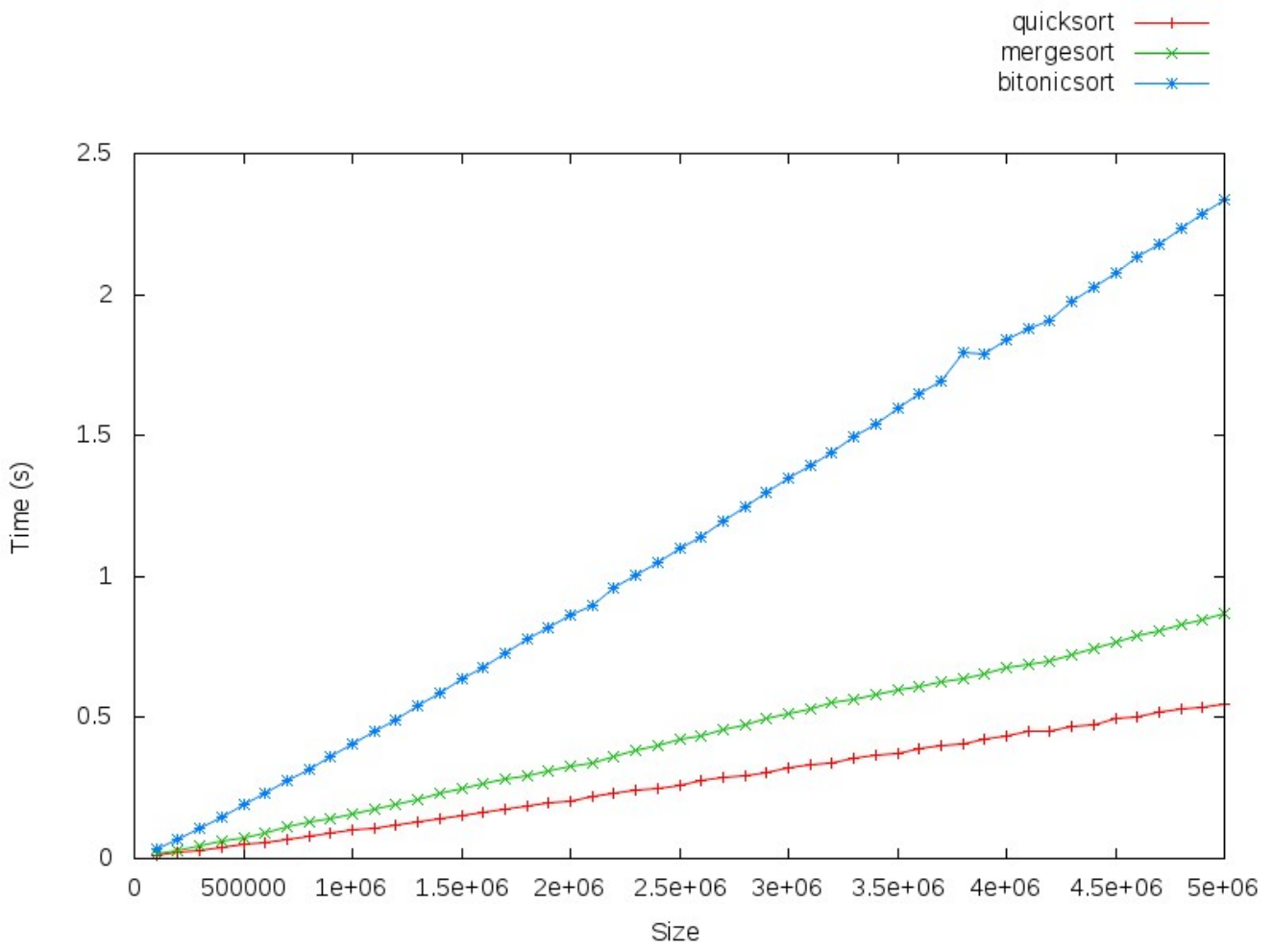
Eficiencia

En el caso secuencial (sin paralelizar), este algoritmo tendrá una eficiencia media de $\mathcal{O}(n \cdot \log_2(n)^2)$, pero como normalmente se usa en paralelo, la eficiencia desciende según el número de procesadores, generalizando a que, suponiendo que hay n procesadores, la eficiencia sería $\mathcal{O}(log_2(n)^2)$, aunque en la práctica realmente sea $\mathcal{O}((\frac{n}{k})log_2(n)^2)$, siendo k el número de procesadores.

Gráfica


bitonicsort
Time (s)
Size
0
0.5
1
1.5
2
2.5
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
4.5e+06
5e+06

Comparación Divide y Vencerás



Introducción

¿Cuáles son?

Estos algoritmos son aquellos en los que se ordena por conjuntos o separando, de tal modo que no son necesarias comparaciones entre dos elementos cualesquiera.

¿Qué utilidad tienen?

Son los algoritmos más rápidos en casos concretos, llegando a alcanzar órdenes de eficiencia de $\mathcal{O}(n)$. Esto es debido a que se aprovechan ciertas propiedades de los datos (como que estos sean números enteros o que no sean negativos) para idear algoritmos pensados para tratar esos tipos de datos.

Radixsort

El Radixsort es un algoritmo que ordena según los dígitos de los datos (es decir, enteros no negativos). Tiene dos variantes

- Dígito Menos Significativo (LSD): Va ordenando empezando por los dígitos menos significativos (unidades, decenas...) hacia los más significativos (millón, decena de millón...).
- Dígito Más Significativo (MSD): Justo al revés que el anterior, solo que para este caso es necesaria la recursividad y para el anterior no.

Eficiencia

La eficiencia de este algoritmo depende del dato que se esté ordenando. Suponiendo que el tamaño del dato en bits es w, la eficiencia sería $\mathcal{O}(w \cdot n)$. Puesto que el w es constante a lo largo del algoritmo, podemos asumir que el orden es $\mathcal{O}(n)$, aunque con unas constantes ocultas un tanto altas.

Radixsort

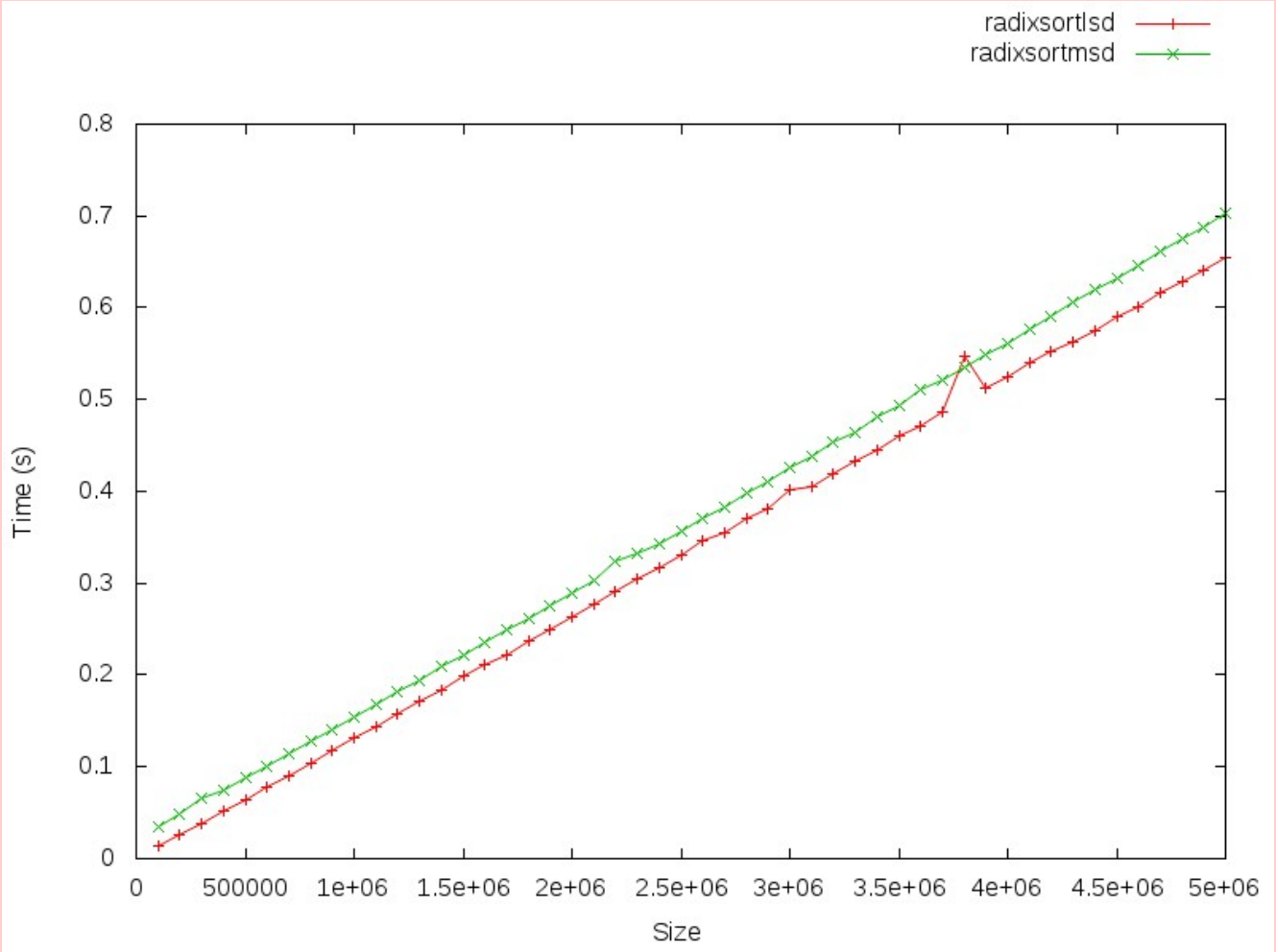
La idea general del LSD es ir recorriendo el vector e ir metiendo los datos en conjuntos diferentes según el dígito que se esté comprobando (si es 1 va al conjunto A, si es 2 va al conjunto B...) y luego sacar todos los del conjunto A primero en el mismo orden que han entrado, después los del B y así hasta el último conjunto. Al repetir esto con dígitos cada vez más significativos, acabaremos con un vector ordenado.

Para el MSD hacemos lo mismo solo que empezando por el dígito más significativo, y luego llamamos recursivamente en cada conjunto por separado.

Nota

Tal vez interese usar colas como conjuntos...

Comparación LSD-MSD



Countingsort

Este, al menos en los que hay implementados, es el más rápido de todos con números enteros y por diferencia. La idea es sumamente simple, aunque muy restringida a que sean enteros (si no lo son, el algoritmo falla de mala manera, cosa que puede no pasarle al radixsort).

Consiste en crear un vector auxiliar de tamaño igual a la diferencia entre el mayor y el menor del vector dado(notar que si hay mucha diferencia, el algoritmo puede no ser muy eficiente). Suponiendo que el mínimo es 1 y el máximo 10, nuestro vector auxiliar tendrá tamaño 10. Lo que hacemos es, en la posición x del vector meter la cantidad de veces que aparece ese mismo número en el vector dado, y luego volcar el vector auxiliar en el original.

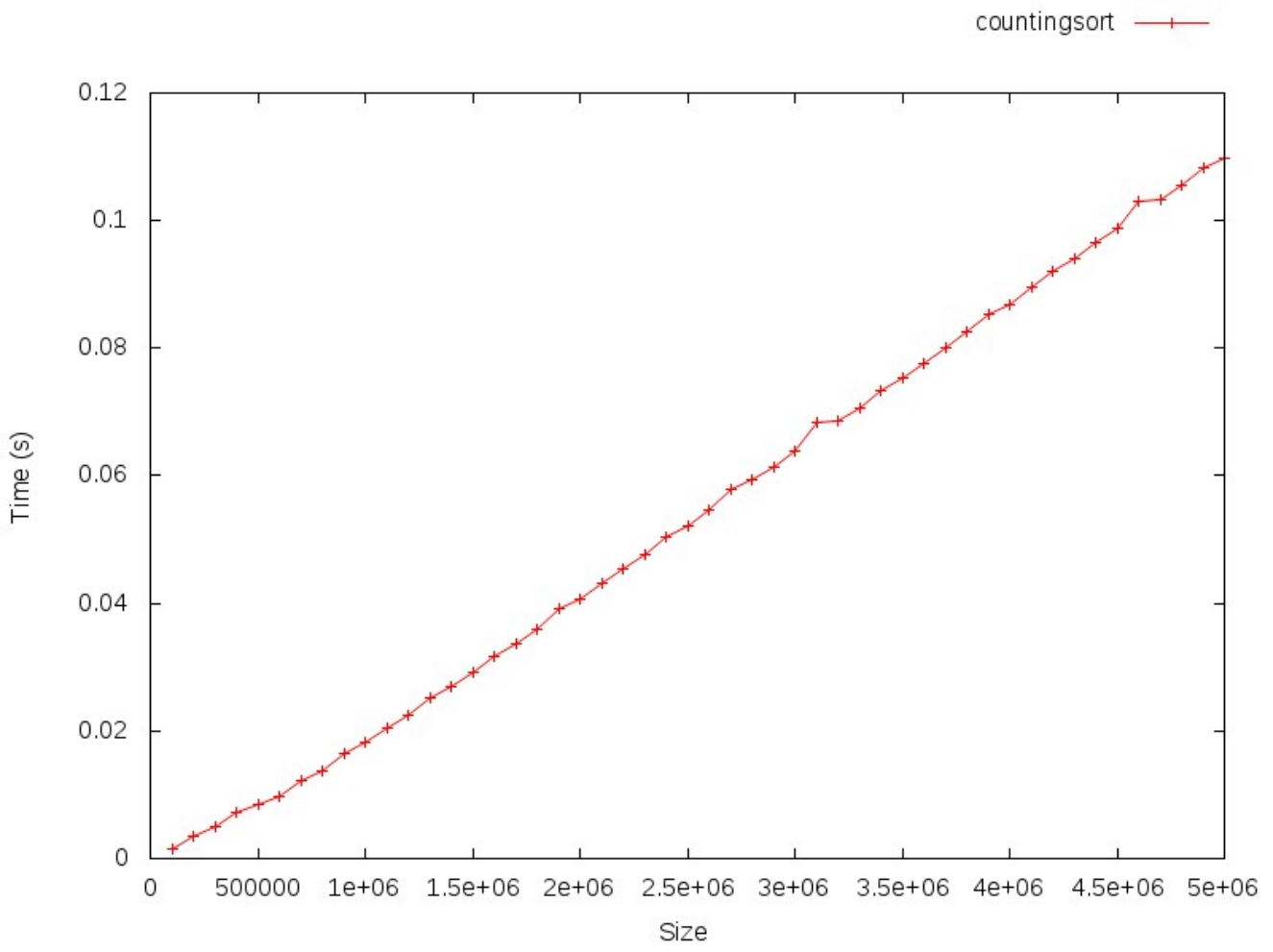
Countingsort

Eficiencia

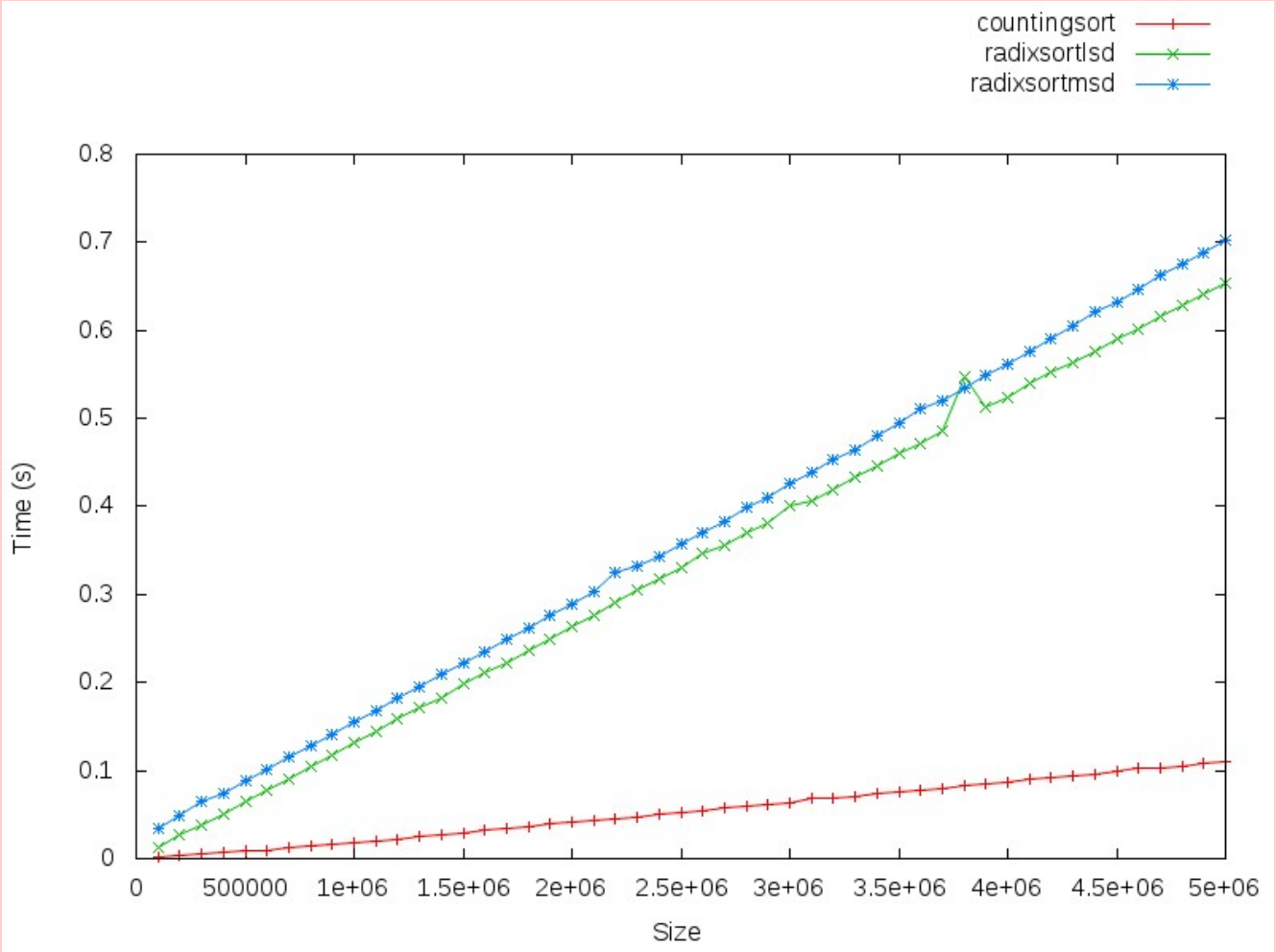
La eficiencia de este algoritmo es en general $\mathcal{O}(n)$ para una distribución más o menos uniforme. Sin embargo, puede dar resultados muy malos si, por ejemplo, tenemos solo 10 datos a ordenar pero el menor es 1 y el mayor 1000000, en cuyo caso pierde mucha velocidad con respecto al número de datos.

En la gráfica comparativa de los tres algoritmos de ordenación por conjuntos y en la final se verá claramente que este es el más rápido y con diferencia

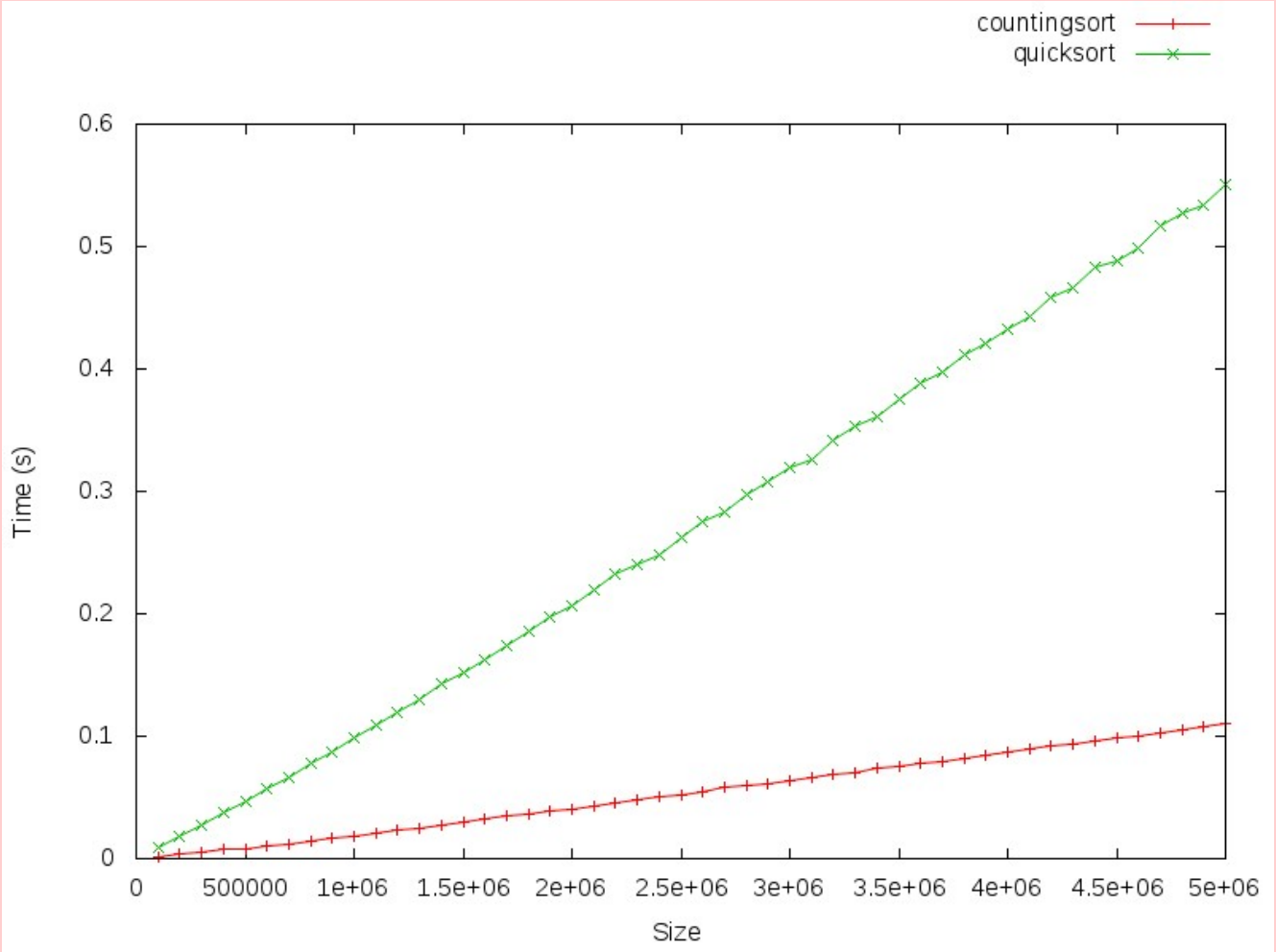
Gráfica


countingsort
Time (s)
0.12
0.1
0.08
0.06
0.04
0.02
0
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
4.5e+06
5e+06
Size

Comparación Ordenación por Conjuntos



Comparación Countingsort-Quicksort



Introducción

En este apartado vamos a tratar algunos algoritmos que no encajan muy bien en las otras categorías, que no están implementados, o que simplemente son una chorrada que no merece mucho la pena prestarle atención.

Nota

No hay gráficas de todos, solo de algunos.

Heapsort

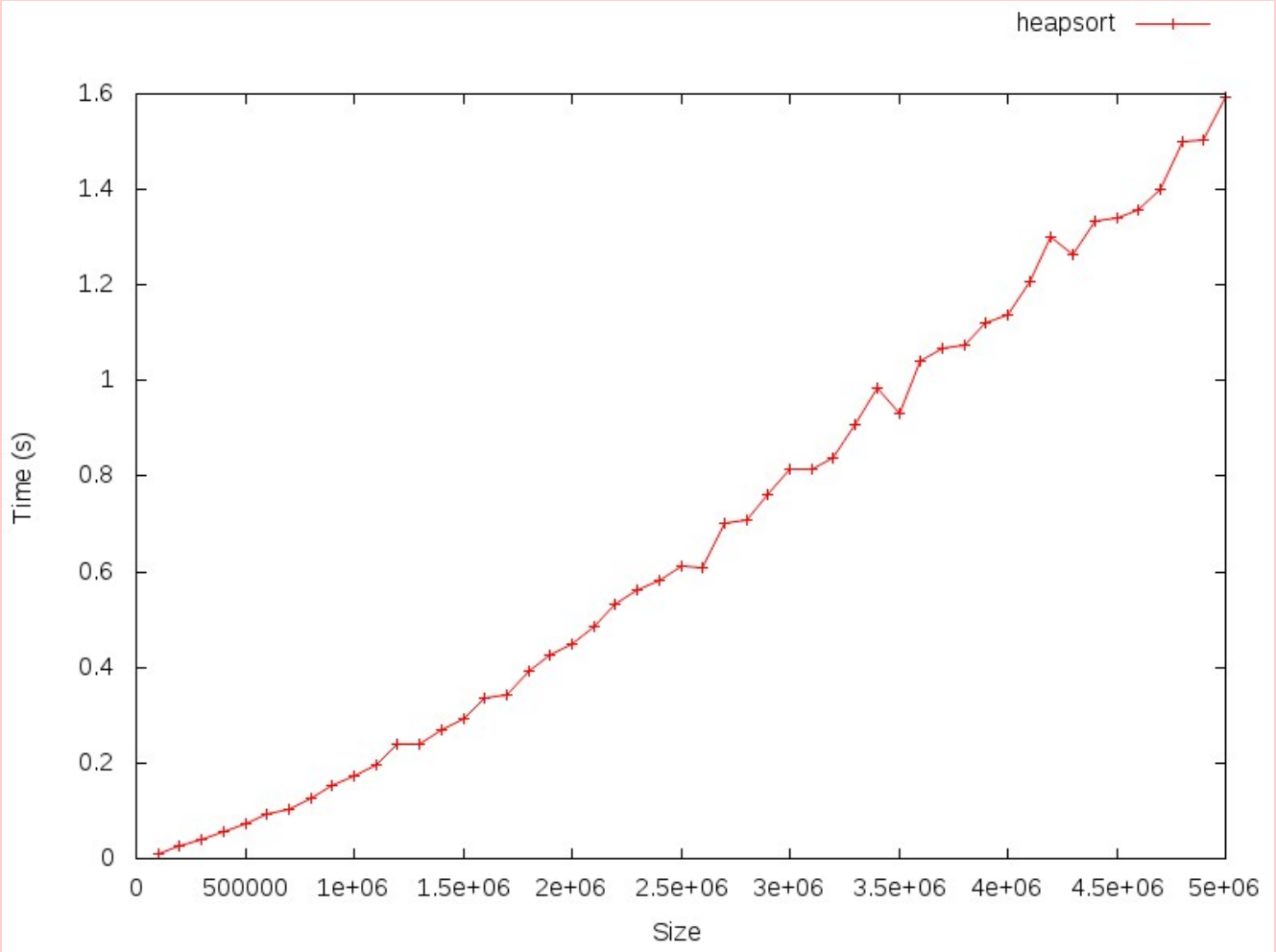
Vamos a empezar con el Heapsort, un algoritmo sumamente potente por el hecho de que trabaja muy bien con cualquier tipo de dato y es muy estable, a costa de algo menos de eficiencia (todo lo contrario que el Countingsort por ejemplo).

La idea es meter todos los datos en un árbol binario parcialmente ordenado, que al volcarlo en el vector, estará ordenado. Implementar un árbol es tedioso, y explicarlo requiere de muchas horas, pero la STL puede ayudarnos en esta tarea proporcionándonos la priority_queue (nos hará todo el trabajo sucio).

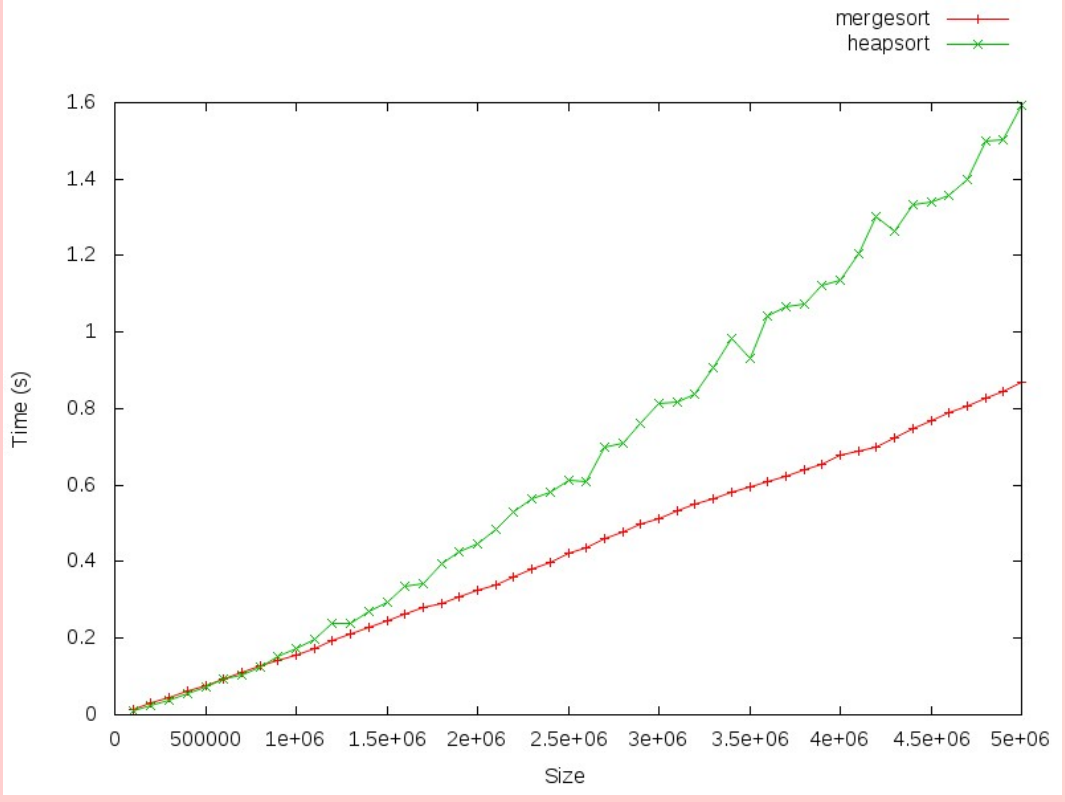
Eficiencia

La eficiencia de este algoritmo es siempre $\mathcal{O}(n \cdot \log_2(n))$, pues la inserción en el árbol es $\mathcal{O}(log_2(n))$ y hacemos n inserciones.

Gráfica


heapsort
Time (s)
Size
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
4.5e+06
5e+06

Comparación Mergesort-Heapsort



Como vemos, en rapidez no es mejor que el Mergesort.

Timsort

Se trata del algoritmo de ordenación que usan Python y OpenJDK.

Es un algoritmo híbrido que hace uso del Mergesort y del Inserción para funcionar. Consiste básicamente en buscar sub-secuencias ordenadas (ya sea ascendente o descendentemente) y sabiendo eso, ordena el resto más eficientemente, usando Insertion cuando se pasa un umbral (normalmente 64). Una vez ordenadas esas sub-secuencias, se aplica una fusión como la del Mergesort para ir obteniendo secuencias más grandes ordenadas.

Con secuencias aleatorias suele ser peor, pero con secuencias de datos reales, el número de sub-secuencias ordenadas tiende a ser mayor, y es lo que aprovecha Timsort.

Timsort

Eficiencia

La eficiencia de este algoritmo es $\mathcal{O}(n \cdot \log_2(n))$, con mejores resultados en vectores parcialmente ordenados.

Una implementación de este algoritmo puede encontrarse en github.com/gfx/cpp-TimSort.

Como se puede comprobar, no es algo precisamente trivial y requiere de muchas horas implementar algo así.

Slowsort

El nombre lo dice todo. Este algoritmo es más una curiosidad que otra cosa, una oda a como coger el camino más largo para resolver un problema.

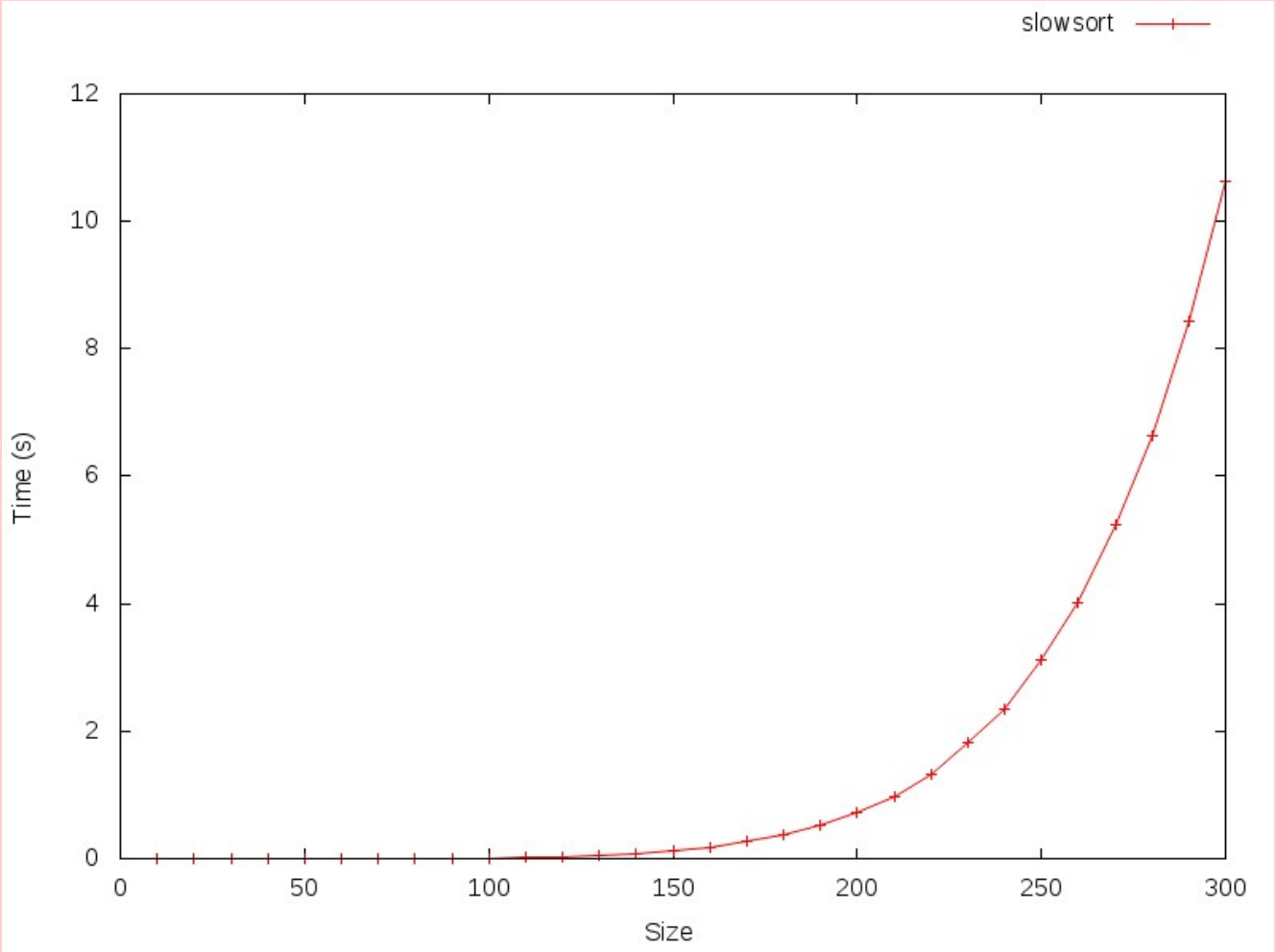
Básicamente consiste en ordenar las dos mitades del vector, quedarse con el mayor elemento de ambas mitades (el que está en la posición límite) y dejarlo al final del vector. Hecho esto, llamamos a Slowsort con todos los elementos menos el último. Simplemente absurdo.

Eficiencia

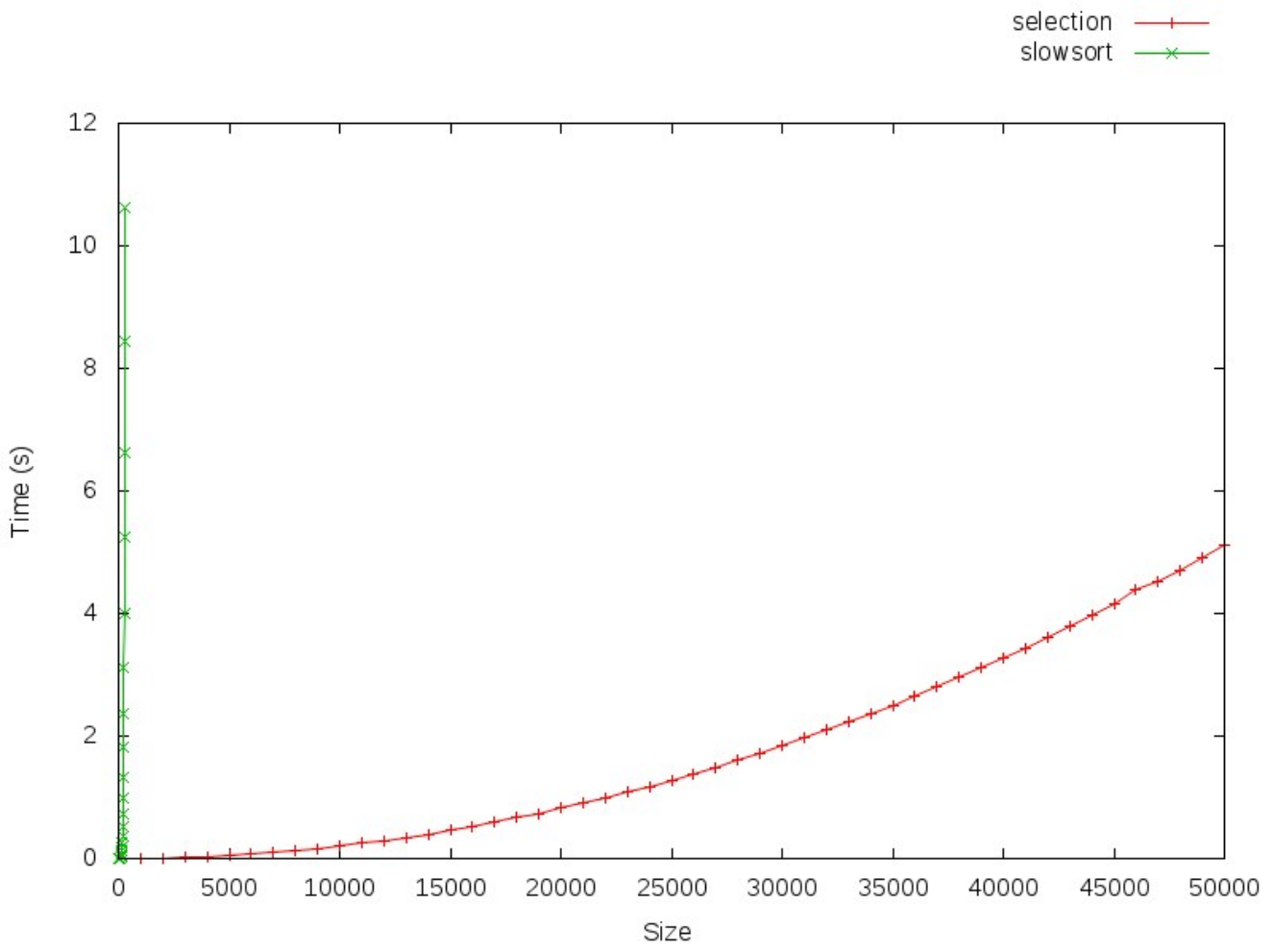
La eficiencia de este algoritmo es bastante compleja de calcular. De hecho no hay un orden exacto, aunque por aproximación se ha llegado a acotar por $\mathcal{O}(n^{\log_2(n)})$, aunque no es exacto.

Ahora veremos en las gráficas por qué es tan malo.

Gráfica



Comparativa Slowsort-Selección



Permutationsort

Este algoritmo es muy simple de entender. Busca en todas las permutaciones posibles del vector (sin repetir permutaciones) hasta dar con la que esté ordenada.

Puede optimizarse un poco si se evitan recorridos que seguro no llevan a la solución, cosa que es totalmente prescindible.

Eficiencia

Un vector de n elementos tiene $n!$ permutaciones posibles. Dado que debemos comprobarlas todas y, por cada una, comprobar si está ordenada, la eficiencia se nos va a $\mathcal{O}(n \cdot n!)$, algo obviamente inviable para lo que ni hay gráfica (con 12 elementos tarda más de 10 segundos).

Bogosort

Este algoritmo, también llamado Randomsort o Monkeysort, es igual que el anterior solo que puede repetir permutaciones.

En efecto, genera una permutación aleatoria del vector y comprueba si está ordenada. En caso negativo, genera otra permutación (puede ser la misma o no).

Eficiencia

De media, la eficiencia es $\mathcal{O}(n \cdot n!)$, como el anterior, pero en el peor caso tiene eficiencia $\mathcal{O}(\infty)$, es decir, no asegura solución, por lo que ni podemos considerarlo un algoritmo.

Final

Muchas gracias a todos por asistir

Fin