# Medición de tiempos

Marco Antonio Gómez Martín

September 12, 2019

## Contents



## 1 Introducción

En esta práctica guiada vamos a aprender a generar gráficas de tiempos de ejecución de algoritmos. Utilizaremos como ejemplo dos algoritmos: el algoritmo de ordenación por selección y el algoritmo de ordenación de la librería de C++ (`sort`, que implementa una variante del *quicksort*).

Las gráficas tendrán en el eje X el tamaño del vector que se ha ordenado y en el eje Y el tiempo invertido en la ejecución. Al generar una única gráfica con ambas series de datos podremos fácilmente comparar la eficiencia de los dos algoritmos.

## 2 Proceso

El proceso requiere dos fases bien diferenciadas:

1. Generar los datos con los que crear las gráficas. Por cada algoritmo se generará un fichero de texto. Cada línea del fichero tiene dos números que representan un valor/punto en la gráfica: el primer número es el tamaño del vector con el que se ha ejecutado el algoritmo y el segundo el tiempo invertido para ese tamaño.

2. Utilizar esos dos ficheros de datos para generar la gráfica. Se puede utilizar Excel, importando los datos en una hoja y creando un gráfico o con alguna herramienta distinta. En este documento explicaremos cómo generar las gráficas utilizando *gnuplot*, una aplicación de software libre.

Por comodidad, en la primera fase el "*fichero*" es en realidad la salida estandar de la aplicación que puede luego redirigirse a un fichero de texto.

## 3 Medición de tiempos

Para medir el tiempo del algoritmo se necesita ejecutar varicas veces el algoritmo, con distintos tamaños. Dependiendo del algoritmo en cuestión los tamaños variarán. Si se están poniendo a prueba algoritmos con complejidad lineal se podrán alcanzar tamaños de los datos grandes (quiza hasta un millón o más), mientras que con algoritmos exponenciales a duras penas se podrá pasar de la veintena de elementos.

La ejecución del algoritmo para un tamaño determinado requiere dos fases:

1. Fase de *preparación*: si, por ejemplo, estamos poniendo a prueba un algoritmo de ordenación, habrá que rellenar un vector con los valores correspondientes. Lo normal es que esos valores sean aleatorios aunque dependiendo de la naturaleza del experimento quizá haya que rellenarlos con valores crecientes o decrecientes.

2. Ejecución del algoritmo midiendo el tiempo invertido. Para esa medición se necesita un *reloj de alta precisión*, como el que hay en `std::chrono` desde C++11. Tras la medición, escribimos el resultado.

```
#include <chrono>
using namespace std::chrono;

void rellenaVectorAleatorio(int v[], int n) {
   // ... TODO
```

```
}

// ...

// Para cada tamaño n:

// Preparación
rellenaVectorAleatorio(v, n);

// Medición
high_resolution_clock::time_point inicio = high_resolution_clock::now();
seleccion(v, n);
high_resolution_clock::time_point fin = high_resolution_clock::now();

// Escribimos en la salida estándar
auto duracion = duration_cast<milliseconds>( fin - inicio ).count();

cout << n << " " << duracion << '\n';
```

## 4 Generación de datos de un algoritmo

Para generar los datos completos de un algoritmo hay que repetir ese proceso para distintos tamaños de los datos, empezando por un tamaño pequeño e incrementandolo hasta llegar a un máximo. Conviene, además, para el experimento (dejar de probar con tamaños más grandes) si el tiempo invertido ha sido más de un valor. De esta forma, si al establecer el tamaño inicial y final somos demasiado optimistas y ponemos un valor final excesivo, la propia aplicación parará antes.

Algo así:

```
#include <chrono> // Requiere C++11
#include <iostream>
using namespace std;
using namespace std::chrono;

#include <stdlib.h> // rand

#define MAX_SEGUIDAS 3 // Si hay más de MAX_SEGUIDAS ejecuciones seguidas
#define MAX_TIME 2000  // que requieren más de MAX_TIME milisegundos,
                       // se termina el experimento
```

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

void seleccion(int v[], int n) {

   for (int i = 0; i < n-1; ++i) {

      int pmin = ii;
      for (int j = i+1; j < n; ++j) {
         if (v[j] < v[pmin])
            pmin = j;

         int temp = v[i];
         v[i] = v[pmin];
         v[pmin] = temp;
      }
  }
}

// Vector en variable global (mal) para poder tener
// arrays grandes sin desbordar la pila.
int v[1000000];

int main() {

   int tamInicial = 1000;
   int tamFinal = 100000;
   int incremento = 1000;
   int numSeguidas = 0;

   for (int n = tamInicial;
        (n <= tamFinal) && (numSeguidas < MAX_SEGUIDAS);
        n += incremento) {

      rellenaVectorAleatorio(v, n);

      // Medición
```

```
      high_resolution_clock::time_point inicio = high_resolution_clock::now();
      seleccion(v, n);
      high_resolution_clock::time_point fin = high_resolution_clock::now();

      // Escribimos en la salida estándar
      auto duracion = duration_cast<milliseconds>(fin - inicio).count();

      cout << n << " " << duracion << '\n';
   }

   return 0;
}
```

Para generar el fichero con los datos, basta compilar el código anterior y ejecutarlo desde la línea de órdenes redirigiendo la entrada al fichero deseado:

```
$ <ejecutable> > seleccion.txt
```

Conseguir los tiempos de ejecución para el *quicksort* de las STL de C++ es similar pero cambiando la llamada del algoritmo a `sort(v, v + n)` (habrá que incluir `algorithm.h`)[1].

## 5 Generación de las gráficas

Para generar las gráficas se puede utilizar Excel, copiando los datos de los ficheros en una hoja y generando las gráficas.

Aquí utilizaremos *gnuplot*. Si tenemos los dos ficheros, `seleccion.txt` y `sortSTL.cpp` con los tiempos de ejecución, podemos lanzar *gnuplot* y escribir[2] (las líneas que comienzan con # son comentarios; no hace falta escribirlas):

```
# Establecemos el título que aparecerá en la ventana
set title "Comparación de tiempos de dos algoritmos de ordenación"

# Dibujamos los datos de sortSTL.txt con una línea azul
```

[1] Dado que todos los programas que generan estos tiempos comparten mucho código, existen formas más recomendables de programarlo para evitar tanta duplicidad.

[2] *gnuplot* es lo suficientemente potente como para no exigir que los tamaños utilizados para ambos algoritmos coincidan, algo que sería incómodo en Excel pues habría que realizar a mano la "mezcla" de ambas colecciones de datos.

```
# de ancho 3 y leyenda 'sort C++' y los datos de seleccion.txt
# con una linea negra
plot 'sortSTL.txt' with lines title 'sort C++' lw 3 lt rgb "blue", \
     'seleccion.txt' with lines title 'seleccion' lw 3 lt rgb "black"
```

Se abrirá una ventana con la gráfica.

Se puede pedir a *gnuplot* que en lugar de abrir una ventana, guarde la imagen en un `.png`. Para eso basta establecer una salida distinta *antes* del `set title`:

```
# Establecemos el tipo de salida
set terminal pngcairo enhanced font "arial,14" \
    fontscale 1.0 size 1024, 576
# Y el nombre del fichero
set output 'grafica.png'

# Establecemos el título que aparecerá en la ventana
set title "Comparación de tiempos de dos algoritmos de ordenación"

# Dibujamos los datos de sortSTL.txt con una línea azul
# de ancho 3 y leyenda 'sort C++' y los datos de seleccion.txt
# con una linea negra
plot 'sortSTL.txt' with lines title 'sort C++' lw 3 lt rgb "blue", \
     'seleccion.txt' with lines title 'seleccion' lw 3 lt rgb "black"
```

Hacerlo así permite generar la gráfica desde la linea de órdenes directamente. Para eso basta con almacenar el código anterior en un fichero como `generaGrafica.gnuplot` y utilizarlo como entrada:

```
$ gnuplot < generaGrafica.gnuplot
```

## 6 gnuplot en los laboratorios

Si arrancas Linux en los laboratorios, encontrarás que *gnuplot* está instalado y puede ser utilizado desde una consola.

Si utilizas Windows entonces puedes descargartelo desde `http://www.gnuplot.info/` y descomprimirlo en `hlocal`. No requiere instalación, por lo que se puede ejecutar directamente.

## 7 Experimenta

- Genera la gráfica para comparar el algoritmo de selección y el `sort` de la STL.
- Genera una gráfica para comparar los tiempos de ejecución de dos algoritmos de ordenación de la STL: `sort` (visto antes) y `stable_sort`. ¿Crees que tienen los dos la misma complejidad? ¿Cuál es más rápido?
- Algunas veces nos interesa saber cuál es el elemento que ocupa el centro de una colección (o lo que es lo mismo, si ordenamos el vector, el valor que hay en el centro). En la librería de C++ hay una función que ayuda a calcularlo: `nth_element`, que recibe el principio y final de la colección y la posición que ocupa el elemento que queremos conocer. Al final en esa posición queda el elemento. Genera la gráfica para conocer el elemento utilizando `nth_element(v,v+n/2,v+n)` y compara el tiempo de ejecución con ordenar el vector entero con `sort`. ¿Crees que tienen los dos la misma complejidad? ¿Cuál es más rápido?