UNIVERSIDAD DE GRANADA

Práctica 1

Eficiencia Algorítmica

Estructura de Datos

Curso 2025/2026

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

29 de septiembre de 2025

Índice

1 Introducción

El análisis de eficiencia de algoritmos es una herramienta fundamental en programación. A través de esta práctica, el alumnado aprenderá a diferenciar entre eficiencia teórica (medida mediante notación asintótica) y eficiencia empírica (medida a través del tiempo de ejecución en diferentes casos).

Se trabajará exclusivamente en C++ desde terminal Linux, utilizando herramientas estándar del lenguaje y visualización de resultados mediante gráficos con `gnuplot`. El objetivo final es que el alumnado sea capaz de analizar, medir, comparar y reflexionar sobre el rendimiento de distintos algoritmos sencillos.

2 Objetivos

- Comprender el concepto de eficiencia algorítmica teórica y empírica.
- Medir el tiempo de ejecución de un programa en C++ mediante código.
- Automatizar la recolección de datos de eficiencia mediante scripts.
- Visualizar resultados empíricos usando `gnuplot`.
- Comparar resultados reales con la eficiencia teórica esperada.
- Documentar adecuadamente un análisis algorítmico.

3 Tutorial básico: suma de elementos

Antes de abordar los ejercicios, vamos a practicar el proceso completo con un ejemplo muy sencillo: sumar los elementos de un vector de enteros. Como se ha comentado en clase, los vectores son muy visuales para ver con claridad la eficiencia de los algoritmos.

Código en C++

El siguiente programa realiza una tarea muy sencilla: sumar los elementos de un vector de números enteros. Aunque el algoritmo es trivial, es perfecto para entender los pasos básicos que seguiremos en todos los ejercicios de esta práctica:

- Reservar memoria dinámica con `new`.
- Rellenar un vector con números aleatorios.
- Medir el tiempo de ejecución del algoritmo usando `clock()`.
- Imprimir por pantalla el tamaño del problema y el tiempo empleado.

Este código será la plantilla sobre la que construiremos el resto de algoritmos. Asegúrate de entender cada línea y prueba a modificar el tamaño del vector para comprobar que el tiempo crece de forma proporcional.

Observa también que el programa toma como argumento de entrada (`argv[1]`) el tamaño del vector. Esto es importante porque luego automatizaremos la ejecución del programa con distintos tamaños usando un script.

A continuación, el código completo en el Listing 1

```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

int suma(const int *v, int n) {
    int total = 0;
    for (int i = 0; i < n; i++)
        total += v[i];
    return total;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        cerr << "Uso: " << argv[0] << " TAM" << endl;
        return 1;
    }

    int tam = atoi(argv[1]);
    int *v = new int[tam];
    for (int i = 0; i < tam; i++)
        v[i] = rand() % 1000;

    clock_t start = clock();
    int resultado = suma(v, tam);
    clock_t end = clock();

    double tiempo = (end - start) / (double)CLOCKS_PER_SEC;

    cout << tam << "\t" << tiempo << endl;

    delete[] v;
    return 0;
}
```

Listing 1: suma_vector.cpp

Compilación y ejecución

Podemos compilar el programa manualmente usando el compilador g++, o bien utilizar un Makefile si queremos practicar con una herramienta profesional muy habitual en proyectos en C++. A continuación, mostramos una línea de compilación y una ejecución del programa.

La ejecución que se muestra en el Listing 2 es una sola ejecución del programa, pasando como argumento el valor 10000, que representa el tamaño del vector, es decir, el parámetro n sobre el que se mide la eficiencia temporal.

```
g++ suma_vector.cpp -o suma_vector
./suma_vector 10000
```

Listing 2: Compilación y ejecución

Si queremos automatizar el proceso de compilación, podemos crear nuestro archivo Makefile con el siguiente contenido:

```
# Makefile para compilar suma_vector

CXX = g++
CXXFLAGS = -Wall -O2

all: suma_vector
```

```
suma_vector: suma_vector.cpp
	$(CXX) $(CXXFLAGS) suma_vector.cpp -o suma_vector

clean:
	rm -f suma_vector
```

Listing 3: Makefile para suma_vector

Para compilar el programa usando este Makefile, basta con ejecutar en la terminal:

```
make
```

Y para eliminar el ejecutable generado:

```
make clean
```

Automatización de ejecuciones mediante un script

La eficiencia de un algoritmo no se debe medir con una sola ejecución puntual, ya que esta puede verse afectada por múltiples factores: carga del sistema, estado de la memoria, o incluso pequeñas variaciones internas del sistema operativo.

Para obtener resultados más representativos, debemos ejecutar el programa con distintos tamaños de entrada y registrar el tiempo que tarda en cada uno. De esta forma, obtendremos una colección de pares (n, t) donde n es el tamaño del problema y t el tiempo medido. Con estos datos podremos generar una gráfica que refleje cómo crece el tiempo de ejecución a medida que aumenta el tamaño del problema.

Para ello, vamos a utilizar un **script de consola** que automatiza el proceso. En este script:

- Se inicializa un fichero llamado `tiempos_suma.dat` donde se guardarán los resultados.
- Se ejecuta el programa `suma_vector` repetidamente con diferentes tamaños de entrada.
- Cada ejecución escribe una línea en el fichero con el tamaño y el tiempo correspondiente.

A continuación se muestra el contenido del script:

```
#!/bin/bash

echo -n > tiempos_suma.dat

for tam in $(seq 1000 1000 30000); do
  ./suma_vector $tam >> tiempos_suma.dat
done
```

Listing 4: ejecutar_suma.sh

Este script recorre valores desde 1000 hasta 30000, aumentando de 1000 en 1000. Por cada uno de estos tamaños, ejecuta el programa y añade la salida (tamaño y tiempo) al fichero de datos. Posteriormente, este fichero podrá usarse con herramientas como `gnuplot` para generar una representación visual del crecimiento del algoritmo.

Para ejecutar el script desde la terminal, asegúrate primero de darle permisos de ejecución:

```
chmod +x ejecutar_suma.sh
./ejecutar_suma.sh
```

Instalación de `gnuplot`

Para poder representar los gráficos de eficiencia, necesitaremos instalar la herramienta **`gnuplot`**, que es un programa de consola muy utilizado en entornos científicos.

Si estás usando una distribución de Linux basada en Ubuntu, puedes instalarlo desde la terminal con el siguiente comando:

```
sudo apt update
sudo apt install gnuplot
```

Una vez instalado, puedes abrir **`gnuplot`** simplemente escribiendo:

```
gnuplot
```

Se abrirá un entorno interactivo desde el que podrás ejecutar los comandos necesarios para representar tus datos y ajustar curvas.

Gráfico con gnuplot

Una vez que hemos ejecutado nuestro programa para distintos tamaños de entrada y hemos almacenado los resultados en el fichero `tiempos_suma.dat`, el siguiente paso es representar esos datos gráficamente para observar el comportamiento del algoritmo.

Para ello usaremos **gnuplot**, una herramienta de consola muy potente que permite representar funciones y datos de forma sencilla y rápida.

A continuación se muestran los comandos básicos que puedes introducir dentro de gnuplot para representar tus datos y ajustar una curva teórica:

```
gnuplot
plot "tiempos_suma.dat" with linespoints title "Suma"

f(x) = a*x + b
fit f(x) "tiempos_suma.dat" via a,b
plot "tiempos_suma.dat" with linespoints, f(x) title "Ajuste"
```

Listing 5: Comandos de gnuplot

Vamos a explicar qué hace cada uno de estos pasos:

- `gnuplot`: abre la herramienta interactiva de gráficos. Aparecerá un prompt desde donde podemos escribir instrucciones.
- `plot "tiempos_suma.dat"with linespoints title "Suma"`: esta orden dibuja un gráfico usando los datos de nuestro fichero. Cada línea contiene el tamaño del vector y el tiempo de ejecución. Se conectan los puntos con líneas para visualizar la tendencia.
- `f(x) = a*x + b`: definimos una función teórica del tipo lineal, que es la que esperamos que tenga este algoritmo. Los coeficientes `a` y `b` serán ajustados automáticamente.
- `fit f(x) "tiempos_suma.dat"via a,b`: gnuplot ajusta la función definida a los datos del fichero, calculando los mejores valores posibles para los coeficientes `a` y `b` mediante un proceso de regresión lineal.
- `plot "tiempos_suma.dat"with linespoints, f(x) title .ªjuste"`: finalmente, dibujamos tanto los datos experimentales como la función ajustada para poder compararlos visualmente.

Este proceso es fundamental para comprobar si el comportamiento empírico del algoritmo coincide con su eficiencia teórica. Si la línea ajustada se superpone (o se aproxima) a los datos experimentales, podremos concluir que nuestra hipótesis teórica era adecuada.

4 Ejercicios

Ejercicio 1: Algoritmo de búsqueda secuencial

En este ejercicio trabajaremos con un algoritmo de búsqueda secuencial. El objetivo es comprobar cuánto tiempo tarda un programa en recorrer un vector buscando un valor que no está presente. Esto nos permite estudiar el **peor caso posible**, ya que el algoritmo tiene que revisar todos los elementos antes de devolver que no lo ha encontrado.

Para este ejercicio se pide implementar un programa en C++ que:

- Reciba como argumento el tamaño del vector.
- Genere un vector de números aleatorios.
- Busque un número que se sabe con certeza que no está en el vector.
- Devuelva una salida en formato `tamaño_vector  tiempo_ejecución`, como en el ejemplo del tutorial.

Restricción: Recuerda que el número a buscar no puede aparecer en el vector. Una forma de hacerlo es asegurándonos que está fuera del rango de los números generados. Por ejemplo, si los valores aleatorios están entre 0 y 9999, puedes buscar el número 10000.

Ejercicio 2: Ordenación iterativa

En este ejercicio implementaremos un algoritmo de ordenación muy básico, pero útil para estudiar el comportamiento en distintos escenarios: el algoritmo consiste en recorrer el vector varias veces, intercambiando los elementos vecinos si están desordenados, y detenerse cuando ya no se realizan cambios. Este tipo de ordenación es conocido como **ordenación de la burbuja optimizada**, aunque en este ejercicio no mencionaremos ese nombre para que el alumnado lo deduzca por el comportamiento.

Escribe un programa en C++ que:

- Genere un vector de números enteros según el caso de prueba.
- Aplique el algoritmo de ordenación explicado.
- Mida el tiempo de ejecución usando `clock()`.
- Imprima por pantalla el tamaño del vector y el tiempo de ejecución.

Casos a medir:

Para estudiar cómo se comporta el algoritmo en distintas situaciones, deberás generar tres tipos de entrada:

a) **Vector aleatorio**: genera números aleatorios entre 0 y 10000.

b) **Vector ya ordenado**: genera una secuencia ascendente (`0, 1, 2, ..., n-1`).

c) **Vector en orden inverso**: genera una secuencia descendente (`n-1, n-2, ..., 0`).

Pasos a seguir:

1. Crea el programa `ordenacion.cpp` con el código que realiza la ordenación y la medición de tiempo.

2. Escribe un script para automatizar las ejecuciones en los tres casos anteriores. Guarda los datos en ficheros como `tiempos_ordenado.dat`, `tiempos_inverso.dat` y `tiempos_aleatorio.dat`.
3. Representa gráficamente cada uno con `gnuplot`.
4. Ajusta los datos empíricos a una función teórica y compara.

Ejercicio 3: Comparación de dos algoritmos

En este ejercicio vas a comparar el comportamiento de dos algoritmos de búsqueda que resuelven el mismo problema utilizando estrategias diferentes. Implementa ambos algoritmos y analiza experimentalmente cómo se comportan conforme aumenta el tamaño del vector.

- **Algoritmo A: búsqueda secuencial:** recorre el vector desde el principio hasta encontrar el número buscado.
- **Algoritmo B: búsqueda binaria:** parte el vector por la mitad y decide en qué mitad continuar buscando.

Ambos algoritmos deberán aplicarse sobre vectores ordenados, y el número a buscar debe estar presente en el vector (por ejemplo, el valor central), para que se puedan comparar en igualdad de condiciones.

Pasos a seguir:

1. Implementa ambos algoritmos de búsqueda.
2. Crea un vector ordenado de tamaño variable y busca un número que esté seguro dentro del vector.
3. Mide el tiempo de ejecución de cada algoritmo para varios tamaños.
4. Guarda los resultados en dos ficheros: `tiempos_seq.dat` y `tiempos_bin.dat`.
5. Representa ambas curvas en un mismo gráfico con `gnuplot`.
6. Ajusta cada conjunto de datos a una función matemática y compara su crecimiento.

Ejercicio 4: Multiplicación de matrices

En este ejercicio vamos a trabajar con un problema clásico en programación y estructuras de datos: la multiplicación de dos matrices cuadradas. Es un buen ejemplo para observar cómo crece el tiempo de ejecución cuando aumenta el tamaño del problema. Implementa un programa en C++ que:

- Genere dos matrices cuadradas de tamaño $N \times N$, con datos aleatorios enteros.
- Multiplique ambas matrices de forma clásica (sin optimizaciones ni librerías externas).
- Calcule y almacene el tiempo de ejecución del proceso.
- Imprima en pantalla el tamaño N y el tiempo medido (como en los ejercicios anteriores).

Pasos a seguir:

1. Crea el programa `multiplica_matrices.cpp`.

2. Automatiza la ejecución con un script que varíe N desde 50 hasta 500, en incrementos de 50.
3. Guarda los resultados en un fichero como `tiempos_matrices.dat`.
4. Representa los datos con `gnuplot` y ajusta una función matemática que describa su crecimiento.
5. Realiza también el análisis teórico del algoritmo: ¿cuántas operaciones realiza en función de N?

5 Entrega

La práctica debe organizarse de forma clara y profesional, utilizando una estructura de carpetas que facilite su corrección. La justificación del trabajo realizado será tan importante como el propio código: no basta con que el programa funcione; es necesario demostrar que has entendido lo que has hecho y que todas las respuestas estén lo suficientemente justificadas.

Estructura de carpetas

Deberás crear una carpeta principal que entregues comprimida en un .zip que contenga:

- Una subcarpeta por cada ejercicio:
  - `Ejercicio1`
  - `Ejercicio2`
  - `Ejercicio3`
  - `Ejercicio4`
- Un archivo PDF final llamado `documentacion.pdf`

Contenido de cada subcarpeta

Cada carpeta `EjercicioX` deberá incluir:

- El código fuente en C++ del algoritmo/los algoritmos implementado/s.
- El script de bash usado para automatizar las ejecuciones.
- El fichero de datos con los resultados obtenidos (`.dat`).
- (Opcional) Las gráficas generadas como imágenes (`.png` o `.pdf`).

documentacion.pdf

Este archivo será el informe final de la práctica. Es el documento principal y debe recoger de forma estructurada y razonada:

- Introducción breve de la práctica y metodología utilizada.
- Para cada ejercicio:
  - Descripción del problema abordado.
  - Captura o gráfico de resultados.

  - Análisis y reflexión sobre la eficiencia observada.
  - Ajuste de curva si se ha realizado.
  - Cálculo teórico de eficiencia (cuando se pida).
- Conclusión final sobre los aprendizajes y observaciones generales.

Nota sobre el uso de herramientas de IA

El uso de herramientas de inteligencia artificial como ChatGPT, Copilot o similares está permitido únicamente como apoyo puntual para resolver dudas o consultar documentación. Bajo ningún concepto se permitirá entregar código o informes generados íntegramente por IA sin comprensión ni justificación por parte del estudiante.

Esta práctica tiene como objetivo que desarrolles tu razonamiento algorítmico, tu capacidad de análisis y tu criterio propio. Por tanto, todo el contenido debe estar redactado en tus propias palabras y debes ser capaz de explicar y defender cualquier parte del código o reflexión presentada.

La profesora de la asignatura podrá realizar entrevistas breves o revisiones orales en caso de sospecha de uso inapropiado.