UNIVERSIDAD
DE GRANADA

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

ALGORÍTMICA

Práctica 1: Análisis de Eficiencia de Algoritmos

Shao Jie Hu Chen
Mario Megías Mateo
Jesús Samuel García Carballo

Grupo Rojo

1 de abril de 2022

Resumen

Esto es una prueba

Índice

1. Introducción

En esta primera práctica de la asignatura Algorítmica vamos a realizar un análisis teórico y empírico de diversos algoritmos vistos en la asignatura. En particular, dicho análisis se hará de los siguientes:

- **Algoritmos de ordenación:** Inserción, selección, quicksort y mergesort.
- **Algoritmo de análisis de caminos mínimos sobre grafos:** Algoritmo de Floyd.
- **Hanoi.** Resolución del juego de las torres de Hanoi.

1.1. Estructura de la memoria

Esta memoria presenta la estructura típica de un trabajo académico. En primer lugar, presentamos brevemente el contexto en el que nos manejamos en esta práctica. Posteriormente, analizaremos teóricamente cada uno de los algoritmos propuestos para, seguidamente, comprobar mediante un análisis empírico las soluciones obtenidas. Asimismo, se ha realizado un análisis híbrido contrastando los resultados teóricos con los experimentales. Este análisis se ha realizado para cada algoritmo, extrayéndose información relevante asociada a ellas.

Además, para cada uno de los algoritmos considerados, se ha elaborado conjuntamente entre los miembros del grupo tablas asociadas a cada una de ellas, que se irán presentando a lo largo de esta memoria. Finalmente, se escribirán unas líneas asociadas a las conclusiones obtenidas tras el desarrollo de esta práctica, así como los resultados más relevantes obtenidos.

1.2. Objetivos

Entre los objetivos que vamos a realizar asociadas a esta práctica, caben destacar los siguientes:

- **Estudio teórico, empírico e híbrido** y **comparación** de los algoritmos de ordenación más empleados, verificando los resultados teóricos.
- **Estudio teórico, empírico e híbrido** de algoritmos de alta complejidad, poniendo especial énfasis en su **viabilidad** en diferentes equipos.
- Estudio del **aumento de eficiencia** de un mismo algoritmo para diferentes **tipos de optimización** del compilador.
- Determinación del algoritmo **más adecuado** para cada situación en función del estado de los datos.

1.3. Equipo empleado

Para el desarrollo de esta práctica, se han empleado diferentes equipos, cuyas respectivas especificaciones técnicas se especifican a continuación:

- **ASUS**
  - **Modelo**: ZenBook 15 UX534F
  - **Procesador**: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz5
  - **Memoria Ram**: 16 GB DDR4 @ 2.133 MHz.
  - **Sistema Operativo** Ubuntu 20.04.2 LTS

- **LENOVO**
  - **Modelo**: YOGA 530-14IKB
  - **Procesador**: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
  - **Memoria RAM:** 8 GB DDR4
  - **Sistema Operativo**: Ubuntu 20.04.4 LTS
- **HP**
  - **Modelo**: HP Pavilion Gaming Laptop 15-dk0xxx
  - **Procesador**: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
  - **Memoria RAM:** 32 GB DDR4
  - **Sistema Operativo**: Ubuntu 20.04.4 LTS

Debido a las limitaciones técnicas de estos equipos, para el cálculo de la eficiencia empírica se ha optado por medir tiempos de ejecución en tiempos y **tamaños razonables** para obtener datos relevantes para el comportamiento tanto a pequeñas escalas como a nivel asintótico de ellas.

2. Eficiencia teórica

Para el análisis de los siguientes algoritmos se ha empleado la notación $O(.)$.

2.1. Algoritmo de inserción

El código de este algoritmo lo podemos encontrar a continuación.

```
1  static void insercion_lims(int T[], int inicial, int final)
2  {
3    int i, j;
4    int aux;
5    for (i = inicial + 1; i < final; i++) {
6      j = i;
7      while ((T[j] < T[j-1]) && (j > 0)) {
8        aux = T[j];
9        T[j] = T[j-1];
10       T[j-1] = aux;
11       j--;
12     };
13   };
14 }
```

Listing 1: Implementación en C++ del algoritmo de inserción.

El análisis de eficiencia se ha de hacer sobre el número de elementos del vector n.

Para el análisis asintótico, calculamos el número de operaciones elementales asociadas al código. Las tres primeras líneas tienen complejidad $O(1)$. Para el ciclo for, démonos cuenta de que itera sobre el número de componentes del vector. Por su parte, dentro del for existe un while que realiza un conjunto de operaciones de orden $O(1)$ tantas veces como se indica en el iterador. Por tanto, llamando $T(n)$ al números de operaciones asociadas al algoritmo, tenemos que:

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

Por tanto, deducimos que $T(n) \in O(n^2)$.

2.2. Algoritmo de selección

```
1  static void seleccion_lims(int T[], int inicial, int final)
2  {
3    int i, j, indice_menor;
4    int menor, aux;
5    for (i = inicial; i < final - 1; i++) {
6      indice_menor = i;
7      menor = T[i];
8      for (j = i; j < final; j++)
9        if (T[j] < menor) {
10   indice_menor = j;
11   menor = T[j];
12       }
13     aux = T[i];
14     T[i] = T[indice_menor];
15     T[indice_menor] = aux;
16   };
17 }
```

Listing 2: Implementación en C++ del algoritmo de selección.

La estructura principal del código anterior consta de dos bucles for anidados. Las primeras sentencias de declaración antes del inicio del primer bucle for no repercuten en el análisis. Procedamos a analizar la estructura repetitiva: el primer for recorre el vector desde su inicio hasta la penúltima componente del mismo incluída. El segundo for comienza en la posición dada por el índice del primero hasta la última componente del vector incluída. Dentro del bucle interno tenemos una operación elemental correspondiente a la comparación realizada en el if, que es $O(1)$. Dentro del bucle externo excluyendo el código del bucle interno nos encontramos con dos sentencias de asignación que no se tendrán en cuenta para el análisis. De igual modo, fuera del bucle externo al final del código nos encontramos con otras sentencias de asignación que también serán despreciadas. También tendremos en cuenta las operaciones elementales aritméticos correspondientes a la gestión de los bucles, que son $O(1)$. En conclusion, acotando las expresiones $O(1)$ por k tenemos que:

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i}^{n-1} k = \sum_{i=0}^{n-2} k(n-1-i+1) = \sum_{i=0}^{n-2} k(n-i) = kn \sum_{i=0}^{n-2} 1 - k \sum_{i=0}^{n-2} i$$
$$= kn(n-2+1) - k\frac{(n-2)(n-1)}{2} = kn^2 + kn - \frac{kn^2 - 3kn + 2k}{2}$$

Deducimos, por tanto, que $T(n) \in O(n^2)$.

2.3. Algoritmo HeapSort

```
1    for (i = num_elem/2; i >= 0; i--)
2      reajustar(T, num_elem, i);
```

```
  for (i = num_elem - 1; i >= 1; i--)
    {
      int aux = T[0];
      T[0] = T[i];
      T[i] = aux;
      reajustar(T, i, 0);
    }
}


static void reajustar(int T[], int num_elem, int k)
```

Listing 3: Implementación en C++ del algoritmo heapsort.

El heapsort es un algoritmo de ordenación por montículos que consta de dos bucles for independientes, que llaman a una función llamada reajustar. En la primera línea de código podemos encontrar una declaración de variable cuya eficiencia es $O(1)$ y luego los dos ciclos mencionados, que interan sobre el número de elementos del vector que denotaremos por n, luego realizaremos la eficiencia en torno a este dato. Ambos ciclos son independientes, así que vamos a analizar cada uno por separado y aplicando la regla del máximo nos quedaremos con el mayor.

La función reajustar trata de imitar el uso de un árbol APO mediante vectores y recorrerlo insertando elementos, luego como en un árbol APO la inserción se realiza con eficiencia $O(log_2(n))$. Así que llamando $T(n)$ al números de operaciones asociadas al algoritmo, tenemos lo siguiente:

- **Primer ciclo**

$$T(n) = \sum_{i=0}^{n/2+1} log_2(i) = log_2(k) \in O(n * log(n))$$

- **Segundo ciclo**

$$\sum_{i=2}^{n} \log_2(i) = \log_2(2) + ... + \log_2(n) = \log_2(n!) \in \mathcal{O}(n \log_2(n))$$

En conclusión vemos que como tenemos que aplicar la regla del máximo y ambos ciclos tienen de eficiencia $O(n * log(n))$ podemos afirmar que esa es la eficiencia teórica del algoritmo,

2.4. Algoritmo QuickSort

```
inline void quicksort(int T[], int num_elem)
{
  quicksort_lims(T, 0, num_elem);
}

static void quicksort_lims(int T[], int inicial, int final)
{
  int k;
  if (final - inicial < UMBRAL_QS) {
    insercion_lims(T, inicial, final);
  } else {
    dividir_qs(T, inicial, final, k);
    quicksort_lims(T, inicial, k);
    quicksort_lims(T, k + 1, final);
  };
```

```
16 }
17
18
19 static void dividir_qs(int T[], int inicial, int final, int & pp)
20 {
21   int pivote, aux;
22   int k, l;
23
24   pivote = T[inicial];
25   k = inicial;
26   l = final;
27   do {
28     k++;
29   } while ((T[k] <= pivote) && (k < final-1));
30   do {
31     l--;
32   } while (T[l] > pivote);
33   while (k < l) {
34     aux = T[k];
35     T[k] = T[l];
36     T[l] = aux;
37     do k++; while (T[k] <= pivote);
38     do l--; while (T[l] > pivote);
39   };
40   aux = T[inicial];
41   T[inicial] = T[l];
42   T[l] = aux;
43   pp = l;
44 };
```

Listing 4: Implementación en C++ del algoritmo QuickSort.

En este apartado, consideramos n como el número de componentes del vector.

El código del algoritmo comienza en la función quicksort_lims en la línea 6. Tomaremos inicial como ñaa posición de comienzo del vector ($inicial = 0$), y final como el siguiente al último elemento, que coincide con el número de componentes del vector ($final = n$). Tenemos una estructura condicional que determina el comportamiento del algoritmo según si el tamaño del vector supera o no un cierto umbral, por lo que el tiempo de ejecución será una función definida a trozos. Si es menor que el umbral, ejecutamos la ordenación con el algoritmo de inserción previamente analizado, que es $O(n^2)$. De lo contrario, realizamos la ordenación según el propio algoritmo quicksort. Dicho algoritmo se basa en la técnica de Divide y Vencerás, implementándose de forma recursiva. En primer lugar, se realiza una llamada a la función dividir dividir_qs, cuya eficiencia procedemos a analizar. En las líneas 24-26 y 40-43 tenemos sentencias de asignación que serán despreciadas. Siguiendo código de forma secuencial, en las líneas 27 y 30 tenemos dos bucles do-while, si consideramos el peor caso para el primero (el pivote es mayor que el resto de elemento) tiene eficiencia $O(n)$, y el caso contrario al anterior sería el más desfavorable para el segundo (el pivote es menor que el resto de los elementos), luego tendríamos $O(n)$. Como el caso más favorable para uno (eficiencia $O(1)$) es el más desfavorable para el otro, aplicando la regla del Máximo tenemos que la eficiencia de la parte del código de los do-while sería $O(n)$. A continuación, encontramos un bucle while que se ejecuta un número indeterminado de veces en función de la condición, luego podemos acotar estas ejecuciones por una constante c. En las líneas 34-36 tenemos sentencias de asignación que serán ignoradas, y a continuación dos bucles do-while con la misma funcionalidad que los analizados anteriormente. Por tanto, deducimos que la eficiencia del bucle while es $O(n)$, y aplicando la regla del Máximo con el bloque de código de los bucles do-while obtenemos que la eficiencia de la función es

$O(n)$. A continuación hacemos dos llamadas recursivas, una que procesará los datos desde el comienzo del mismo hasta la posición en la que hemos ubicado el pivote con la función dividir_qs, y la otra se encarga de procesar el resto del vector. Por tanto, el tiempo de ejecución sería:

$$T(n) = \begin{cases} T(k_n) + T(n - k_n) + n & \text{si } n \geq \text{UMBRAL} \\ n^2 & \text{si } n < \text{UMBRAL} \end{cases} \tag{1}$$

Supongamos que $k_n = \frac{n}{2}$. Resolvamos la recurrencia de (1):

$$T(n) = T(k_n) + T(n - k_n) + n \text{si n} \geq \text{UMBRAL} \tag{2}$$

Realizamos el cambio de variable $n = 2^n$ en la ecuación (2) obtenemos:

$$T(2^m) = 2T(2^{m-1}) + 2^m sin \leqslant log_2(\text{UMBRAL}) \tag{3}$$

$$T(2^m) - 2T(2^{m-1}) = 2^m \tag{4}$$

Renombramos la expresión anterior como $T(2^m) = t_m$ obtenemos:

$$t_m - 2t_{m-1} = 2^m \tag{5}$$

Obtenemos la siguiente ecuación de recurrencia de la ecuación (4):

$$(x - 2)^2 = 0 \tag{6}$$

Luego la solución general de la recurrencia es:

$$t_m = c_1 2^m + c_2 m 2^m \tag{7}$$

Deshacemos el cambio de variabe:

$$T(n) = c_1 n + c_2 n log_2(n) \tag{8}$$

Por tanto $T(n) \in O(nlog_2(n))$

2.5. Algoritmo de Floyd

A continuación se presenta el código de este algoritmo:

```
1 void Floyd(int **M, int dim)
2 {
3   for (int k = 0; k < dim; k++)
4     for (int i = 0; i < dim;i++)
5       for (int j = 0; j < dim;j++)
6         {
7     int sum = M[i][k] + M[k][j];
8         M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
```

```
9           }
10  }
```

Listing 5: Implementación en C++ del algoritmo de Floyd.

En este caso, consideramos como n el número de filas/columnas de la matriz. Como los tres bucles iteran cada uno sobre n, siendo el número de iteraciones independientes entre sí y se realizan k operaciones en cada una de ellas, tenemos que llamando $T(n)$ al número de operaciones para un tamaño n, se verifica:

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} k = kn^3$$

Deducimos fácilmente que $T(n) \in O(n^3)$.

2.6. Algoritmo Hanoi

```
1  void hanoi (int M, int i, int j)
2  {
3    if (M > 0)
4      {
5        hanoi(M-1, i, 6-i-j);
6        // cout << i << " -> " << j << endl;
7        hanoi (M-1, 6-i-j, j);
8    }
9  }
```

Listing 6: Implementación en C++ del algoritmo de resolución de las torres de Hanoi.

Estamos ante el estudio de un algoritmo recursivo, así que si T(n) es el número de movimientos necesarios para mover n discos, tenemos que el algoritmo sigue la recurrencia $T(n) = 2T(n-1) + 1 \forall n \geq 1 y con T(0) = 0 como caso base$. Por lo tanto, vamos a resolverla para hallar su eficiencia teórica: Tenemos $t_n - 2t_(n-1) = 1$ luego si hacemos un cambio de varible nos queda como ecuación característica $(x-2)(x-1) = 0$ que son las propias raíces de la recurrencia luego la solución nos queda $t_n = c_1 1 * *n + c_2 2 * *n$, entonces como $t_0 = 0$ nos sale que $t_0 = c_1 + c_2$ y $t_1 = 2t_0 + 1 = 1$ luego en la solución $t_1 = c_1 + 2c_2$, luego si resolvemos el sistema $c_1 = 1$ y $c_2 = -1$nos queda la ecuación $2 * *n - 1 * *n$, por lo que como se trata de la ecuación que modela la recurrencia, podemos concluir con que la eficiencia teórica del algoritmo es $O(2 * *n)$.

3. Eficiencia empírica

Para esta práctica hemos ejecutado en cada uno de los equipos los diversos algoritmos presentados previamente. Las tablas obtenidas, así como las gráficas que representan estas tablas se representan a continuación.

3.1. Algoritmo de inserción

En la siguiente tabla se encuentran los datos obtenidos en este algoritmo por cada uno de los equipos en los que hemos ejecutado el algoritmo.

$N_{componentes}$	t_{ASUS}	t_{HP}	t_{LENOVO}
50	0.14	0.17	0.33
4048	453.56	515.99	651.62
8046	1754.95	1812.13	2302.02
12044	3939.03	3964.86	5151.86
16042	7001.02	7121.99	9086.88
20040	10881.51	11267.74	14199.60
24038	15631.83	16306.35	20487.40
28036	21153.50	22116.26	27616.21
32034	27637.29	28858.12	36049.05
36032	34987.36	36138.26	45555.31
40030	43165.74	44968.61	55917.47
44028	52206.10	54424.15	67842.75
48026	62037.85	64616.22	80669.90
52024	72855.27	75586.21	94386.29
56022	84416.77	86679.96	109490.70
60020	97044.37	98427.26	125816.55
64018	110296.65	112276.70	143332.05
68016	124351.80	125903.00	161613.60
72014	139565.40	140651.00	180965.90
76012	155826.80	154545.50	201664.70
80010	172265.90	171206.85	223414.75
84008	189906.35	187602.35	246361.85
88006	208708.60	206964.30	269969.40
92004	228250.85	229561.45	295370.30
96002	248827.55	245400.70	321198.25

Cuadro 1: Tiempos de ejecución (en μs) del algoritmo de inserción

Tal y como podemos observar, tenemos que nuevamente el equipo de ASUS ofrece un **mejor rendimiento** respecto a los otros dos equipos en los que se ha realizado el análisis. Asimismo, el tiempo de ejecución aumenta considerablemente en cuanto el número de componentes de vector en comparación con los algoritmos de QuickSort o MergeSort.

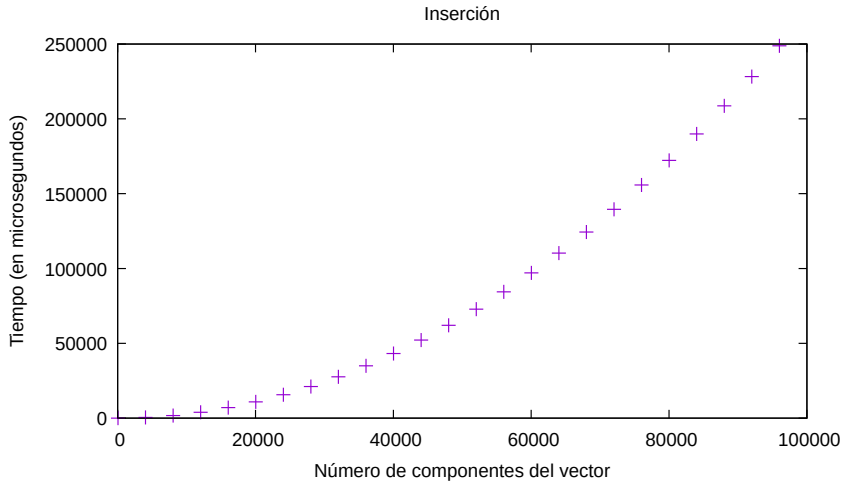
Figura 1: Ilustración del funcionamiento interno del algoritmo de inserción

Las gráficas asociadas a cada equipo se pueden encontrar a continuación:

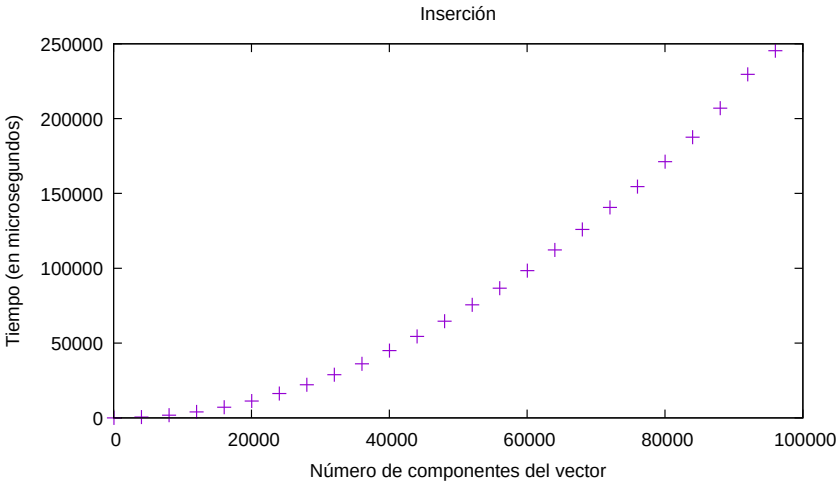
3.2. Algoritmo de selección

3.3. Algoritmo HeapSort

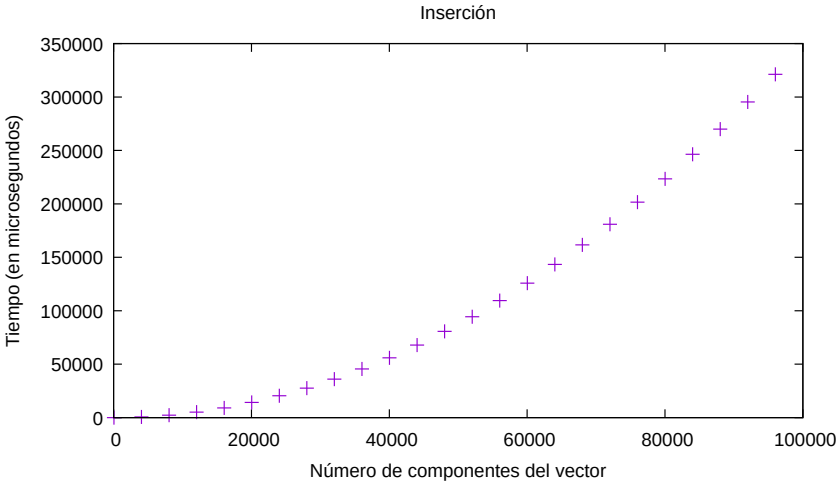
3.4. Algoritmo QuickSort



(a) ASUS



(b) HP



(c) LENOVO

Figura 2: Representación gráfica de tiempos de ejecución del algoritmo de inserción.

3.5. Algoritmo de Floyd

En la siguiente tabla se encuentran los datos obtenidos en este algoritmo por cada uno de los equipos en los que hemos ejecutado el algoritmo.

N_{nod}	t_{ASUS}	t_{HP}	t_{LENOVO}
5	0.09	0.10	0.13
35	4.63	5.04	3.79
65	28.11	36.27	22.68
95	98.97	59.42	76.20
125	206.65	291.02	171.84
155	250.74	340.20	294.25
185	368.58	490.17	466.89
215	572.04	588.29	727.49
245	846.54	873.17	1069.24
275	1193.13	1234.56	1530.57
305	1642.20	1666.67	2058.39
335	2164.11	2205.70	2755.76
365	2826.06	2846.01	3574.31
395	3528.34	3583.40	4484.65
425	4410.26	4465.64	5606.11
455	5383.56	5485.51	6820.05
485	6554.77	6483.90	8154.31
515	7855.67	7928.90	9760.62
545	9386.40	9268.55	11542.12
575	10957.08	10777.34	13740.25
605	12786.02	12425.00	15864.58
635	14618.06	14709.94	18181.38
665	16823.92	16768.84	21031.77
695	19147.07	18815.95	23953.15
725	21807.53	21948.21	27128.60

Cuadro 2: Tiempos de ejecución (en μs) del algoritmo de Floyd

En concordancia con los resultados obtenidos en las anteriores tablas, el ordenador que ofrece peores tiempos es el LENOVO, siendo el de ASUS el que ofrece resultados con mayor rapidez.

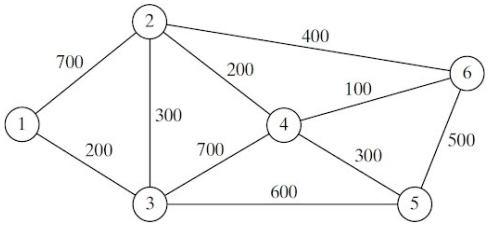
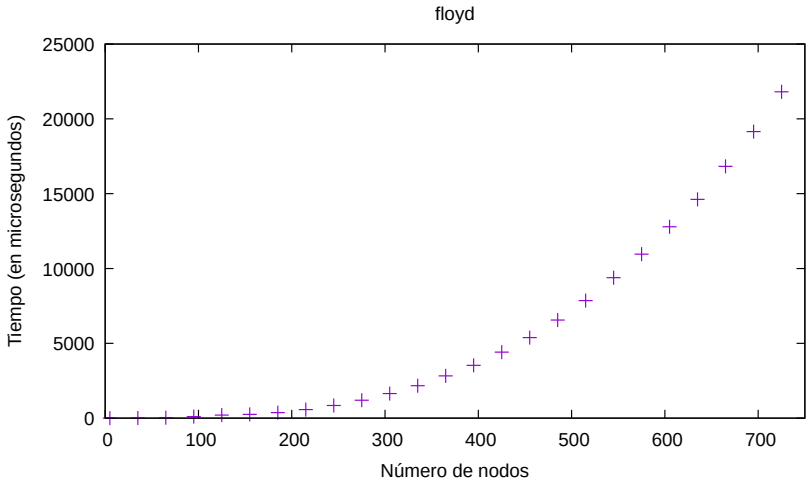
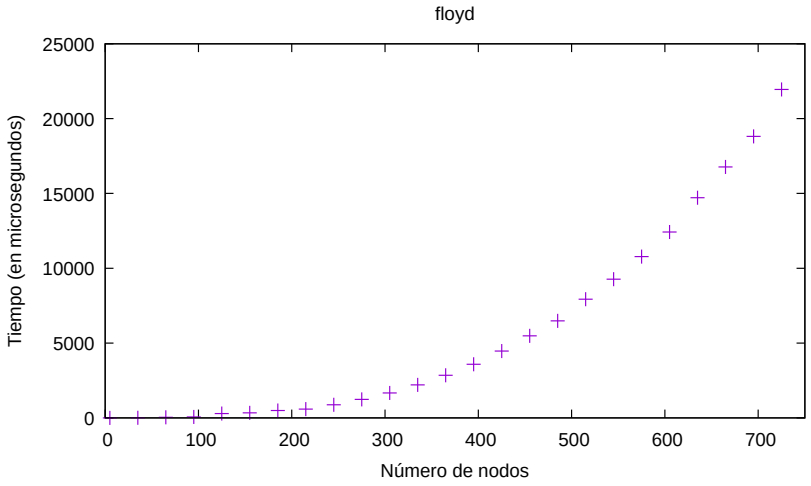


Figura 3: Representacion gráfica de un grafo.

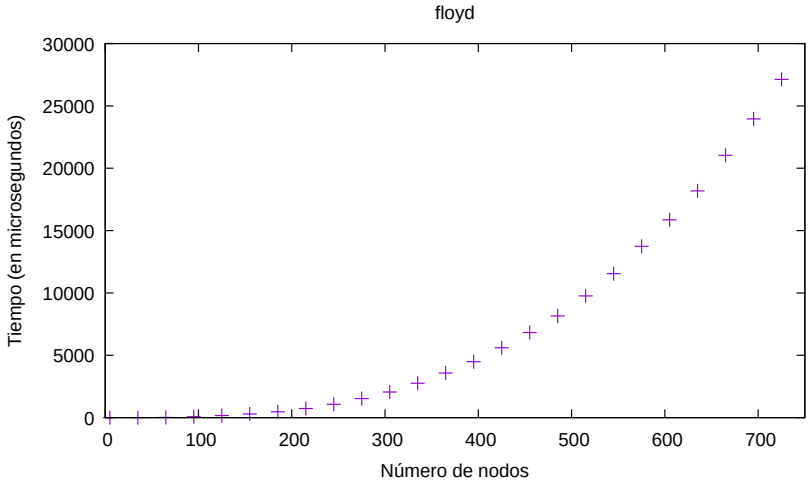
A continuación, usaremos el algoritmo de Floyd para ilustrar uno de los objetivos mencionados de la práctica, el cual consiste en analizar la eficiencia de un mismo algoritmo para optimizaciones del compilador diferentes. En concreto, al igual que en el resto de ejecuciones, usamos



(a) ASUS



(b) HP



(c) LENOVO

Figura 4: Representación gráfica de tiempos de ejecución del algoritmo Floyd.

g++ como compilador, y analizaremos las siguientes opciones de optimización:

- **-O0 (sin optimización)**: Desconecta por completo la optimización y es el predeterminado si no se especifica ningún nivel.
- **-O1**: El nivel de optimización más básico. El compilador intentará producir un código rápido y pequeño sin tomar mucho tiempo de compilación.
- **-O2**: Un paso delante de -O1. Es el nivel recomendado de optimización, a no ser que el sistema tenga necesidades especiales. -O2 activará algunas opciones añadidas a las que se activan con -O1. Con -O2, el compilador intentará aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación.
- **-O3**: El nivel más alto de optimización posible. Activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. El hecho de compilar con -O3 no garantiza una forma de mejorar el rendimiento y, de hecho, en muchos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria.