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.

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 empírico** y **comparación** de los algoritmos de ordenación más empleados, verificando los resultados teóricos.
- **Estudio empírico** 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.

```
static void insercion_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial + 1; i < final; i++) {
    j = i;
    while ((T[j] < T[j-1]) && (j > 0)) {
      aux = T[j];
      T[j] = T[j-1];
      T[j-1] = aux;
      j--;
    };
  };
}
```

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:

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

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

2.3. Algoritmo heapsort

```
1  static void heapsort(int T[], int num_elem)
2  {
```

```
  int i;
  for (i = num_elem/2; i >= 0; i--)
    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);
    }
}
```

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.

```
{
  int j;
  int v;
  v = T[k];
  bool esAPO = false;
  while ((k < num_elem/2) && !esAPO)
    {
      j = k + k + 1;
      if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
  j++;
      if (v >= T[j])
  esAPO = true;
      T[k] = T[j];
      k = j;
    }
  T[k] = v;
}
```

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

$$T(n) = \sum_{i=0}^{log_2(n)} k = k(n/2) => log_2(n).$$

En la primera línea de código de la función heapsort 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, luego vamos a analizar cada uno por separado.

El primero itera $n/2 + 1$ veces la función reajustar, que está formada por cuatro operaciones elementales al inicio del código, de eficiencia $O(1)$, al tratarse de declaraciones y asignaciones, y un bucle while cuyo contenido va iterando en torno a potencias de dos, pues k se va decrementando a la mitad en cada iteración, así que si $T(n)$ es el números de operaciones asociadas al algoritmo, tenemos que:

$$T(n) = \sum_{i=0}^{log_2(n)} k = k(n/2) => log_2(n).$$

la función reajustar, cuya eficiencia teórica acabamos de estudiar que es $k * n/2$. El primer ciclo va iterando $n/2+1$ veces y en su cuerpo posee la función reajustar, que llama en cada iteración, así que:

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

Para el siguiente ciclo podemos darnos cuenta de que

2.4. Algoritmo QuickSort

```
1  inline void quicksort(int T[], int num_elem)
2  {
3    quicksort_lims(T, 0, num_elem);
4  }
5
6  static void quicksort_lims(int T[], int inicial, int final)
7  {
8    int k;
9    if (final - inicial < UMBRAL_QS) {
10     insercion_lims(T, inicial, final);
11   } else {
12     dividir_qs(T, inicial, final, k);
13     quicksort_lims(T, inicial, k);
14     quicksort_lims(T, k + 1, final);
15   };
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 5: Implementación en C++ del algoritmo QuickSort.

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

Para el análisis de este algoritmo, vamos a calcular previamente la complejidad de las funciones auxiliares. Para la función dividir_qs, tenemos que su eficiencia es $O(n)$. En efecto, para cada bucle de la función tenemos que se procesa 1 (o una cantidad finita de veces) un elemento del vector.

Ahora bien, para la eficiencia del algoritmo, sea $T(n)$ el número de operaciones elementales que realiza el algoritmo. Tenemos que:

$$T(n) = \begin{cases} T(k_n) + T(n - k_n) + n & \text{si } n > \text{UMBRAL} \\ n^2 & \text{si } n \leq \text{UMBRAL} \end{cases} \quad (1)$$

Resolvamos la recurrencia de (1):

$$T(n) = \{\ T(k_n) + T(n - k_n) + n (2)$$

2.5. Algoritmo 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 6: 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 $T(n) \in O(n^3)$.

3. Eficiencia empírica