UNIVERSIDAD
DE GRANADA

Universidad de Granada

Facultad de Ingeniería Informática y telecomunicaciones

Práctica 1: Eficiencia de algoritmos

Doble Grado Ingeniería Informática y Matemáticas

Autores:
Adolfo Martínez Olmedo, Pablo Delgado Galera,
Marcos Baena Solar

Marzo 2025

Índice

1. Introducción

Comenzemos estableciendo las características de cada uno de nuetros ordenadores, ya que tienen prestaciones diferentes

	CPU	RAM	Caché L1d	Caché L1i	Caché L2	Caché L3	SO
Adolfo Martínez	AMD Ryzen 7 4800HS	16GB	256 KiB	256 KiB	4 MiB	8 MiB	Ubuntu 22.04.4
Marcos Baena	AMD 3020e with Radeon Graphics	5,7Gi	64 KiB (2 instances)	128 KiB (2 instances)	1 MiB (2 instances)	4 MiB (1 instance)	Ubuntu 24.04.1 LTS
Pablo Delgado	11th Gen Intel(R) Core(TM) i7-11800H	15 Gi	384 KiB (8 instances)	256 KiB (8 instances)	10 MiB (8 instances)	24 MiB (1 instance)	Ubuntu 24.04.1

Cuadro 1: Características de los ordenadores

En esta práctica vamos a discutir la eficiencia de los algoritmos desde tres puntos de vista distintos:

- **Punto 1:** Descripción del primer punto.
- **Punto 2:** Descripción del segundo punto.
- **Punto 3:** Descripción del tercer punto.

1.1. Análisis de la eficiencia teórica

En el análisis de la eficiencia teórica estudiaremos el tiempo de ejecución del algoritmo mediante funciones en notación *Big-O*, que representarán el peor caso posible. En este análisis, no usaremos medidas reales de computación, sino que calcularemos funciones mediante técnicas vistas en Estructuras de Datos y Algorítmica.

1.2. Análisis de la eficiencia empírica

Para el análisis de la eficiencia empírica ejecutaremos los algoritmos implementados en C++ en cada una de nuestras máquinas y mediermos el tiempo de ejecución mediante la clase <chrono>. Cada miembro del equipo ejecutará cada algoritmo 10 veces con todos los tamaños especificados, para luego hacer una media y obtener resultados más fiables.

1.3. Análisis de la eficiencia híbrida

En el análisis de la eficiencia hibrída, tomammos los resultados de los integrantes del grupo y hallamos la constante κ. En la representación de los resultados usaremos la herramienta gnuplot.

Para poder completar esta parte del estudio de la eficiencia usaremos los resultados del análisis teórico, para poder conocer la forma de la función a

la que queremos ajustar los datos. Por ejemplo para representar en gnuplot $O(n^3)$:

```
gnuplot> f(x) = a0*x*x+a1*x*x+a2
```

Listing 1: Ejemplo de $O(n^2)$

Después de esto debemos hacer la regresión usando el método de mínimos, cuyo funcionamiento conoces gracias a la asignatura EDIP:

```
gnuplot> fit f(x) 'result.dat' via a0,a1,a2
```

Listing 2: Uso de gnuplot para la regresión

En este caso result.dat es nuestro fichero de datos. Nos centraremos en *Final set of Parameters*, que nos muestra los coeficientes de la fórmula de regresión junto con la bondad del ajuste realizado.

Finalmente, hacemos el plot de los puntos y la curva de ajuste para ver como de buena es el cálculo de la eficiencia híbrida. Usaremos el siguiente comando:

```
gnuplot> plot 'result.dat', f(x) title 'Curva de
   ajuste'
```

Listing 3: Representación de la regresión

2. Desarrollo

Una vez que hemos discutido las maneras de estudiar la eficiencia, veamos los problemas que vamos a analizar: La **Ordenación de vectores**, los **Números de Catalan**, y las **Torres de Hanoi.**

2.1. Ordenación de vectores

2.1.1. QuickSort

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
int partition(vector<int> &vec, int low, int high) {
        int pivot = vec[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
                if (vec[j] <= pivot) {
```

```
7                           i++;
8                           swap(vec[i], vec[j]);
9                   }
10          }
11          swap(vec[i + 1], vec[high]);
12          return i + 1;
13  }
14
15  void QuickSort(vector<int> &vec, int low, int high) {
16          if (low < high) {
17                  int pi = partition(vec, low, high);
18                  QuickSort(vec, low, pi - 1);
19                  QuickSort(vec, pi + 1, high);
20          }
21  }
```

Listing 4: Código de QuickSort

Tras analizar el código hemos llegado a la siguiente fórmula de recurrencia:

$$T(n) = 2 \cdot T(n/2) + n$$

Concluimos que en las dos llamadas recursivas de QuickSort se utiliza *pi* que es el resultado de hacer la partición por la mitad del vector. En cuanto al término independiente este se corresponde al bucle de la función auxiliar *partition*, que en el peor de los casos recorre el vector totalmente.

Procedemos a resolver la fórmula de recurrencia utilizando el cambio de variable $n = 2^h$:

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

El resultado de la parte homogénea es el siguiente:

$$\begin{aligned} x - 2 &= 0 \\ x &= 2 \end{aligned}$$

El resultado de la parte independiente es:

$$\begin{aligned} p(h) &= 1 \\ b &= 2 \\ d &= 0 \end{aligned}$$

Por tanto la solución es $(x-2)^2$, con multiplicidad 2,y deshaciendo el cambio de variable y expresando la ecuación con coeficientes llegamos finalmente a:

$$c_1 \cdot 2^{\log_2(n)} + c_2 \cdot \log_2(n) \cdot 2^{\log_2(n)}$$

De donde se deduce que QuickSort tiene una eficiencia de $O(n \cdot \log(n))$

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 100 hasta 1.000.000 con saltos de 31.000:

2.2. Los números de Catalan

2.2.1. Versión recursiva

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 100 hasta 1.000.000 con saltos de 31.000:

2.2.2. Versión iterativa (programación dinámica)

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 100 hasta 1.000.000 con saltos de 31.000:

2.2.3. Versión iterativa directa usado el coeficiente binomial

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 100 hasta 1.000.000 con saltos de 31.000:

2.3. Las Torres de Hanoi

2.3.1. Versión recursiva

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1  void Hanoi(int n, char from_rod, char to_rod,
2             char aux_rod)
3  {
4      if (n == 0) {
5          return;
6      }
7      Hanoi(n - 1, from_rod, aux_rod, to_rod);
8      cout << "Move disk " << n << " from rod " << from_rod
9           << " to rod " << to_rod << endl;
10     Hanoi(n - 1, aux_rod, to_rod, from_rod);
11 }
```

Listing 5: Código de Hanoi

Tras analizar el código hemos llegado a la siguiente fórmula de recurrencia:

$$H(n) = \begin{cases} 0, & \text{si } n = 0, \\ 2 \cdot H(n-1), & \text{si } n > 0 \end{cases}$$

Concluimos que en las dos llamadas recursivas de *Hanoi* se utiliza *n - 1*, de donde sale $2 \cdot H(n-1)$, y el caso base 0 ocurre cuando hay 0 discos.

Procedemos a resolver la ecuación:

$$\begin{aligned} x - 2 &= 0 \\ x &= 2 \end{aligned}$$

Por tanto la solución es $(x - 2)$, con multiplicidad 1, y expresando la ecuación con coeficientes llegamos finalmente a:

$$c_1 \cdot 2^n$$

De donde se deduce que Hanoi tiene una eficiencia de $O(2^n)$

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 100 hasta 1.000.000 con saltos de 31.000:

2.3.2. Versión iterativa usando una pila

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1  void moveDisk(int a, int b)
2  {
3          if (!stacks[a].empty() && (stacks[b].
               empty() || stacks[a].top() < stacks[b
               ].top()))
4          {
5                  stacks[b].push(stacks[a].top());
6                  stacks[a].pop();
7          }
8  }
9
10 void Hanoi(int n)
11 {
12
13         int src = 0, aux = 1, dest = 2;
14         for (int i = n; i > 0; i--)
15                 stacks[src].push(i);
16
17         int totalMoves = (1 << n) - 1;
18         if (n % 2 == 0)
19                 swap(aux, dest);
20
21         for (int i = 1; i <= totalMoves; i++)
22         {
23                 if (i % 3 == 0)
24                         moveDisk(aux, dest);
25                 else if (i % 3 == 1)
26                         moveDisk(src, dest);
27                 else
28                         moveDisk(src, aux);
29         }
30 }
```

Listing 6: Código de Hanoi Iterativo con pila

Tras analizar el código vemos que:

El primer bucle claramente tiene una eficiencia de $O(\mathrm{n})$ ya que vemos que comienza con $i = n$ (el número de discos) y va decrementando hasta llegar a 0.

El segundo bucle recorre desde $i = 1$ hasta llegar a la variable totalMoves

que se obtiene con la función $(1 \ll n) - 1$ que equivale a $2^n - 1$ por tanto el segundo bucle tiene una eficiencia de $O(2^n)$.

Por tanto, por la regla del máximo, al no estar los bucles anidados obtenemos que la eficiencia total de la función es $O(2^n)$.

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 100 hasta 1.000.000 con saltos de 31.000:

2.3.3. Versión iterativa sin usar la pila

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1   void moveDisksBetweenTwoPoles(struct Stack *src,
        struct Stack *dest, char s, char d) {
2       int pole1TopDisk = pop(src);
3       int pole2TopDisk = pop(dest);
4
5       if (pole1TopDisk == INT_MIN) {
6           push(src, pole2TopDisk);
7       } else if (pole2TopDisk == INT_MIN) {
8           push(dest, pole1TopDisk);
9       } else if (pole1TopDisk > pole2TopDisk) {
10          push(src, pole1TopDisk);
11          push(src, pole2TopDisk);
12      } else {
13          push(dest, pole2TopDisk);
14          push(dest, pole1TopDisk);
15      }
16  }
17
18  void tohIterative(int num_of_disks, struct Stack
        *src, struct Stack *aux, struct Stack *dest) {
19      char s = 'S', d = 'D', a = 'A';
20      if (num_of_disks % 2 == 0) swap(d, a);
21
22      int total_num_of_moves = pow(2,
            num_of_disks) - 1;
23      for (int i = num_of_disks; i >= 1; i--)
            push(src, i);
```

```
24
25                  for (int i = 1; i <= total_num_of_moves;
                        i++) {
26                        if (i % 3 == 1)
                             moveDisksBetweenTwoPoles(src,
                             dest, s, d);
27                        else if (i % 3 == 2)
                             moveDisksBetweenTwoPoles(src,
                             aux, s, a);
28                        else moveDisksBetweenTwoPoles(aux
                             , dest, a, d);
29                  }
30            }
```

Listing 7: Código de Hanoi Iterativo sin pila

Tras analizar el código vemos que:
El primer bucle claramente tiene una eficiencia de $O(\mathrm{n})$ ya que vemos que comienza con $i = n$ (el número de discos) y va decrementando hasta llegar a 1.

El segundo bucle recorre desde $i = 1$ hasta llegar a la variable total_num_of_moves que se obtiene con la función pow(2, num_of_disks) - 1 que equivale a $2^n - 1$ por tanto el segundo bucle tiene una eficiencia de $O(2^n)$.

Por tanto, por la regla del máximo, al no estar los bucles anidados obtenemos que la eficiencia total de la función es $O(2^n)$.

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 100 hasta 1.000.000 con saltos de 31.000:

3. Conclusión

En conclusión, somos sudorosos.