AED II - PRÁCTICA 1

Tema 2 - Divide Y Vencerás

2022/2023

Subgrupo 1.2
Profesor: Francisco José Montoya

Sergio Ros Liarte
Jose Mª López Serrano

ÍNDICE

1. Introducción

En esta memoria se explica el algoritmo y su análisis de eficiencia planteado para resolver el problema asignado en la práctica 1 de la asignatura AED II.En la resolución de este problema se debe usar la técnica de Divide y vencerás. Nuestro problema a resolver era el número 2.

2) Dadas dos cadenas A y B de la misma longitud n, y un natural m n, hay que encontrar el índice p de inicio de la subcadenas de tamaño m con más diferencia total en valor absoluto entre los caracteres en cada posición de A y B (suma de los m valores A[i]-B[i] entre las posiciones p y p+m-1). Devolver el índice p de comienzo de la solución y el valor de la mayor diferencia total. En caso de empate, será válida cualquiera de las soluciones óptimas.
Ejemplo:
n=10,
m=5
A= c d d a b c d a c c
B= c a c d d b c a d c
dife: 0 3 1 3 2 1 1 0 1 0
Solución: posición de inicio igual a 2, y diferencia total igual a 10.

ALFABETO para todos los problemas: abcdefghijklmnopqrstuvwxyz.
Para todos los ejercicios, 10^4<= n<=10^6 . Donde aparezca m, m = n/1000

2. Pseudocódigo y explicación del algoritmo

*Este algoritmo en pseudocódigo está planteado para trabajar el problema sobre un array de enteros con las diferencias de cada carácter en valor absoluto precalculado. Por ejemplo, si se quiere resolver el problema para A={“a”,”z”} y B={“z”,”a”} este algoritmo trabajará sobre D[2] = {25,25}. Tomamos esta decisión porque simplifica los cálculos en el propio algoritmo y al fin y al cabo siempre se tiene que calcular la diferencia de las cadenas.

```
Tipo Solucion
        posicion: entero
        diferencia: entero

operacion Pequeno (inicio, final: entero): booleano
        SI (final-inicio+1) <= 2*m ENTONCES
                devolver (VERDADERO)
        SINO
                devolver (FALSO)
        FINSI
FIN

operacion DivideYVenceras (D: array[entero]; inicio,final: entero): Solucion
        SI Pequeno(inicio, final) = VERDADERO ENTONCES
                devolver (SolucionDirecta (D, inicio, final))
        SINO
                sol1: Solucion
                sol2: Solucion
                pm: entero
                pm:= (inicio+final)/2

        sol1:= DivideYVenceras (D, inicio, pm)
        SI (sol1.diferencia = m*25) ENTONCES
                devolver (sol1)
        FINSI

        sol2:= DivideYVenceras (D,pm+1, final)
        SI  (sol2.diferencia = m*25) ENTONCES
                devolver (sol2)
        FINSI
        devolver (CombinarSoluciones (sol1, sol2, pm-m+2, pm+m-1, D))
        FINSI
FIN
```

Para resolver el problema mediante divide y vencerás debemos crear una operación recursiva que vaya dividiendo el problema en subproblemas más pequeños hasta que sean lo suficientemente pequeños. Nosotros hemos decidido establecer 2*m cómo un tamaño de subproblema suficientemente pequeño porque no es recomendable tener un tamaño demasiado grande, pero por otra parte si se escoge m y al dividir un subproblema este queda más pequeño que m aquí habría un problema, pues esa solución no sería válida. Con 2*m este problema se anula.
Por otra parte, el tipo Solucion está creado para representar las soluciones del problema de una manera más fácil a la hora de programarlo en C++ (porque las funciones de C++ solo devuelven un tipo de dato). El algoritmo recursivo se divide en dos subproblemas, uno por la izquierda y otro por la derecha, del mismo tamaño (esto es, se divide por el centro). Para que sea más eficiente, al hacer la llamada recursiva, después de que se haya calculado la solución de cada subproblema lo primero que hace el algoritmo es comprobar si esa solución es la óptima, es decir, que haya encontrado una subcadena de diferencia total 25*m. Si bien sol1 o bien sol2 es una solución óptima entonces se devuelve y se evita el hacer ka operación de combinar (y la segunda llamada de recurrencia, si se ha descubierto en el subproblema izquierdo).Si esto no sucede, una vez se han calculado ambas soluciones se combinan estas y los (2*m)-2 elementos del centro del array, porque estos, al ser divididos por la mitad podrían contener la solución y ser cortados. Para ello cogemos los m-1 elementos de la izquierda desde la mitad (en los que se incluye el punto medio, pm) y los m-1 de la derecha.

```
operacion SolucionDirecta (D: array[entero]; inicio, final: entero): Solucion
        s: Solucion
        suma, pos: entero
        suma:=0
        pos:=0
        PARA k:= inicio…inicio+m-1 HACER
                suma:= suma + D[k]
        FINPARA

        s.diferencia:= suma
        s.posicion:= inicio
        i,j: entero
        i:= inicio
        j:= inicio+m
        MIENTRAS (j<=final) HACER
                suma:= suma - D[i] + D[j]
```

```
            SI (suma > s.diferencia) ENTONCES
                s.diferencia:= suma
                s.posicion:= i+1
            FINSI
            i:= i+1
            j:= j+1
        FINMIENTRAS

        devolver (s)
FIN
```

La solución directa debe encontrar en el array de <2*m elementos la combinación con más diferencia. Para ello, calculamos primero la diferencia de los m primeros elementos y después vamos iterando sobre el array, restando un elemento por la izquierda, sumando uno por la derecha y comparando con la combinación más alta anterior hasta llegar al fin del subproblema. Con este método estamos comprobando todas las posibles combinaciones de m elementos del subproblema.

```
operacion CombinarSoluciones (sol1, sol2: Solucion; c1, c2: entero; D:
        array[entero]): Solucion

        solMedio: Solucion
        solMedio:= SolucionDirecta(D, c1, c2)

        SI (sol1.diferencia >= sol2.diferencia) ENTONCES
            SI (sol1.diferencia >= solMedio.diferencia) ENTONCES
                devolver (sol1)
            SINO devolver (solMedio)
            FINSI
        SINO
            SI (sol2.diferencia >= solMedio.diferencia) ENTONCES
                devolver (sol2)
            SINO devolver (solMedio)
            FINSI
        FINSI
FIN
```

Para combinar soluciones se calcula la combinación más alta de los 2*m-2 elementos del medio y se compara esta con los 2 subproblemas originales. Para calcular la del medio reutilizamos la misma operación SolucionDirecta porque si se aplicara el método recursivo de Divide y Vencerás, al ser el problema de menos de 2*m elementos, este haría la solución directa, y

tampoco tiene sentido copiar el código de la solución directa otra vez dentro de esta operación. Una vez calculada simplemente se decide cuál de las 3 es la mayor y se devuelve.

Para usar este algoritmo para resolver el problema propuesto primero hay que calcular el array D de diferencias y establecer una constante o variable de solo lectura global que contenga m y hacer una primera llamada DivideYVenceras(D,0,n-1), siendo n el tamaño de las cadenas A y B (es decir, n-1 es el último elemento de D).

3. Estudio teórico del algoritmo

Para obtener el tiempo de ejecución y el orden de magnitud del algoritmo de divide y vencerás primero hay que obtener el tiempo de ejecución de las operaciones auxiliares.

$$t_{Pequeno}(n) = 2 \Rightarrow t_{Pequeno} \in O(1)$$

$$t_{SolucionDirecta}(n,m) = 10 + \sum_{i=1}^{m}(2) + \sum_{i=m}^{n}(\frac{1}{2}2 + 5) =$$
$$= 10 + 2m + 6 + 6n - 6m = 16 + 6n - 4m$$
$$t_{SolucionDirecta} \in O(n)$$

$$t_{CombinarSoluciones}(m) = 5 + t_{SolucionDirecta}(2m-2, m) =$$
$$= 21 - 12 + 12m - 4m = 9 + 8m$$
$$t_{CombinarSoluciones} \in O(m)$$

Pequeno es simplemente un condicional que comprueba una condición y devuelve verdadero o falso con lo cual su tiempo es constante. SolucionDirecta tiene varias asignaciones inmediatas de orden constante y luego tiene dos bucles (sumatorios), uno que hace m repeticiones y otro que hace n-m repeticiones. Dentro de este segundo hay una comprobación de condición, no obstante hemos decidido contar que en la mitad de las ejecuciones se ejecutan las instrucciones de dentro de la condición porque son solamente 2 instrucciones y contar casos mejores y peores complica bastante el cálculo teórico del tiempo de ejecución del algoritmo, porque esto se extendería hasta el algoritmo principal. En CombinarSoluciones se realizan unas instrucciones de comprobación pero lo más importante es el cálculo de la solución del medio de la cadena. Para ello se llama a SolucionDirecta. Como en cualquier ejecución de esta operación en la llamada a SolucionDirecta n = 2m-2 hemos decidido sustituirla en la fórmula y así obtener el tiempo de ejecución de CombinarSoluciones sólo en función de m.

$$t_{M:\,DivideYVenceras}(n,m) = \left\{2 + t_{SolucionDirecta}(n,m) \quad Si\ n \leq 2m\right\}$$

$$= \left\{10 + 2t(\tfrac{n}{2}) + t_{CombinarSoluciones}(m) \quad Si\ n > 2m\right\}$$

$$t_M(n) = 2\,t(\tfrac{n}{2}) + 19 + 8m = 2\,t(\tfrac{n}{2}) + 19 + \tfrac{8}{1000}m =$$

$$\Rightarrow (n = 2^k) \Rightarrow t(2^k) = 2t(2^{k-1}) + (19 + \tfrac{8}{1000}m) \Rightarrow$$

$$(x-2)\cdot(x-2) = 0 \Rightarrow t_k = C_1 \cdot 2^k + C_2 \cdot 2^k \cdot k \Rightarrow$$

$$t_M(n) = C_1 \cdot 2^{log_2 n} + C_2 \cdot 2^{log_2 n} \cdot log_2 n$$

$$t_M(n) = C_1 \cdot n + C_2 \cdot n \cdot log\, n$$

$$t_M(n) \in O(n \cdot log\, n)$$

Para resolver esta ecuación de recurrencia había que hacer un cambio de variable de n=2^k. Para cambiar m hemos supuesto que, como dice el enunciado, para valores suficientemente grandes de n, m=n/1000. El tiempo peor del algoritmo es cuando en todo el array de diferencias no encuentra ningún subproblema de diferencia total m*25 (ya que estas se comprueban y se lanzan hacia arriba sin combinar con sus subproblemas hermanos), por

tanto el peor caso es cuando siempre se ejecuta la operación CombinarSoluciones.

$$t_{m:\,DivideYVenceras}(n,m) = \left\{2 + t_{SolucionDirecta}(n,m) \quad Si\ n \leq 2m\right\}$$

$$= \left\{8 + t(\tfrac{n}{2}) \;\; Si\ n > 2m\right\}$$

$$t_m(n) = t(\tfrac{n}{2}) + 8 =$$

$$\Rightarrow (n = 2^k) \Rightarrow t(2^k) = t(2^{k-1}) + 8 \cdot 1^k$$

$$(x-1)\cdot(x-1) = 0 \Rightarrow t_k = C_1 \cdot 1^k + C_2 \cdot 1^k \cdot k \Rightarrow$$

$$t_M(n) = C_1 \cdot 1^{log_2 n} + C_2 \cdot 1^{log_2 n} \cdot log_2 n$$

$$t_m(n) \in O(log\, n)$$

El mejor caso es cuando el primer subproblema que se resuelve por solución directa es de diferencia m*25 (esto es, cuando hay una subcadena de m ‘a’ y otra de m ‘z’ en las cadenas de origen justo al principio de ambas). Cuando esto ocurre, solo se realizan las llamadas izquierdas a subproblemas desde DivideYVenceras hasta que retorna a la primera llamada, con lo cual solo hay que tener en cuenta una llamada recursiva, excluyendo la otra y el método CombinarSoluciones.

4. Programación del algoritmo

Para no hacer demasiado extensa esta memoria el código fuente de C++ no se incluye. Este está contenido en su totalidad en el fichero “problema2.cpp” que se incluye en la entrega de esta práctica. El código está comentado explicando qué es cada función o trozo de código y que es lo que hace. En cuanto a como se ejecuta el programa, primero hay que compilarlo con g++ prueba2.cpp -o <<*nombre*>> (aunque se incluye un fichero precompilado y listo para ejecutar). Para hacer una ejecución hay que darle una entrada al programa, bien por la terminal, con teclado, o con un fichero con el siguiente formato:

d
t1 n1 m1

…..
td nd md

Donde d es el número de experimentos que se quieren realizar, después d líneas, una por cada experimento, indicando si se quiere probar el mejor caso (t = 1), el caso promedio (t=2) o el peor caso (t=3), y la n y la m con la que se quiere realizar el experimento. También se incluye un fichero con unos casos de prueba que hemos utilizado para realizar el estudio experimental del tiempo de ejecución.
IMPORTANTE: en la salida se imprime la posición en la que se ha encontrado la solución como posición+1, para que la salida quede igual que la que se esperaba en el enunciado de la práctica, donde el índice de las cadenas comenzaba en 1 (Si no lo hicieramos, lógicamente, empezaría en 0, como todos los índices de arrays en C++).

5. Validación del algoritmo

Para saber que el algoritmo funciona correctamente utilizamos la forma en la que el programa recibe casos para probar varios casos a la vez, casos promedio aleatorios con n y m muy dispares, desde lo más pequeño a las n que el enunciado dice que se deben probar (entre diez mil y 1 millón), también con el caso promedio y el peor caso. Pero lógicamente no vale con ejecutarlo sin más, hay que comprobar que las soluciones cuadran con lo que debería dar. Para comprobarlo comparamos la salida del algoritmo divide y vencerás con la salida que devuelve solución directa si se le llama por separado, una vez validamos que este funcionaba. Para validar el algoritmo de solución directa comprobamos a mano que sin importar el tamaño o contenido de la entrada el algoritmo funcionaba, y al ser un algoritmo que no compara subproblemas, ni hace llamadas recursivas ni divide cadenas sino que es mucho más simple se puede comprobar que su funcionamiento es correcto. A partir de aquí comparamos las salidas de ambos algoritmos.

6. Estudio experimental

Para cada tipo de caso usaremos generadores diferentes:

Caso promedio

En en caso promedio simplemente genera 2 cadenas aleatorias completamente. Para ello concatena una letra aleatoria (la suma del código ascii del carácter ‘a’ con un valor aleatorio entre 0 y 26 (así puede coger todas las letras del abecedario del que disponemos)) n veces a cada cadena. El código del generador es:

```
void Generador_promedio (string &a, string &b, int n)
{
	srand(time(NULL));
	char r;
	for (int i = 0; i < n; i++)
	{
		r = 'a' + rand()%26;
		a += r;
		r = 'a' + rand()%26;
		b += r;
	}
}
```

Mejor caso

Para obtener el mejor caso usamos un generador distinto, en el que para la cadena A se generan m a’s al principio de esta, y para la cadena B se generan m ‘z’ al principio de esta. El resto de la cadena se genera de manera aleatoria. De modo que si tenemos una n y m de 10 y 3 respectivamente, se genera la cadena1 tal que aaa******* y la cadena2 tal que zzz******* (siendo * una letra aleatoria entre la a y la z). El código de este generador es:

```
void Generador_mejor (string &a, string &b, int n)
{
	for (int i = 0; i < m; i++)
	{
		a += 'a';
		b += 'z';
	}

	srand(time(NULL));
	char r;
	for (int i = m; i < n; i++)
	{
		r = 'a' + rand()%26;
		a += r;
		r = 'a' + rand()%26;
		b += r;
	}
}
```

Peor caso

Hemos decidido que la mejor manera para obtener el peor caso es que no se pueda generar ninguna subcadena de diferencia m*25. Para esto solo hemos modificado el generador del caso promedio, haciendo que solo pueda concatenar a las cadenas letras de la a a la y , sumando un número aleatorio mod 25, evitando que se pueda generar pares a-z de una cadena a otra y que por tanto estos pares puedan producirse m veces seguidas (lo que daría lugar a el mejor caso posible). El código de este generador es:

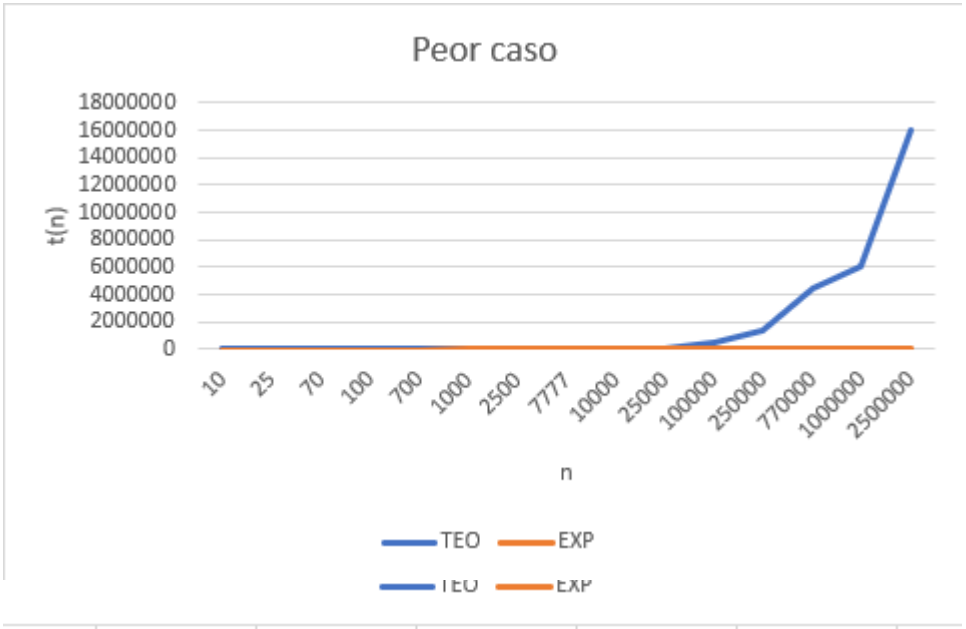
```
void Generador_peor (string &a, string &b, int n) {
	srand(time(NULL));
	char r;
	for (int i = 0; i < n; i++)
	{
		r = 'a' + rand()%25;
		a += r;
```

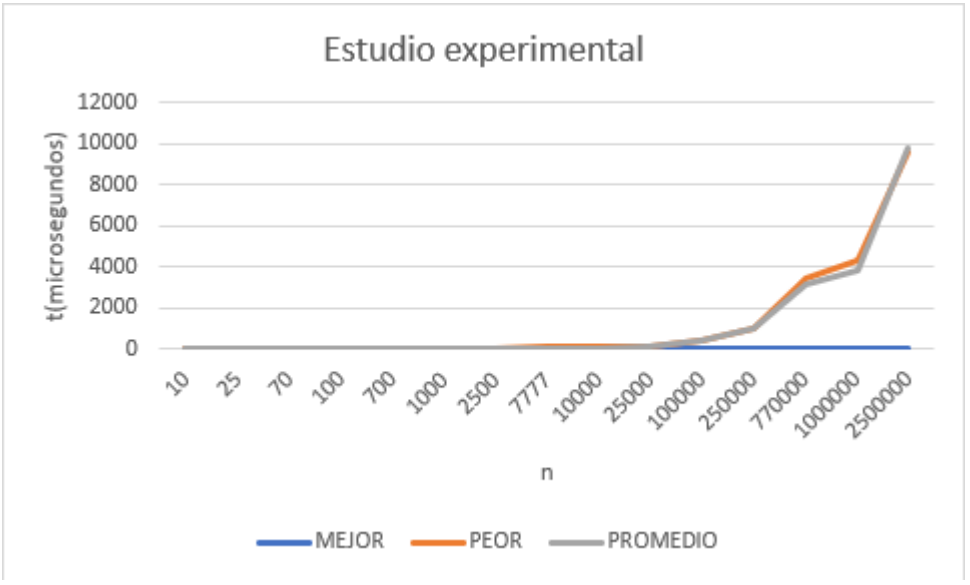
```
                r = 'a' + rand()%25;
                b += r;
        }
}
```

Para estudiar el tiempo de ejecución del programa hemos usado los métodos de la librería sys/time.h para medir el tiempo del sistema entre ejecuciones. El código que usamos está comentado en el fichero de código fuente para que quede constancia de cómo lo utilizamos. Se han probado 15 casos para el peor caso, 15 para el mejor caso y 15 para el caso promedio. El fichero con la entrada de todos estos casos es “pruebas.txt” y se encuentra también en la entrega de esta práctica. Los casos que se probaron fueron:

n=10 m=5
n=25 m=5
n=70 m=7
n=100 m=50
n=700 m=70
n=1000 m=10
n=2500 m=25
n=7777 m=7
n=10000 m=10
n=25000 m=25
n=100000 m=100
n=250000 m=250
n=770000 m=770
n=1000000 m=1000
n=2500000 m=2500

7. Contraste entre los resultados del estudio experimental y el teórico





De estos tres gráficos se pueden sacar varias conclusiones: lo primero es que, curiosamente, el estudio teórico sí que nos da una aproximación del tiempo de ejecución en caso del tiempo mejor (salvo algún valor anómalo en el estudio experimental), sin embargo, el peor caso, si bien acota superiormente a lo que realmente sucede, está lejos de ser una cota fiable, pues arroja resultados astronómicamente superiores a los tiempos recogidos

por el estudio experimental. Esto se puede deber a varios factores pero el principal es sin duda que los compiladores modernos (como g++ en este caso) optimizan el código para traducirlo a ensamblador de una forma a la que nosotros no podríamos ni acercarnos, haciendo que lo que nosotros pensabamos teóricamente que tardaría varios segundos ha tardado apenas unos milisegundos. Eso sí, si comparamos las gráficas de peor caso y la de el estudio experimental (las dos últimas) vemos que realmente sí se aprecia que el orden de ejecución es similar, pues las gráficas crecen con una forma parecida (pero con números mucho más altos).

Por otro lado observamos que el caso promedio en el que se generan cadenas aleatorias está prácticamente igualado al peor caso. Esto nos dice claramente que la probabilidad de que se genere una solución óptima de 25*m es ínfima, dado que debe generar m veces el mismo número aleatorio (no uno cualquiera sino 0 y 25, respectivamente en cada cadena), y esto, según m va haciéndose más grande tiene una probabilidad muy pequeña.

8. Conclusiones

Con esta práctica creemos haber comprendido los fundamentos de la programación de algoritmos con la técnica de divide y vencerás. No obstante, en cuanto al análisis de la eficiencia del algoritmo nos quedamos con dudas porque no entendemos la diferencia abismal en el peor caso entre el estudio teórico y el experimental, porque dudamos si realmente la optimización de un compilador puede llegar a ese punto o es que a lo mejor nosotros hemos cometido un error obteniendo el tiempo teórico por caso. Nos hubiera gustado hacer unas pruebas más exhaustivas, quizás con unos cuantos cientos de casos, pero por tiempo (no solo de generación sino también de análisis, para formar las hojas de cálculo y las gráficas) es algo que no hemos podido realizar pues nos hubiera supuesto un trabajo enorme. Si entendemos la improbabilidad de que se genere el mejor caso y creemos poder afirmar que el orden de magnitud del algoritmo es de O(nlogn).

La práctica nos ha supuesto a los 2 compañeros en total aproximadamente cerca de 20 horas entre la planificación, diseño, programación, comprobación y estudio de eficiencia del algoritmo así como a la redacción de la memoria.