PRÁCTICA 2: ALGORITMOS DIVIDE Y VENCERÁS

Autores:
José Teodosio Lorente Vallecillos
Miguel Torres Alonso
Mario Soriano Morón

Índice

1. Ejercicio 1
   1.1 Diseño e implementación algoritmo solución del problema.
   1.2 Diseño de componentes, diseño en pseudocódigo e implementación del algoritmo DyV solución del problema.
   1.3 Análisis y comparación de la eficiencia de ambos.
   1.4 Ejemplos de uso de ambos algoritmos.

2. Ejercicio 2
   2.1 Diseño e implementación algoritmo solución del problema.
   2.2 Diseño de componentes, diseño en pseudocódigo e implementación del algoritmo DyV solución del problema.
   2.3 Análisis y comparación de la eficiencia de ambos.
   2.4 Ejemplos de uso de ambos algoritmos.

3. Ejercicio 3
   3.1 Diseño e implementación algoritmo solución del problema.
   3.2 Diseño de componentes, diseño en pseudocódigo e implementación del algoritmo DyV solución del problema.
   3.3 Análisis y comparación de la eficiencia de ambos.
   3.4 Ejemplos de uso de ambos algoritmos.

1. Ejercicio 1

1.1 Diseño e implementación algoritmo solución del problema.

Se tiene una ALU la cual no dispone de la operación de multiplicación. Sin embargo, necesitamos multiplicar un número natural i por otro número natural j, que notamos como i*j. Sabemos, por la definición matemática de la operación de multiplicación en los números naturales, que:

$$i * j = \begin{cases} i & \text{si j = 1} \\ i * (j - 1) + i & \text{si j > 1} \end{cases} \forall i, j \in N$$

Al no disponer de la operación de multiplicación, podríamos sumar i j veces, para $j > 1$.
Por tanto, nos quedaría implementado algo así:

```
int ejercicio1basico(int i, int j){
        int aux = 0;
        for(int m = 0; m < j; m++){
                i = i + aux;
        }
        return i;
}
```

1.2 Diseño de componentes, diseño en pseudocódigo e implementación del algoritmo DyV solución del problema.

Para poder aplicar la técnica Divide y Vencerás se deben cumplir los siguientes requisitos:

- El problema inicial debe poder ser divisible en subproblemas que tengan aproximadamente el mismo tamaño, independientes entre sí, que sean de la misma naturaleza del problema inicial y que se puedan resolver por separado.

- Las soluciones de los subproblemas deben poder combinarse entre sí para poder dar lugar a la solución del problema inicial.

- Debe existir un caso base en el que el problema sea ya indivisible o, en su defecto, un algoritmo básico que pueda resolverlo.

El diseño propuesto sería el siguiente:

- **División del problema en subproblemas:**

El problema se puede dividir en K = 1 subproblema. Para ello, hemos de tener en cuenta si j es par o impar. En el caso que j sea par, se divide j entre 2 y se suma i j/2 veces. Si j es impar, se suma i floor(j/2) veces y se vuelve a sumar i.

- **Existencia de caso base:**

En este problema el caso base se da cuando j <= 1; es decir, cuando el resultado de la "multiplicación" es la propia i (si j es 1) o el resultado es 0 (si j es 0).

- **Combinación de soluciones:**

Cuando j es par, la combinación es la suma de la subsolución consigo misma, mientras que cuando j es impar, la combinación es la suma de la subsolución consigo misma + i.

<u>El diseño del pseudocódigo del algoritmo con las descripciones dadas anteriormente sería el siguiente:</u>

```
Algoritmo S = DyV(i : integer, j : integer)
        Si j <= 1, hacer: # Caso base
                Si j == 1, hacer:
                        Devolver i
                Si j == 0, hacer:
                        Devolver 0
        En otro caso: # Caso general
                s1 = DyV(i, floor(j/2)) // floor redondea al entero
                Si j es par, hacer:
                        Devolver s1 + s1
                Si j es impar, hacer:
                        Devolver s1 + s1 + i
```

Implementación en C++:

```
#include <cstdlib> // Para usar srand y rand
#include <chrono>
#include <iostream>
#include <fstream> // Para usar ficheros
#include <math.h> // Para usar floor

using namespace std;

int OpMult(int x, int y){
  int aux;
  if(y <= 1){ // Caso base
                if(y == 1)
```

```
      aux=x;
    if(y == 0)
      aux=0;
  }else{ // Caso general
    int s1 = OpMult(x, floor(y/2)); // floor redondea al entero
            if(y%2==0)
                    aux=s1+s1;
            if(y%2!=0)
                    aux=s1+s1+x;
  }

  return aux;
}

int main(int argc, char *argv[]){
        int x=0, y=0, result=0;
  chrono::time_point<std::chrono::high_resolution_clock> t0, tf; // Para medir el
tiempo de ejecución
        unsigned long tejecucion; // tiempo de ejecucion del algoritmo en ms
        ofstream fsalida;

  if (argc != 2) {
                cerr<<"\nError: El programa se debe ejecutar de la siguiente
forma.\n\n";
                cerr<<argv[0]<<" NombreFicheroSalida\n\n";
                return 0;
        }

        // Abrimos fichero de salida
        fsalida.open(argv[1]);
        if (!fsalida.is_open()) {
                cerr<<"Error: No se pudo abrir fichero para escritura
"<<argv[1]<<"\n\n";
                return 0;
        }


        // Inicializamos generador de no. aleatorios
  srand(time(NULL));
  x = rand()%100;
  y = rand()%100;

        cerr << "Ejecutando operacion de la ALU para valores x: " << x << " y: " << y
<< endl;
```

```
        t0= std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que
comienza la ejecuciÛn del algoritmo
        result = OpMult(x, y); // Ejecutamos el algoritmo para tamaÒo de caso tam
        tf= std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que
finaliza la ejecuciÛn del algoritmo

        tejecucion= std::chrono::duration_cast<std::chrono::microseconds>(tf -
t0).count();

        cerr << "\tResultado x*y: " << result << "\nTiempo de ejec. (us): " <<
tejecucion <<endl;

        // Guardamos tam. de caso y t_ejecucion a fichero de salida
        fsalida<<tejecucion<<"\n";

        // Cerramos fichero de salida
        fsalida.close();

  return 0;
}
```

1.3 Análisis y comparación de la eficiencia de ambos.

Para conocer si la solución Divide y Vencerás propuesta es más eficiente que el algoritmo básico inicial, calcularemos la eficiencia de ambos métodos. El algoritmo básico del apartado 1.1 es un O(n), pues depende del número de iteraciones del bucle for que a su vez depende de la variable *j*. Para el método DyV, al ser recursivo, tenemos la siguiente ecuación recurrente en el caso general:

$$T(n) = T(n/2) + 1$$

Es una ecuación lineal no homogénea. Se resuelve haciendo un cambio de variable $n = 2^m$, luego $m = \log_2(n)$.
$T(2^m) = T(2^{m-1}) + 1$, pasamos las “T” a un lado.
$T(2^m) - T(2^{m-1}) = 1$. El polinomio característico de la parte homogénea es: $p_H(x) = x-1$

Para la parte no homogénea hacemos $1 = b_1^m * q_1(m)$, luego $b_1 = 1$, $q_1(m) = 1$ con grado $d_1 = 0$.
$p(x) = (x - 1) * (x - b_1)^{d1+1} = (x - 1)^2$, donde $R_1 = 1$ y $M_1 = 2$.

$t_m = c_{10}1^m m^0 + c_{11}1^m m^1$
Al deshacer el cambio de variable, nos queda:
$t_n = c_{10}\ 1^{\log(n)}\log(n)^0 + c^{11}1^{\log(n)}\log(n)^1$, luego el algoritmo es O(log(n)).

Dado el resultado de la eficiencia del método DyV el cual es O(log(n)), podemos afirmar que es más eficiente que el método básico con un O(n).

1.4 Ejemplos de uso de ambos algoritmos.

2. Ejercicio 2

2.1 Diseño e implementación algoritmo solución del problema.

Un número natural n se dice que es cuadrado perfecto si se corresponde con el cuadrado de otro número natural. Por ejemplo, 4 es cuadrado perfecto dado que $4=2^2$. También son cuadrados perfectos $25=5^2$, o $100=10^2$. Al no disponer de la función “sqrt()”, podríamos utilizar un bucle de 1 hasta n/2 para el caso general (n > 1) en el cual se multiplique la variable que itera por sí misma y se verifique si es igual a n.

Por tanto nos quedaría implementado algo así:

```
bool ejercicio2basico(int n){
        for(int i = 2; i <= (n/2); i++){
                if(i*i == n){
                        return true;
                }
        }
        return false;
}
```

2.2 Diseño de componentes, diseño en pseudocódigo e implementación del algoritmo DyV solución del problema.

Para poder aplicar la técnica Divide y Vencerás se deben cumplir los requisitos ya expuestos en el apartado 1.2.

El diseño propuesto sería el siguiente:

- **División del problema en subproblemas:**

El problema se puede dividir en K = n/2 subproblema. Para ello, hemos de tener en cuenta si **$(n/4)^2$** es mayor o inferior que n. En el caso que **n/4** sea mayor, se procede a realizar el mismo proceso con el subconjunto desde **2** hasta **(n/4 - 1)**, comprobando nuevamente si **$((n/4 - 1 + 2)/2)^2$**, es el cuadrado perfecto de **n**, repitiendo el proceso hasta que se encuentre o no el cuadrado perfecto de **n**, y empezamos desde 2 debido a la comprobación previa, si n es mayor a 3 ya que si n fuera 0,1,2 ó 3 no sería necesario seguir con la comprobación ya que para 0 y 1 es **n** mismo y para 2 y 3 no existe. Si **n/4** es inferior, es el subconjunto desde **n/4+1** hasta **n/2**, siguiendo el mismo procedimiento descrito anteriormente.

- **Existencia de caso base:**

En este problema el caso base se da cuando n <= 1, es decir, cuando n es 0 es cuadrado perfecto de sí mismo y lo mismo pasa cuando n es 1.

- **Combinación de soluciones:**

En este caso no es necesaria una combinación de soluciones ya que la solución es si procede un entero.

<u>El diseño del pseudocódigo del algoritmo con las descripciones dadas anteriormente sería el siguiente:</u>

```
f tiene como valor inicial floor(n/2) # valor final del subconjunto
i tiene como valor inicial 2 # valor inicial del subconjunto
Algoritmo C = DyV(i : integer, f : integer, n : integer)
    Si n <= 3, hacer:
        Si n < 2, hacer:
            Devolver n
        En otro caso:
            # No tiene cuadrado perfecto
    En otro caso:
        Si n = f ² , hacer:
            Devolver f
        En otro caso Si n = ( floor((i+f) / 2) )² , hacer:
            Devolver  floor((i+f) / 2)
        En otro caso Si n < ( floor((i+f) / 2))² , hacer:
            DyV( i,  floor((i+f) / 2)-1 , n)
        En otro caso Si n > ( floor((i+f) / 2))² , hacer:
            DyV( floor((i+f) / 2)+1, f-1, n)
```

2.3 Análisis y comparación de la eficiencia de ambos.

Para conocer si la solución Divide y Vencerás propuesta es más eficiente que el algoritmo básico inicial, calcularemos la eficiencia de ambos métodos. El algoritmo básico del apartado 2.1 es un O(n), pues depende del número de iteraciones del bucle for. Para el método DyV, al ser recursivo, tenemos la siguiente ecuación recurrente en el caso general:

$$T(n) = T(n/2) + 1$$

Es una ecuación lineal no homogénea la cual se resuelve haciendo un cambio de variable $n = 2^m$, luego $m = \log_2(n)$.
$T(2^m) = T(2^{m-1}) + 1$, pasamos las "T" a un lado.
$T(2^m) - T(2^{m-1}) = 1$. El polinomio característico de la parte homogénea es: $p_H(x) = x-1$

Para la parte no homogénea hacemos $1 = b_1^m * q_1(m)$, luego $b_1 = 1$, $q_1(m) = 1$ con grado $d_1 = 0$.
$p(x) = (x - 1) * (x - b_1)^{d1+1} = (x - 1)^2$, donde $R_1 = 1$ y $M_1 = 2$.

$t_m = c_{10}1^m m^0 + c_{11}1^m m^1$
Al deshacer el cambio de variable, nos queda:
$t_n = c_{10}\ 1^{\log(n)}\log(n)^0 + c^{11}1^{\log(n)}\log(n)^1$, luego el algoritmo es O(log(n)).

Dado el resultado de la eficiencia del método DyV el cual es O(log(n)), podemos afirmar que es más eficiente que el método básico con un O(n).

2.4 Ejemplos de uso de ambos algoritmos.

3. Ejercicio 3

3.1 Diseño e implementación algoritmo solución del problema.

Dado un número natural n, deseamos saber si existe otro número natural y de modo que n=y*(y+1)*(y+2). Por ejemplo, para n=60, existe y=3 de modo que:
y * (y+1) * (y+2) = 3 * 4 * 5 = 60. Podríamos utilizar este método básico para resolverlo:

```
int ejercicio3basico(int n){
  int y=0, aux=0;
```

```
  for(int i=1; i < n/2; i++){
    aux = i * (i+1) * (i+2);
    if(aux == n)
      y=i;
  }
  return y;
}
```

3.2 Diseño de componentes, diseño en pseudocódigo e implementación del algoritmo DyV solución del problema.

Para poder aplicar la técnica Divide y Vencerás se deben cumplir los requisitos ya expuestos en el apartado 1.1.

El diseño propuesto sería el siguiente:

- **División del problema en subproblemas:**

El problema se puede dividir en K = 2 subproblemas. El primer subproblema consta del subconjunto desde 1 hasta n/4 y nos decantamos por él en el caso de que **n/4*n/4+1*n/4+2** sea mayor que n, y se realiza una búsqueda en ese subconjunto de la secuencia que coincida con la igualdad. En el segundo subproblema nos decantamos por el si se produce lo contrario al primero, es decir, que **n/4*n/4+1*n/4+2** sea menor que n, con el mismo procedimiento anterior.

- **Existencia de caso base:**

En este problema el caso base se da cuando n < 6, puesto que para n = 6 existe y = 1 tal que 1 * 2 * 3 = 6. Es decir, cuando n < 6, n no tiene un número y tal que n = y * (y+1) * (y+2). El caso general se da cuando n >= 6.

- **Combinación de soluciones:**

En este caso no es necesaria una combinación de soluciones ya que la solución es si procede un entero.

El diseño del pseudocódigo del algoritmo con las descripciones dadas anteriormente sería el siguiente:

```
f tiene como valor inicial floor(n/2) # valor final del subconjunto
i tiene como valor inicial 1 # valor inicial del subconjunto
Algoritmo N = DyV(i : integer, f : integer, n : integer)
        Si n <= 6, hacer:
                Si n = 6, hacer:
                        Devolver 1
```

```
        En otro caso:
            # No tiene número natural que cumpla la igualdad
    En otro caso:
        Si n = f*(f+1)*(f+2) , hacer:
            Devolver f
        En otro caso Si n < f*(f+1)*(f+2) , hacer:
            Para i desde 1 hasta i < f/2, hacer: i++
                Si i*i+1*i+2=n, hacer:
                    Devolver i
        En otro caso Si n > f*(f+1)*(f+2) , hacer:
            Para i desde floor(f/2)+1 hasta i < f, hacer: i++
                Si i*i+1*i+2=n, hacer:
                    Devolver i
```

3.3 Análisis y comparación de la eficiencia de ambos.

Para conocer si la solución Divide y Vencerás propuesta es más eficiente que el algoritmo básico inicial, calcularemos la eficiencia de ambos métodos. El algoritmo básico del apartado 3.1 es un O(n), pues depende del número de iteraciones del bucle for. Para el método DyV, al ser recursivo, tenemos la siguiente ecuación recurrente en el caso general:

$$T(n) = 2T(n/4) + c \Rightarrow T(n) \in O(n^{(\log_4 2)}) = O(n^{0,5}) = O(\sqrt{n})$$

Dado el resultado de la eficiencia del método DyV el cual es $O(\sqrt{n})$, podemos afirmar que es más eficiente que el método básico con un O(n).

3.4 Ejemplos de uso de ambos algoritmos.