

Algorítmica

Práctica 2

Algoritmos
Divide y
Vencerás

Diseño de los algoritmos Divide y Vencerás (Multiplicación de la ALU)	2
Descripción del problema	2
Solución: Diseño de componentes	2
Solución: Diseño del algoritmo	3
Solución: Implementación	3
Estudio de eficiencia de la solución	4
Análisis del algoritmo bruto	5
Análisis del algoritmo DyV	5
Ejecuciones de los algoritmos para comprobar que resuelven ambos resuelven el problema:	6
Algoritmo Bruto	6
Algoritmo DyV	6
Diseño de los algoritmos Divide y Vencerás (Cuadrados Perfectos)	7
Descripción del problema	7
Solución: Diseño de componentes	7
Solución: Implementación	9
Estudio de eficiencia de la solución	9
Análisis del algoritmo base	10
Análisis del algoritmo DyV	11
Ejecuciones de los algoritmos para comprobar que resuelven ambos resuelven el problema:	12
Algoritmo Bruto	12
Algoritmo DyV	12
Diseño de los algoritmos Divide y Vencerás (Número Natural)	13
Descripción del problema	13
Solución: Diseño de componentes	13
Solución: Diseño del algoritmo	14
Solución: Implementación	15
Estudio de eficiencia de la solución	15
Análisis del algoritmo base	16
Análisis del algoritmo DyV	17
Ejecuciones de los algoritmos para comprobar que resuelven ambos resuelven el problema:	17
Algoritmo bruto	18
Algoritmo DyV	18
Conclusión	18

Diseño de los algoritmos Divide y Vencerás (Multiplicación de la ALU)

Descripción del problema

El primer algoritmo resuelve la multiplicación de la unidad aritmético-lógica de un microcontrolador de consumo ultrabajo 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:

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

Para resolver el problema de multiplicación, hemos implementado el siguiente algoritmo sin aplicar ninguna técnica como tal de diseño. Se podría decir que la imagen adjuntada a continuación se corresponde con el algoritmo básico.

```
46  int multALU(int i, int j){
47      if (j == 1)
48          return i;
49      else {
50          int result = 0;
51          for (int k = 0; k < j-1; k++)
52              result += i;
53          return result + i;
54      }
55  }
```

Solución: Diseño de componentes

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.

Los tres requisitos se dan en el problema de resolución de la multiplicación. Su diseño es el siguiente:

- **División del problema en subproblemas.** Tomando como número a multiplicar, el segundo parámetro, denotado como j . En primer lugar, dividimos dicho parámetro entre dos, para simplificar así su cálculo. Posteriormente, llamamos recursivamente a la función una sola vez para que realice la multiplicación pero únicamente con la

mitad del tamaño del problema (Posteriormente explicaremos el motivo de esto). Así hasta llegar al caso base ($j=1$).

- **Existencia de caso base.** En este algoritmo, el caso base se corresponde cuando el parámetro j pasado como argumento es 1, es decir, se está multiplicando por 1 y no hay que multiplicar (el resultado será directamente i). Obviamente al no ser un número natural, no se tiene en cuenta el 0 como caso base.
- **Combinación de soluciones:** Como mencionamos en la división en subproblemas, únicamente se llamaba a la función recursivamente para una mitad de la multiplicación, ya que, ahora, en la combinación de soluciones duplicaremos dicha mitad calculada por la función. Adicionalmente, debido al truncamiento que existe al dividir un número j impar (se multiplicaría por $j-1$), siendo éste entero, si es impar se le suma de nuevo i , para que entonces, se haya multiplicado correctamente por j .

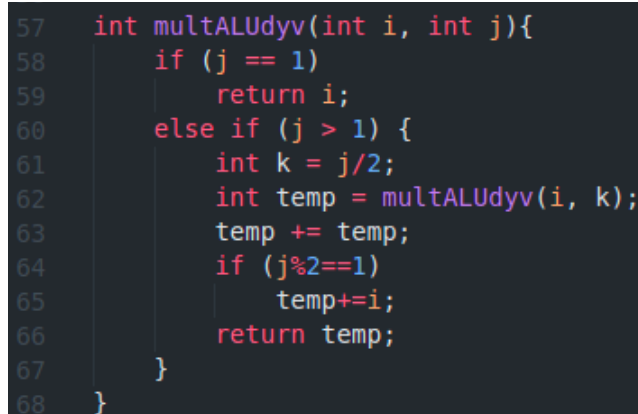
Solución: Diseño del algoritmo

```
int multALUdyv (int i, int j) {           //Tamaño del problema n, dado por j.
    si j es 1, hacer:                      // Caso base del problema
        devolver i
    si j es mayor que 1, hacer:            // Caso general del problema
         $k = j/2$                           // División en subproblema de tamaño  $n/2$ 
         $temp = multALUdyv(i, k)$ 

         $temp += temp$ ;                    // Combinación de las soluciones
        si j es impar, hacer:              // Si j es impar, se multiplicaría por  $j-1$ 
             $temp += i$                     // Debemos sumarle i para que sea por j
        devolver temp
```

Solución: Implementación

La implementación del algoritmo en C++ se corresponde con la siguiente figura, dónde apreciamos la técnica Divide y Vencerás para el algoritmo de multiplicación de dos naturales.



```
57  int multALUdyv(int i, int j){
58      if (j == 1)
59          return i;
60      else if (j > 1) {
61          int k = j/2;
62          int temp = multALUdyv(i, k);
63          temp += temp;
64          if (j%2==1)
65              temp+=i;
66          return temp;
67      }
68  }
```

Tomando dos números cualesquiera pertenecientes al conjunto de los números naturales, se realiza de forma recursiva la multiplicación de los dos elementos. Requiere tener en cuenta, que el algoritmo está diseñado para el conjunto de los naturales, cualquier otro caso de conjunto, no está recogido en dicho algoritmo, al igual que el cero que tampoco lo hemos incorporado a dicho conjunto.

Estudio de eficiencia de la solución

Para conocer si la técnica aplicada de diseño Divide y Vencerás, ha mejorado o no la implementación del algoritmo, realizaremos el estudio de la eficiencia tanto teórica como práctica y las dos en su conjunto, es decir la híbrida. En particular, la eficiencia del algoritmo base es $O(n)$, ya que es un algoritmo iterativo y con ello, la eficiencia viene determinada por la iteración del bucle, dónde se calcula el producto de los dos números tantas veces como j . Para el método DyV, al presentar una estructura de código, recursiva, tenemos la siguiente ecuación recurrente, en el caso general:

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

Como la ecuación no presenta las variables requeridas de una ecuación característica, tenemos que aplicar el siguiente cambio de variable, para poder convertirla. En este caso, $n = 2^m$, quedando dicha ecuación de la siguiente forma:

$$T(2^m) = T(2^{m-1}) + a$$

Resolvemos la **parte homogénea**:

$$T(2^m) - T(2^{m-1}) = 0$$

$$x^m - x^{m-1} = 0$$

$$x^{m-1} (x-1) = 0$$

$$P_H(x) = x-1$$

Parte no homogénea:

$$a = b \cdot q(m)$$

$$b = 1, q(m) = a \text{ (grado } d = 0)$$

$$P(x) = P_H(x)(x-b)^{d+1} = (x-1)^2 \text{ (Raíz 1 con multiplicidad } M = 2)$$

$$T(2^m) = \sum_{i=1}^r \sum_{j=0}^{M_i-1} c_{ij} R_i^m m^j = c_{10} 1^m m + c_{20} 1^m$$

Deshaciendo el cambio de variable

$$T(n) = c_1 \log_2(n) + c_2$$

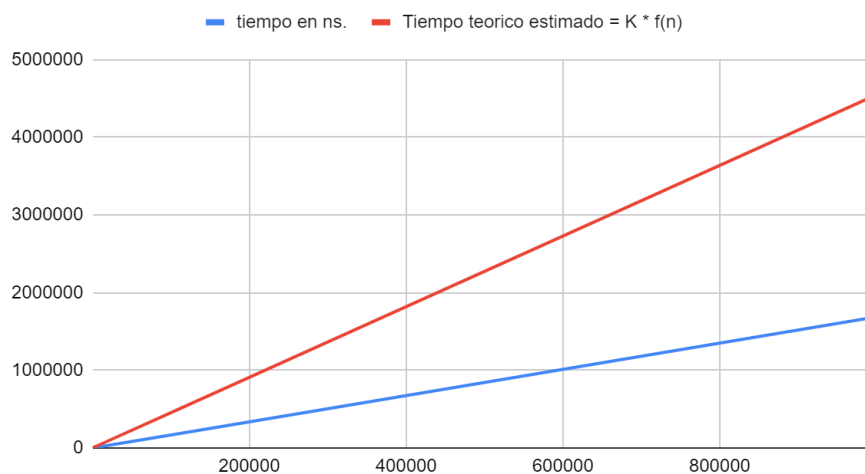
Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es **$O(\log_2(n))$**

Para complementar, el análisis de la eficiencia aprendido en la práctica anterior realizaremos una comparativa más detallada de dicha eficiencia, desde el punto de vista práctico e híbrido.

Análisis del algoritmo bruto

Tamaño	Tiempo en ns.	$K = \text{Tiempo} / F(n)$	Tiempo teórico estimado = $K * f(n)$
10	175	17,5	45,480095
100	309	3,09	454,80095
1000	1671	1,671	4548,0095
10000	16672	1,6672	45480,095
100000	167225	1,67225	454800,95
1000000	1687607	1,687607	4548009,5
	K Promedio	4,5480095	

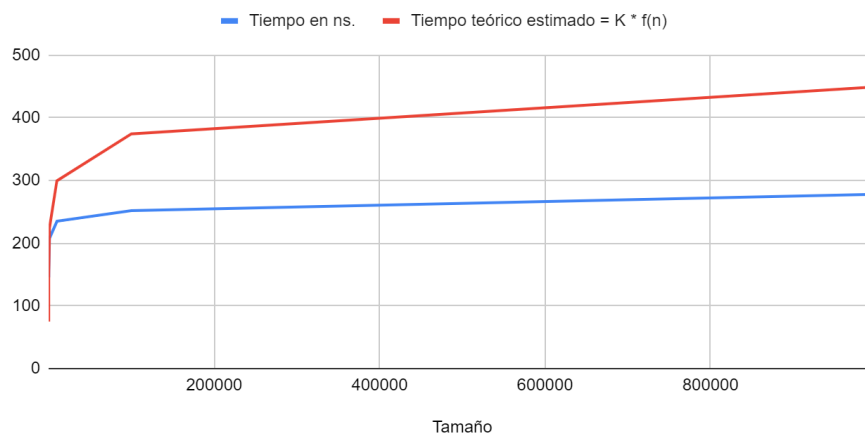
Tiempo en ns. y Tiempo teórico estimado = $K * f(n)$



Análisis del algoritmo DyV

Tamaño	Tiempo en ns.	$K = \text{Tiempo} / F(n)$	Tiempo teórico estimado = $K * f(n)$
10	146	146	74,88611111
100	157	78,5	149,7722222
1000	208	69,33333333	224,6583333
10000	235	58,75	299,5444444
100000	252	50,4	374,4305556
1000000	278	46,33333333	449,3166667
	K Promedio	74,88611111	

Tiempo en ns. y Tiempo teórico estimado = $K * f(n)$



La eficiencia teórica, queda acotando a la práctica en los dos algoritmos, tal y como se recoge en las gráficas. En conclusión, el algoritmo básico, presenta una eficiencia lineal, debido a su implementación, mientras que el algoritmo Divide y Vencerás presenta una eficiencia logarítmica. Tal y como hemos recogido, en las diferentes gráficas se puede observar que el segundo algoritmo es más eficiente que el primero, desde el punto de vista de los tres tipos de eficiencia.

Ejecuciones de los algoritmos para comprobar que resuelven ambos resuelven el problema:

Cabe aclarar que para cada ejercicio hemos creado un único fichero con código fuente el cual contiene ambos algoritmos (bruto y DyV), por lo que las ejecuciones tendrán el mismo nombre, aunque el código se habrá modificado para cada caso para que ejecute uno de los dos algoritmos. Se ha seguido pasando el parámetro *i* por argumento. No obstante, la *j* tendrá valores prefijados del 100 al 1000000. Los resultados son los siguientes:

Algoritmo Bruto

```
[alecoman@fedora DyV]$ ./ejercicio1 salidamult.txt 10
Tiempo de ejec. (us): 299 para tam. caso 100. Result = 1000
Tiempo de ejec. (us): 2014 para tam. caso 1000. Result = 10000
Tiempo de ejec. (us): 18572 para tam. caso 10000. Result = 100000
Tiempo de ejec. (us): 182366 para tam. caso 100000. Result = 1000000
Tiempo de ejec. (us): 1682748 para tam. caso 1000000. Result = 10000000
```

Algoritmo DyV

```
[alecoman@fedora DyV]$ ./ejercicio1 salidamult.txt 10
Tiempo de ejec. (us): 226 para tam. caso 100. Result = 1000
Tiempo de ejec. (us): 198 para tam. caso 1000. Result = 10000
Tiempo de ejec. (us): 225 para tam. caso 10000. Result = 100000
Tiempo de ejec. (us): 263 para tam. caso 100000. Result = 1000000
Tiempo de ejec. (us): 255 para tam. caso 1000000. Result = 10000000
```

Como podemos apreciar, los algoritmos realizan ambas multiplicaciones correctamente.

Diseño de los algoritmos Divide y Vencerás (Cuadrados Perfectos)

Descripción del problema

Se dispone de un algoritmo que resuelve el problema, de saber si un número entero es cuadrado perfecto o no lo es. Siendo este cuadrado perfecto si, al realizarle la raíz cuadrada sobre dicho número, es otro número natural. Para ello, tal y como se especificó en clase, no se podía usar la raíz cuadrada. Para ello realizamos otro método de comprobación, alternativo a la raíz.

```
45  bool Square(int n) {  
46      bool result = false;  
47      for (int i = 0; i < n && !result; i++){  
48          if (i*i == n)  
49              result = true;  
50      }  
51      return result;  
52  }
```

Solución: Diseño de componentes

En primer lugar, hemos recurrido a emplear intervalos delimitados por dos enteros (llamémoslos ini y fin) para así ir estimando aquel número entero cuyo cuadrado se corresponda con n (así daríamos con la solución del problema. La manera en la que hemos procedido para implementar el algoritmo divide y vencerás en este caso ha sido, comenzar con un intervalo de 0 a n, e ir dividiendo por la mitad dicho intervalo realizando una búsqueda binaria del entero objetivo, si es que existe.

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.

Los tres requisitos se dan en el problema de resolución de la multiplicación. Su diseño es el siguiente:

División del problema en subproblemas. Como bien se ha explicado anteriormente, comenzamos con un problema, en el que contamos con un intervalo de 0 a n, en el cual se contendrá, si es que existe, el entero cuyo cuadrado coincide con n. Iremos dividiendo dicho intervalo por la mitad, realizando búsquedas binarias, hasta así dar con el entero.

Existencia del caso base. En este algoritmo podríamos incluir tres casos base:

- El primero es que n sea 1, ya que es el único cuadrado perfecto cuya raíz es él mismo. Se concluye con una respuesta verdadera para el algoritmo.
- El segundo, tiene que ver con el intervalo empleado y será la condición para el cual no hemos podido encontrar el elemento y es que la diferencia entre los extremos sea igual o menor que 1. (Ya que si la raíz de n es entera y está en el intervalo, la diferencia entre ini y fin debería ser 2, por como hemos implementado el algoritmo). Se concluye con una respuesta falsa para el algoritmo.
- El tercer caso base trata de que el elemento mitad del intervalo sea justo la raíz de n . Por lo que habríamos finalizado el algoritmo con una respuesta verdadera.

Combinación de soluciones. La combinación de soluciones tratará de ir arrastrando a los intervalos anteriores (más grandes) los distintos casos base a los cuales puede llegar el algoritmo. Es decir, desde el más pequeño, volviendo hacia atrás hasta el intervalo original del problema, ir devolviendo el valor (true o false) devuelto por alguno de los casos base cuando se haya encontrado el elemento o no se pueda dividir más el intervalo. Finalmente nos quedará que la función del intervalo original nos habrá devuelto dicho valor.

Solución: Diseño del algoritmo

```
bool squareDyV (int n, int ini, int fin) {
    comprobamos que fin >= ini                // El intervalo es correcto
    si fin - ini <= 1, hacer:
        si fin == 1, hacer:
            devolver true                    // Caso base 2
        si fin no es 1, hacer:
            devolver false                    // Caso base 1

    si fin - ini > 1, hacer:
        mid = (ini+fin)/2, square = mid*mid
        si square (mid*mid) = n, hacer:
            devolver true                    // Caso base 3
        si square > n, hacer:
            buscar en intervalo mitad inferior    // squareDyV(n, 0, mid)
        si square < n, hacer:
            buscar en el intervalo mitad superior // squareDyV(n, mid, fin)
}
```

Solución: Implementación

La implementación del algoritmo en C++ se corresponde con la siguiente figura, dónde apreciamos la técnica Divide y Vencerás para el algoritmo de cuadrado perfecto.

```
54 bool squareDyV(int n, int ini, int fin) {  
55     if (fin >= ini){  
56         if (fin-ini <= 1){  
57             if (fin == 1) return true;  
58             else return false;  
59         }  
60         else {  
61             int mid = (ini+fin)/2, square = mid*mid;  
62             if (square == n) return true;  
63             else if (square > n) return squareDyV(n, 0, mid);  
64             else if (square < n) return squareDyV(n, mid, fin);  
65         }  
66     }  
67 }
```

Tomando dos números cualesquiera pertenecientes al conjunto de los números naturales, se realiza de forma recursiva la comprobación de si el elemento al cuadrado coincide con el elemento que estamos buscando. Requiere tener en cuenta, que el algoritmo está diseñado para el conjunto de los naturales, cualquier otro caso de conjunto, no está recogido en dicho algoritmo, al igual que el cero que tampoco lo hemos incorporado a dicho conjunto.

Estudio de eficiencia de la solución

Para conocer si la técnica aplicada de diseño Divide y Vencerás, ha mejorado o no la implementación del algoritmo, realizaremos el estudio de la eficiencia tanto teórica como práctica y las dos en su conjunto, es decir la híbrida. En particular, la eficiencia del algoritmo base es $O(n)$, ya que es un algoritmo iterativo y con ello, la eficiencia viene determinada por la iteración del bucle, dónde se calcula el producto de los dos números tantas veces como j . Para el método DyV, al presentar una estructura de código, recursiva, tenemos la siguiente ecuación recurrente, en el caso general:

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

Como la ecuación no presenta las variables requeridas de una ecuación característica, tenemos que aplicar el siguiente cambio de variable, para poder convertirla. En este caso, $n = 2^m$, quedando dicha ecuación de la siguiente forma:

$$T(2^m) = T(2^{m-1}) + a$$

Resolvemos la **parte homogénea**:

$$T(2^m) - T(2^{m-1}) = 0$$

$$x^m - x^{m-1} = 0$$

$$x^{m-1} (x-1) = 0$$

$$P_H(x) = x-1$$

Parte no homogénea:

$$a = b \cdot q(m)$$

$$b = 1, q(m) = a \text{ (grado } d = 0)$$

$$P(x) = P_H(x)(x-b)^{d+1} = (x-1)^2 \text{ (Raíz 1 con multiplicidad } M = 2)$$

$$T(2^m) = \sum_{i=1}^r \sum_{j=0}^{M_i-1} c_{ij} R_i^m m^j = c_{10} 1^m m + c_{20} 1^m$$

Deshaciendo el cambio de variable ($2^m = n$)

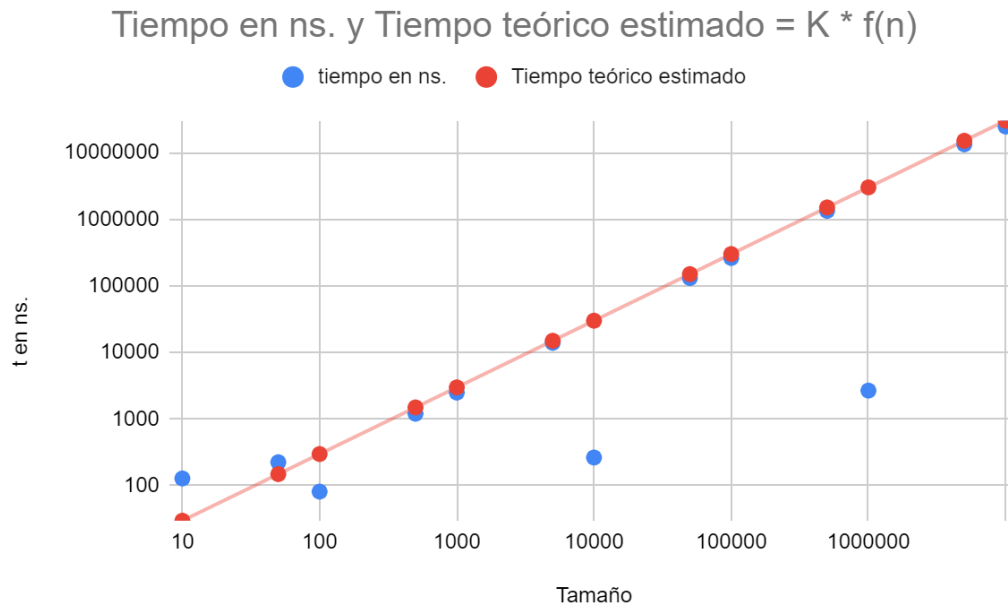
$$T(n) = c_1 \log_2(n) + c_2$$

Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es **$O(\log_2(n))$**

Para complementar, el análisis de la eficiencia aprendido en la práctica anterior realizaremos una comparativa más detallada de dicha eficiencia, desde el punto de vista práctico e híbrido.

Análisis del algoritmo base

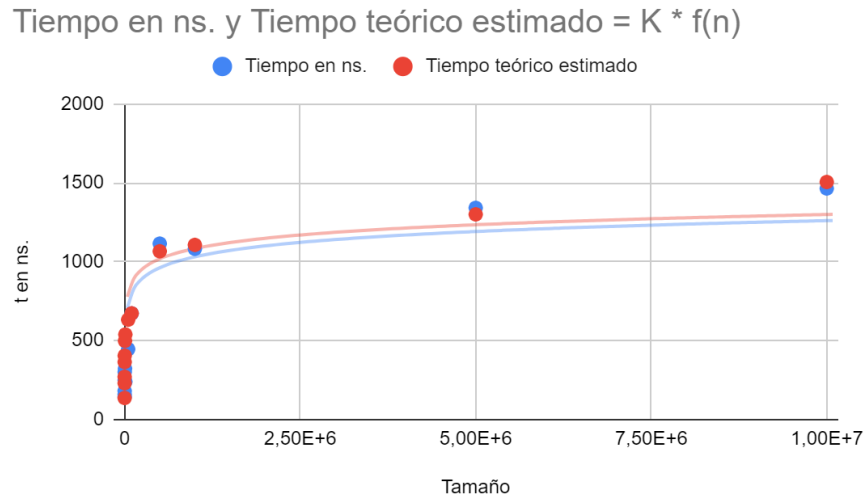
Tamaño	Tiempo en ns.	K = Tiempo / F(n)	Tiempo teórico estimado = K * f(n)
10	129	12,9	29,99513731
50	226	4,52	149,9756865
100	82	0,82	299,9513731
500	1208	2,416	1499,756865
1000	2504	2,504	2999,513731
5000	13986	2,7972	14997,56865
10000	266	0,0266	29995,13731
50000	130911	2,61822	149975,6865
100000	261754	2,61754	299951,3731
500000	1332580	2,66516	1499756,865
1000000	2683	0,002683	2999513,731
5000000	13261876	2,6523752	14997568,65
10000000	24539003	2,4539003	29995137,31
	K Promedio	2,999513731	



Los puntos de caída de tendencia, se corresponde a soluciones óptimas del algoritmo, y por eso provoca la caída significativa del tiempo.

Análisis del algoritmo DyV

Tamaño	Tiempo en ns.	$K = \text{Tiempo} / F(n)$	Tiempo teórico estimado = $K * f(n)$
10	142	142	134,2948079
50	157	92,40892988	228,1628504
100	179	89,5	268,5896159
500	247	91,51639314	362,4576583
1000	299	99,66666667	402,8844238
5000	321	86,78091459	496,7524663
10000	237	59,25	537,1792317
50000	444	94,48879214	631,0472742
100000	671	134,2	671,4740396
500000	1114	195,4739188	1065,342082
1000000	1081	180,1666667	1105,768848
5000000	1341	200,1800275	1299,63689
10000000	1464	209,1428571	1505,063655
50000000	1581	205,3521444	1633,931698
	K Promedio	134,2948079	



La eficiencia teórica, queda acotando a la práctica en los dos algoritmos, tal y como se recoge en las gráficas. En conclusión, el algoritmo básico, presenta una eficiencia lineal, debido a su implementación, mientras que el algoritmo Divide y Vencerás presenta una eficiencia logarítmica. Tal y como hemos recogido, en las diferentes gráficas se puede observar que el segundo algoritmo es más eficiente que el primero, desde el punto de vista de los tres tipos de eficiencia.

Ejecuciones de los algoritmos para comprobar que resuelven ambos resuelven el problema:

De nuevo, hemos automatizado los diferentes casos para comprobar si las raíces de dichos números son enteros.

Algoritmo Bruto

```
[alecoman@fedora DyV]$ ./ejercicio2 salidaCuadrado.txt
Tiempo de ejec. (ns): 170 para tam. caso 100. Result = true
Tiempo de ejec. (ns): 2227 para tam. caso 1000. Result = false
Tiempo de ejec. (ns): 303 para tam. caso 10000. Result = true
Tiempo de ejec. (ns): 262139 para tam. caso 100000. Result = false
Tiempo de ejec. (ns): 2521 para tam. caso 1000000. Result = true
Tiempo de ejec. (ns): 26041784 para tam. caso 10000000. Result = false
```

Algoritmo DyV

```
[alecoman@fedora DyV]$ ./ejercicio2 salidaCuadrado.txt
Tiempo de ejec. (ns): 170 para tam. caso 100. Result = true
Tiempo de ejec. (ns): 2227 para tam. caso 1000. Result = false
Tiempo de ejec. (ns): 303 para tam. caso 10000. Result = true
Tiempo de ejec. (ns): 262139 para tam. caso 100000. Result = false
Tiempo de ejec. (ns): 2521 para tam. caso 1000000. Result = true
Tiempo de ejec. (ns): 26041784 para tam. caso 10000000. Result = false
```

De nuevo, ambos algoritmos parecen realizar los cálculos correctamente, devolviendo como verdadero aquellos casos para el cual el tamaño del caso tiene justamente un número par de ceros (por lo tanto, tiene una raíz entera)

Diseño de los algoritmos Divide y Vencerás (Número Natural)

Descripción del problema

Se dispone de un algoritmo que resuelve el problema, de saber si para un número entero n , existe otro entero y , tal que se cumpla que $n = y*(y+1)*(y+2)$. Como algoritmo que resuelve dicho problema de manera bruta, comprobando todas las posibles soluciones, tenemos:

```
44  bool combinacion(int n) {
45      bool result = false;
46      for (int i = 0; i < n-2 && !result; i++){
47          if (i*(i+1)*(i+2) == n)
48              result = true;
49      }
50      return result;
51  }
```

Solución: Diseño de componentes

Tal y como hemos hecho en el algoritmo anterior de los cuadrados perfectos, también recurriremos al empleo de intervalos delimitados por dos enteros (ini y fin) para poder estimar el entero “ y ” anteriormente mencionado. También hemos ido dividiendo por la mitad los intervalos, comenzando por $ini = 0$ y $fin = n$, realizando la búsqueda binaria del entero objetivo, si es que existe.

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.

Los tres requisitos se dan en el problema de resolución de la multiplicación:

División del problema en subproblemas. Como hemos explicado anteriormente, comenzamos con un problema, en el que contamos con un intervalo de 0 a n , en el cual se contendrá, si es que existe, el entero tal que cumple la condición expuesta. Iremos dividiendo dicho intervalo por la mitad, realizando búsquedas binarias, hasta así dar con el entero, si es que existe.

Existencia del caso base. En este algoritmo podríamos incluir tres casos base:

- El primero es que n sea menor que 6, ya que el “ y ” más pequeño posible para que se cumpla la condición es 1 y $1*(1+1)*(1+2) = 6$. Para cualquier n por debajo de

dicho umbral no existe un entero que cumpla la condición. Se concluye con una respuesta falsa para el algoritmo.

- El segundo, tiene que ver con el intervalo empleado y será la condición para el cual no hemos podido encontrar el elemento y es que la diferencia entre los extremos sea igual o menor que 1. (Ya que si existe el entero "y" y está en el intervalo, la diferencia entre ini y fin debería ser 2, por como hemos implementado el algoritmo). Se concluye con una respuesta falsa para el algoritmo.
- El tercer caso base trata de que el elemento mitad del intervalo sea justo aquel entero que cumple con la condición. Por lo que habríamos finalizado el algoritmo con una respuesta verdadera.

Combinación de soluciones. La combinación de soluciones tratará de ir arrastrando a los intervalos anteriores (más grandes) los distintos casos base a los cuales puede llegar el algoritmo. Es decir, desde el más pequeño, volviendo hacia atrás hasta el intervalo original del problema, ir devolviendo el valor (true o false) devuelto por alguno de los casos base cuando se haya encontrado el elemento o no se pueda dividir más el intervalo. Finalmente nos quedará que la función del intervalo original nos habrá devuelto dicho valor.

Solución: Diseño del algoritmo

```
bool combinacionDyV (int n, int ini, int fin) {
    comprobamos que fin >= ini                // El intervalo es correcto
    si n < 6 o fin - ini < 1 hacer:              // Casos base 1 y 2
        devolver false;
    en otro caso, hacer:
        mid = (ini+fin)/2
        comb = mid*(mid+1)*(mid+2)
        si comb = n, hacer:
            devolver true                      // Caso base 3
        si comb > n, hacer:
            buscar en intervalo mitad inferior // combinacionDyV(n, 0, mid)
        si comb < n, hacer:
            buscar en el intervalo mitad superior //combinacionDyV(n, mid, fin)
}
```

Solución: Implementación

La implementación del algoritmo en C++ se corresponde con la siguiente figura, dónde apreciamos la técnica Divide y Vencerás para el algoritmo de cuadrado perfecto.

```
53  bool combinacionDyV(int n, int ini, int fin) {  
54      if (fin >= ini){  
55          if (n < 6) return false;  
56          else if (fin-ini <= 1) return false;  
57          else {  
58              int mid = (ini+fin)/2, comb = mid*(mid+1)*(mid+2);  
59              if (comb == n) return true;  
60              else if (comb > n) return combinacionDyV(n, ini, mid);  
61              else if (comb < n) return combinacionDyV(n, mid, fin);  
62          }  
63      }  
64  }
```

Tomando dos números cualesquiera pertenecientes al conjunto de los números naturales, se realiza de forma recursiva la comprobación de si el elemento al cuadrado coincide con el elemento que estamos buscando. Requiere tener en cuenta, que el algoritmo está diseñado para el conjunto de los naturales, cualquier otro caso de conjunto, no está recogido en dicho algoritmo, al igual que el cero que tampoco lo hemos incorporado a dicho conjunto.

Estudio de eficiencia de la solución

Para conocer si la técnica aplicada de diseño Divide y Vencerás, ha mejorado o no la implementación del algoritmo, realizaremos el estudio de la eficiencia tanto teórica como práctica y las dos en su conjunto, es decir la híbrida. En particular, la eficiencia del algoritmo base es $O(n)$, ya que es un algoritmo iterativo y con ello, la eficiencia viene determinada por la iteración del bucle, dónde se calcula el producto de los dos números tantas veces como j . Para el método DyV, al presentar una estructura de código, recursiva, tenemos la siguiente ecuación recurrente, en el caso general:

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

Como la ecuación no presenta las variables requeridas de una ecuación característica, tenemos que aplicar el siguiente cambio de variable, para poder convertirla. En este caso, $n = 2^m$, quedando dicha ecuación de la siguiente forma:

$$T(2^m) = T(2^{m-1}) + a$$

Resolvemos la **parte homogénea**:

$$T(2^m) - T(2^{m-1}) = 0$$

$$x^m - x^{m-1} = 0$$

$$x^{m-1} (x-1) = 0$$

$$P_H(x) = x-1$$

Parte no homogénea:

$$a = b \cdot q(m)$$

$$b = 1, q(m) = a \text{ (grado } d = 0 \text{)}$$

$$P(x) = P_H(x)(x-b)^{d+1} = (x-1)^2 \text{ (Raíz 1 con multiplicidad } M = 2 \text{)}$$

$$T(2^m) = \sum_{i=1}^r \sum_{j=0}^{M_i-1} c_{ij} R_i^m m^j = c_{10} 1^m m + c_{20} 1^m$$

Deshaciendo el cambio de variable ($2^m = n$)

$$T(n) = c_1 \log_2(n) + c_2$$

Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es **$O(\log_2(n))$**

Para complementar, el análisis de la eficiencia aprendido en la práctica anterior realizaremos una comparativa más detallada de dicha eficiencia, desde el punto de vista práctico e híbrido.

Análisis del algoritmo base

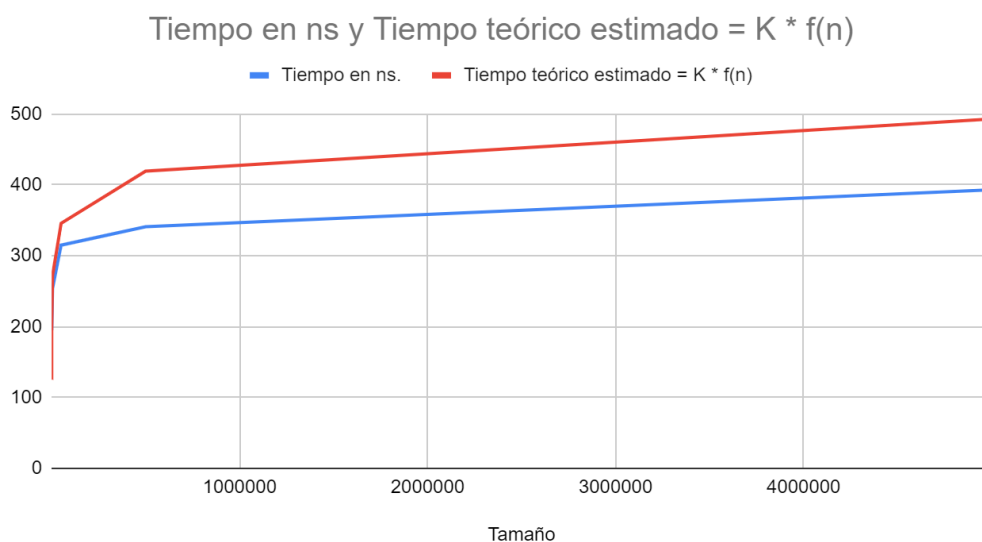
Tamaño	Tiempo en ns.	K = Tiempo / F(n)	Tiempo teórico estimado = K * f(n)
50	230	4,6	145,8691083
500	1326	2,652	1458,691083
5000	12814	2,5628	14586,91083
50000	125147	2,50294	145869,1083
500000	1285069	2,570138	1458691,083
5000000	13082075	2,616415	14586910,83
	K Promedio	2,917382167	

Tiempo en ns. y Tiempo teórico estimado = K * f(n)



Análisis del algoritmo DyV

Tamaño	Tiempo en ns.	$K = \text{Tiempo} / F(n)$	Tiempo teórico estimado = $K * f(n)$
50	195	114,7754225	125,0443505
500	196	72,62029577	198,6444436
5000	254	68,66776419	272,2445367
50000	315	67,03596739	345,8446299
500000	341	59,83537371	419,444723
5000000	393	58,66573514	493,0448161
	K Promedio	73,60009311	



La eficiencia teórica, queda acotando a la práctica en los dos algoritmos, tal y como se recoge en las gráficas. En conclusión, el algoritmo básico, presenta una eficiencia lineal, debido a su implementación, mientras que el algoritmo Divide y Vencerás presenta una eficiencia logarítmica. Tal y como hemos recogido, en las diferentes gráficas se puede observar que el segundo algoritmo es más eficiente que el primero, desde el punto de vista de los tres tipos de eficiencia.

Ejecuciones de los algoritmos para comprobar que resuelven ambos resuelven el problema:

Para estos dos algoritmos también se ha utilizado un número prefijado de casos, empezando por el 6 y multiplicando por 4 cada iteración. Esto se ha realizado para así forzar la aparición de los números 6 y 24, que cumplen la condición de que para el entero existe un y tal que $n = y*(y+1)*(y+2)$, para así ver que los algoritmos funcionan correctamente.

Algoritmo bruto

```
[alecoman@fedora DyV]$ ./ejercicio3 salidaComb.txt
Tiempo de ejec. (ns): 135 para tam. caso 6. Result = true
Tiempo de ejec. (ns): 86 para tam. caso 24. Result = true
Tiempo de ejec. (ns): 251 para tam. caso 96. Result = false
Tiempo de ejec. (ns): 1145 para tam. caso 384. Result = false
Tiempo de ejec. (ns): 4522 para tam. caso 1536. Result = false
Tiempo de ejec. (ns): 17546 para tam. caso 6144. Result = false
```

Algoritmo DyV

```
[alecoman@fedora DyV]$ ./ejercicio3 salidaComb.txt
Tiempo de ejec. (ns): 113 para tam. caso 6. Result = true
Tiempo de ejec. (ns): 147 para tam. caso 24. Result = true
Tiempo de ejec. (ns): 138 para tam. caso 96. Result = false
Tiempo de ejec. (ns): 135 para tam. caso 384. Result = false
Tiempo de ejec. (ns): 192 para tam. caso 1536. Result = false
Tiempo de ejec. (ns): 213 para tam. caso 6144. Result = false
```

De nuevo, podemos ver que los algoritmos solucionan el problema correctamente. Además, como se ha podido apreciar con los otros dos problemas planteados, se consigue reducir considerablemente el tiempo de ejecución con respecto al algoritmo “bruto” (o, en estos casos, iterativo) mediante la aplicación de la técnica Divide y Vencerás.

Conclusión

Como hemos podido observar en los tres problemas planteados en esta práctica, para todos se ha podido implementar un algoritmo iterativo que ha devuelto soluciones satisfactoriamente, otro equivalente pero esta vez aplicando la técnica de Divide y Vencerás, con la cual se han conseguido tiempos de ejecución y órdenes de eficiencia bastante mejores y que igualmente han resuelto los problemas.