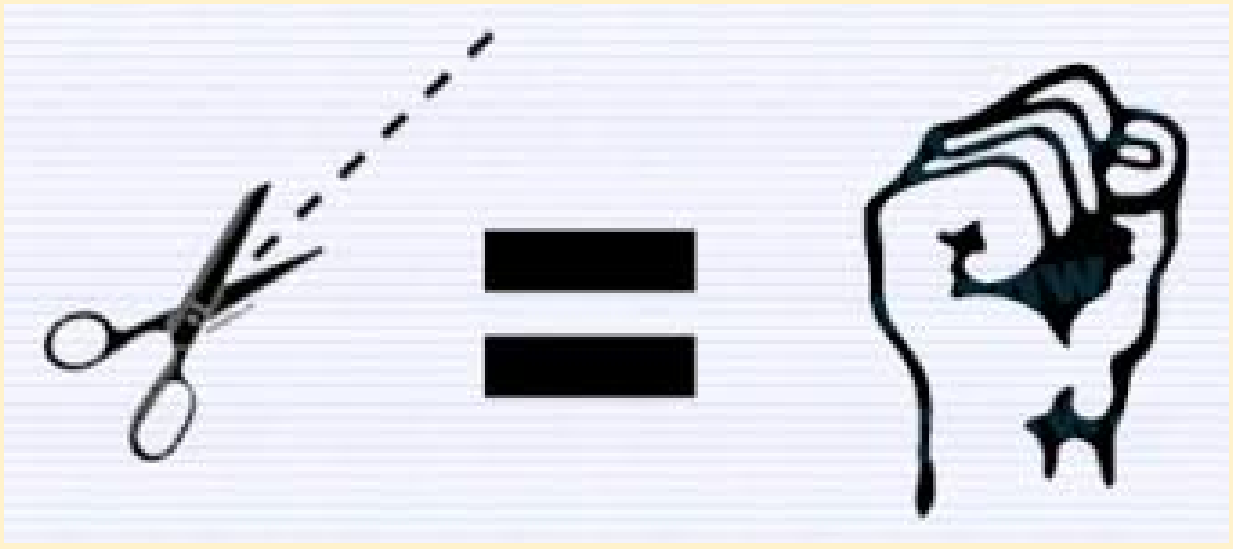
PRÁCTICA 2:

ALGORITMOS DIVIDE Y VENCERÁS



Carmona Molina, José Antonio.
Manzano Mata, Nuria.
Rodríguez Rodríguez, Antonio.

Algoritmo 1:

La unidad aritmético-lógica (ALU) 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:

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

Asumiendo como método básico la implementación directa de la fórmula anterior: ¿Es posible crear un algoritmo Divide y Vencerás más eficiente que la implementación de este algoritmo básico? Se asume que el número de bits de la ALU es desconocido pero fijo (por tanto, no se trata del problema de multiplicación de enteros largos).

1. Diseño del algoritmo básico:

Este algoritmo calcula el producto entre dos valores i y j sin hacer uso del operador *. Se crea un entero suma donde almacenar el valor del producto que en este caso se calculará mediante la suma de el valor j un total de i veces. Si alguno de los factores es 0 la suma dará 0 y si alguno de ellos es 1, el resultado será el del otro dígito. Ahora se itera sobre un bucle de tamaño i multiplicando j por sí misma i veces. Finalmente, se devuelve el resultado de la suma.

A simple vista podemos apreciar que la eficiencia de este algoritmo será O(n).

```
int multiplicador(int i, int j) {
    int suma = 0;
    if (j == 0 || i == 0) return 0;
    if (i == 1) return j;
    if (j == 1) return i;
    for (int n = 0; n < i; n++) {
        suma += j;
    }
    return suma;
}
```

2. Análisis y diseño DyV:

```
int multiplicadorDyV(int i, int j) {
    if (j == 0 || i == 0) return 0;
    if (i == 1) return j;
    if (j == 1) return i;

    int aux, resultado=0, izquierda, derecha;

    int mitad= j / 2;

    if (j%2 == 0) aux = 0;
    else aux = 1;

    izquierda = multiplicadorDyV(i, mitad);
    derecha = multiplicadorDyV(i, mitad + aux);
    resultado = izquierda + derecha;

    return resultado;
}
```

Para plantear una resolución de este algoritmo de manera recursiva hemos de basarnos en la estructura del algoritmo básico, convirtiendo ese bucle en una función recursiva, donde cada suma será un subproblema que repetiremos tantas veces como valor tenga j.

3. Análisis de Eficiencia:

ALGORITMO BÁSICO:

Teniendo en cuenta que el **tamaño del problema** dependerá de la variable i, el **peor caso posible** es que las variables i, j no sean ni 0 ni 1, con lo que, tendríamos que recorrer i veces el problema para ir sumando j (en sustitución de la multiplicación).

Es un **algoritmo básico,** que se compone de tres condicionales con una eficiencia teórica de O(1) y un bucle for con eficiencia O(n) para el cálculo de su adecuada multiplicación en el caso de que los valores de i, j no sean ni 0 ni 1.

Finalmente, con la regla del máximo, obtenemos que la eficiencia teórica de dicho algoritmo es de O(n).

DIVIDE Y VENCERÁS:

- Caso base: Se da cuando se cumplen las condiciones externas a la llamada recursiva de la función, haciendo algunos ajustes para asegurar el funcionamiento del algoritmo, como son meras asignaciones y comparaciones (primeros 3 if) , la eficiencia es O(1).

- Caso general: El resultado va a ser la suma entre los valores "izquierda" y "derecha", los cuales se obtienen a partir de una llamada recursiva respectivamente:
  - Tenemos las inicializaciones de las variables y una sentencia condicional cuya operación es O(1) porque las sentencias son operaciones simples.
  - Luego hace una **llamada recursiva**, para resolver un subproblema cuyo **tamaño es n/2**. Si T(n) es el tiempo que tarda el algoritmo para resolver el problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución **T(n/2)**.
  - Luego hace **otra llamada recursiva**, para resolver un subproblema hasta mitad+1 cuyo tamaño asintótico también se puede aproximar por n/2. Al igual que en la línea anterior, esta llamada recursiva tendrá un tiempo de ejecución **T(n/2)**.
  - Por último, como habremos obtenido un dos valores, bastará con una simple operación, sumando ambos, teniendo esto una eficiencia de orden O(1), al tratarse de operaciones simples.

  Por tanto, podemos aproximar T(n) en el caso general como **T(n)= 2T(n/2)+1**. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de MultiplicadorDyV.

  Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, 2^m , quedando:

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

  Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

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

  Resolvemos la "**parte homogénea**":

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

$$X^m - 2X^{m-1} = 0$$
$$(X^{m-1})\cdot(X-2) = 0$$

Y obtenemos la parte homogénea del polinomio característico como PH(X) = (X-2)

Para resolver la "**parte no homogénea**", debemos conseguir un escalar "b1 " y un polinomio "q1(m)". De modo que:

$$1 = b1\cdot q1(m)$$

Con lo que deducimos que b1 = 1 y q1(m) tendrá un grado d1 = 0.
El **polinomio característico** se obtiene como: P(X)=PH(X)=(X-2)(X-1)= (X-2)(X-1)

De este modo, tenemos r=2 dos raíces diferentes, con valores 1 y 2; y multiplicidades M = 1 en ambos casos.

Aplicando la fórmula de la **ecuación característica**, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

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

Ahora **deshacemos el cambio de variable** para volver al espacio de tiempos inicial:

$$T(n) = c_{10} n + c_{20}$$

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(n)**.

4. Implementación de los métodos y ejecuciones:

En este caso hemos utilizado un bucle anidado para comprobar tanto el método básico o por fuerza bruta como el divide y vencerás con distintos valores y ejecuciones de forma que entremos en todos los caso posibles (casos base y casos generales) de ambos algoritmos, recorriendo ambos bucles de forma opuesta o inversa (uno de 0 a 5 y otro de 5 a 0).

```cpp
int main () {

    cout << "Prueba para algoritmo básico: " << endl;
    for (int i = 0; i < 5; i++) {
        for (int j = 5; j > 0; j--) {
            cout << "Resultado de la multiplicación de " << i << " - " << j << ": " << multiplicador(i, j) << endl;
        }
    }
    cout << endl << endl << "Prueba para algoritmo DyV: " << endl;
    for (int i = 0; i < 5; i++) {
        for (int j = 5; j > 0; j--) {
            cout << "Resultado de la multiplicación de " << i << " - " << j << ": " << multiplicadorDyV(i, j) << endl;
        }
    }

    return 0;
}
```

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Documentos/Escritorio/Practica_2_ALG_DyV/ejercicios$ ./Ejercicio1
Prueba para algoritmo básico:
Resultado de la multiplicación de 0 - 5: 0
Resultado de la multiplicación de 0 - 4: 0
Resultado de la multiplicación de 0 - 3: 0
Resultado de la multiplicación de 0 - 2: 0
Resultado de la multiplicación de 0 - 1: 0
Resultado de la multiplicación de 1 - 5: 5
Resultado de la multiplicación de 1 - 4: 4
Resultado de la multiplicación de 1 - 3: 3
Resultado de la multiplicación de 1 - 2: 2
Resultado de la multiplicación de 1 - 1: 1
Resultado de la multiplicación de 2 - 5: 10
Resultado de la multiplicación de 2 - 4: 8
Resultado de la multiplicación de 2 - 3: 6
Resultado de la multiplicación de 2 - 2: 4
Resultado de la multiplicación de 2 - 1: 2
Resultado de la multiplicación de 3 - 5: 15
Resultado de la multiplicación de 3 - 4: 12
Resultado de la multiplicación de 3 - 3: 9
Resultado de la multiplicación de 3 - 2: 6
Resultado de la multiplicación de 3 - 1: 3
Resultado de la multiplicación de 4 - 5: 20
Resultado de la multiplicación de 4 - 4: 16
Resultado de la multiplicación de 4 - 3: 12
Resultado de la multiplicación de 4 - 2: 8
Resultado de la multiplicación de 4 - 1: 4


Prueba para algoritmo DyV:
Resultado de la multiplicación de 0 - 5: 0
Resultado de la multiplicación de 0 - 4: 0
Resultado de la multiplicación de 0 - 3: 0
Resultado de la multiplicación de 0 - 2: 0
Resultado de la multiplicación de 0 - 1: 0
Resultado de la multiplicación de 1 - 5: 5
Resultado de la multiplicación de 1 - 4: 4
Resultado de la multiplicación de 1 - 3: 3
Resultado de la multiplicación de 1 - 2: 2
Resultado de la multiplicación de 1 - 1: 1
Resultado de la multiplicación de 2 - 5: 10
Resultado de la multiplicación de 2 - 4: 8
Resultado de la multiplicación de 2 - 3: 6
Resultado de la multiplicación de 2 - 2: 4
Resultado de la multiplicación de 2 - 1: 2
Resultado de la multiplicación de 3 - 5: 15
Resultado de la multiplicación de 3 - 4: 12
Resultado de la multiplicación de 3 - 3: 9
Resultado de la multiplicación de 3 - 2: 6
Resultado de la multiplicación de 3 - 1: 3
Resultado de la multiplicación de 4 - 5: 20
Resultado de la multiplicación de 4 - 4: 16
Resultado de la multiplicación de 4 - 3: 12
Resultado de la multiplicación de 4 - 2: 8
Resultado de la multiplicación de 4 - 1: 4
```

Algoritmo 2:

Un número natural "n" se dice que es cuadrado perfecto si se corresponde con el cuadrado de otro número natural. Por ejemplo, 4 es perfecto dado que $4 = 2^2$. También son cuadrados perfectos $25 = 5^2$, o $100 = 10^2$. ¿Qué método básico/por fuerza bruta podemos diseñar para calcular si un número es cuadrado perfecto o no?

¿Es posible diseñar un algoritmo Divide y Vencerás para igualar (o mejorar) la eficiencia de este método básico?

1. Diseño del algoritmo básico:

La primera idea que desarrollamos para la implementación de la función CuadradoPerfecto fue hacer uso de la librería cmath y las funciones sqrt y pow. De esta forma iríamos en el primer caso comparando todos los cuadrados de los números desde 0 hasta n, comprobando si está o no entre ellos (Eficiencia O(n) debido al bucle). En el segundo intento, buscamos la diferencia entre la raíz cuadrada de n y la raíz cuadrada de la parte entera de n, de esta forma si la diferencia es 0, sabemos que se trata de un cuadrado perfecto.

El **problema** de este método reside en el desconocimiento de la eficiencia de las funciones pow y sqrt, de manera que las descartamos.

```
bool CuadradoPerfecto_conCmath1(int n) {
    bool cuadrado_perfecto = false;
    for (int i = 0; i <= int(sqrt(n)); i++) {
        if (n == pow(i, 2)) {
            cuadrado_perfecto = true;
            i = n;
        }
    }
    return cuadrado_perfecto;
}

bool CuadradoPerfecto_conCmath2(int n) {
    bool cuadrado_perfecto = false;
    if (sqrt(n)-int(sqrt(n))== 0) cuadrado_perfecto = true;
    return cuadrado_perfecto;
}
```

Por otra parte, ahora sin librerías, desarrollamos la siguiente función:

```
bool CuadradoPerfecto(int n) {
    bool cuadrado_perfecto = false;
    if (n == 0 || n == 1) cuadrado_perfecto = true;
    else {
        int valor = 0;
        for (int i = 2; valor <= n; i++) {
            valor = i*i;
            if (valor == n) cuadrado_perfecto = true;
        }
    }
    return cuadrado_perfecto;
}
```

Este algoritmo básico se va a encargar de comparar si el cuadrado de alguno de los enteros desde 0 hasta el valor de dicha potencia es igual a "n". Esta condición la utilizamos para evitar recorrer todos los valores desde 0 a n, ya que todos los valores de i cuyos cuadrados superan a n van a ser iteraciones inútiles.

2. Análisis y diseño DyV:

Para la implementación de un algoritmo DyV para la función CuadradoPerfecto hicimos uso del algoritmo básico, pero en este caso aplicamos una búsqueda binaria recursiva, consiguiendo una eficiencia logarítmica para esta función.

```
bool CuadradoPerfectoDyV(const int n, int izquierda, int derecha) {

    if (n == 0 || n == 1) return true;

    int centro = (izquierda + derecha) / 2;
    int valor = centro*centro;

    if (valor == n) return true;

    if (izquierda == derecha) return false;
    if (valor > n)
        derecha = centro-1;
    else
        izquierda = centro+1;

    return CuadradoPerfectoDyV(n, izquierda, derecha);
}
```

3. Análisis de Eficiencia:

ALGORITMO BÁSICO:

Teniendo en cuenta que el **tamaño del problema** dependerá de la variable n, el **peor caso posible** es que i y n/2 sean iguales como sucederían con la búsqueda del cuadrado perfecto de n=4 ya que, i=2 y n/2 = 2.

Es un **algoritmo básico,** que se compone de una condición con una eficiencia teórica de O(1) y otra condición que se compone internamente por un bucle for con eficiencia O(n) que va comprobando con otro condicional si se ha encontrado el cuadrado perfecto que buscamos, cuya eficiencia es 0(1).

Finalmente, con la regla del máximo, obtenemos que la eficiencia teórica de dicho algoritmo es de O(n).

DIVIDE Y VENCERÁS:

Teniendo en cuenta que el **tamaño del problema** dependerá de la variable n, el **peor caso posible es que** sea un número muy grande y a su vez no sea un cuadrado perfecto.

Es un **algoritmo recursivo**. Por tanto, llamemos T(n) al tiempo de ejecución que tarda el algoritmo MaximoMinimoDyV en resolver un problema de tamaño n. Viendo nuestro algoritmo tendremos dos casos (uno general y uno base):

- Caso base: Tenemos un conjunto de sentencias condicionales, cuyo valor de eficiencia es de O(1).

- Caso general: Se da cuando valor < n. En este caso:
  - El algoritmo sitúa el centro a la izquierda del centro anterior, ejecutando de nuevo la función recursiva con la mitad de datos.
  - Luego hace una **llamada recursiva**, con el valor de izquierda y derecha actualizado en cada iteración de la llamada recursiva, hasta encontrar el valor que originaría el cuadrado perfecto o que devuelva false porque no lo encuentre.

  Por tanto, podemos aproximar T(n) en el caso general como **T(n)= T(n/2)+1**. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de MaximoMinimoDyV.

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, 2^m , quedando:

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

Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

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

Resolvemos la "**parte homogénea**":

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

Y obtenemos la parte homogénea del polinomio característico como PH(X) = (X-1)

Para resolver la "**parte no homogénea**", debemos conseguir un escalar "b1 " y un polinomio "q1(m)". De modo que:

$$1 = b1 \cdot q1(m)$$

Con lo que deducimos que b1 = 1 y q1(m) tendrá un grado d1 = 0.
El **polinomio característico** se obtiene como: P(X)=PH(X)=(X-1)(X-1)= (X-1)(X-1)

De este modo, tenemos r=2 dos raíces iguales, con valor 1; y multiplicidades M = 2 en ambos casos.

Aplicando la fórmula de la **ecuación característica**, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

$$T(2^m) = \sum_{i=1}^{r} \sum_{j=0}^{M-1} C_{ij} Ri^m m^j = c_{10} 1^m + c_{21} 1^m m$$

Ahora **deshacemos el cambio de variable** para volver al espacio de tiempos inicial:

$$T(n) = c_{10} + c_{21} \log(n)$$

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(n))**.

4. Implementación de los métodos y ejecuciones:ç

En este caso hemos utilizado un bucle para comprobar tanto el método básico o por fuerza bruta como el divide y vencerás con distintos valores y ejecuciones de forma que entremos en todos los caso posibles (casos base y casos generales) de ambos algoritmos, para ello recorro desde 0 a 10 en ambos casos y uso el método para los cuadrados del contador.

```
int main() {
    cout << "Ejecución para algoritmo básico: " << endl;
    for (int i = 0; i < 10; i++) {
        int valor = pow(i, 2);
        if (CuadradoPerfecto(valor)) cout << valor << ": es cuadrado perfecto" << endl;
        valor++;
        if (!CuadradoPerfecto(valor)) cout << valor << ": NO es cuadrado perfecto" << endl;
    }
    cout << endl;
    cout << "Ejecución para algoritmo DyV: " << endl;
    for (int i = 0; i < 10; i++) {
        int valor = pow(i, 2);
        if (CuadradoPerfectoDyV(valor, 0, valor)) cout << valor << ": es cuadrado perfecto" << endl;
        valor++;
        if (!CuadradoPerfectoDyV(valor, 0, valor)) cout << valor << ": NO es cuadrado perfecto" << endl;
    }
}
```

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Documentos/Escritorio/Practica_2_ALG_DyV/ejercicios$ ./Ejercicio2
Ejecución para algoritmo básico:
0: es cuadrado perfecto
1: es cuadrado perfecto
2: NO es cuadrado perfecto
4: es cuadrado perfecto
5: NO es cuadrado perfecto
9: es cuadrado perfecto
10: NO es cuadrado perfecto
16: es cuadrado perfecto
17: NO es cuadrado perfecto
25: es cuadrado perfecto
26: NO es cuadrado perfecto
36: es cuadrado perfecto
37: NO es cuadrado perfecto
49: es cuadrado perfecto
50: NO es cuadrado perfecto
64: es cuadrado perfecto
65: NO es cuadrado perfecto
81: es cuadrado perfecto
82: NO es cuadrado perfecto

Ejecución para algoritmo DyV:
0: es cuadrado perfecto
1: es cuadrado perfecto
2: NO es cuadrado perfecto
4: es cuadrado perfecto
5: NO es cuadrado perfecto
9: es cuadrado perfecto
10: NO es cuadrado perfecto
16: es cuadrado perfecto
17: NO es cuadrado perfecto
25: es cuadrado perfecto
26: NO es cuadrado perfecto
36: es cuadrado perfecto
37: NO es cuadrado perfecto
49: es cuadrado perfecto
50: NO es cuadrado perfecto
64: es cuadrado perfecto
65: NO es cuadrado perfecto
81: es cuadrado perfecto
82: NO es cuadrado perfecto
```

Algoritmo 3:

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.

¿Qué método básico/por fuerza bruta podemos diseñar para calcular, dado un valor de n de entrada, si existe el número y que hace cumplir la igualdad y cuál es su valor? ¿Es posible diseñar un algoritmo Divide y Vencerás para igualar (o mejorar) la eficiencia de este método básico?

1. Diseño del algoritmo básico:

```
bool existe(int n) {
    if (n <= 5) return false;
    bool encontrado = false;
    int valor = 0;
    for(int i = 0; valor <= n; i++) {
        valor = i*(i+1)*(i+2);
        if (valor == n) encontrado = true;
    }
    return encontrado;
}
```

El algoritmo básico de la función existe se encarga de hacer pivotar un valor desde el 0 hasta que el producto de ese valor por sus dos consecutivos sea igual o superior a n.
De esta forma evitaríamos tener que recorrer desde 0 hasta n con el bucle, ya que una vez el producto de los tres consecutivos superen, todos los demás serán superiores y, por tanto, distintos.

2. Análisis y diseño DyV:

```
bool existeDyV(const int n, int izquierda, int derecha) {

    if (n <= 5) return false;

    int centro = (izquierda + derecha) / 2;
    int valor = centro*(centro+1)*(centro+2);

    if (valor == n) return true;

    if (izquierda == derecha || izquierda > derecha) return false;
    if (valor > n)
        derecha = centro-1;
    else
        izquierda = centro+1;

    return existeDyV(n, izquierda, derecha);
}
```

3. Análisis de Eficiencia:

ALGORITMO BÁSICO:

Teniendo en cuenta que el **tamaño del problema** dependerá de la variable n, el **peor caso posible** es que n = infinito, ya que, tendría una eficiencia O(n).

Es un **algoritmo básico,** que se compone de un bucle for con eficiencia O(n) y un condicional con eficiencia O(1).

Finalmente, con la regla del máximo, obtenemos que la eficiencia teórica de dicho algoritmo es de O(n).

DIVIDE Y VENCERÁS:

Teniendo en cuenta que el **tamaño del problema** dependerá de la variable n, el **peor caso posible es que** sea un número muy grande y a su vez no sea producto de tres consecutivos.

Es un **algoritmo recursivo**. Por tanto, llamemos T(n) al tiempo de ejecución que tarda el algoritmo MaximoMinimoDyV en resolver un problema de tamaño n. Viendo nuestro algoritmo tendremos dos casos (uno general y uno base):

- Caso base: Tenemos un conjunto de sentencias condicionales, cuyo valor de eficiencia es de O(1).

- Caso general: Se da cuando valor $<$ n. En este caso:
  - El algoritmo sitúa el centro a la izquierda del centro anterior, ejecutando de nuevo la función recursiva con la mitad de datos.
  - Luego hace una **llamada recursiva**, con el valor de izquierda y derecha actualizado en cada iteración de la llamada recursiva, hasta encontrar el valor que originaría producto de tres números consecutivos o que devuelva false porque no lo encuentre.

  Por tanto, podemos aproximar T(n) en el caso general como **T(n)= T(n/2)+1**. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de MaximoMinimoDyV.

  Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, 2^m , quedando:

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

  Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

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

Resolvemos la "**parte homogénea**":

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

Y obtenemos la parte homogénea del polinomio característico como PH(X) = (X-1)

Para resolver la "**parte no homogénea**", debemos conseguir un escalar "b1 " y un polinomio "q1(m)". De modo que:

$$1 = b1\cdot q1(m)$$

Con lo que deducimos que b1 = 1 y q1(m) tendrá un grado d1 = 0.
El **polinomio característico** se obtiene como: P(X)=PH(X)=(X-1)(X-1)= (X-1)(X-1)

De este modo, tenemos r=2 dos raíces iguales, con valor 1; y multiplicidades M = 2 en ambos casos.

Aplicando la fórmula de la **ecuación característica**, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

$$T(2^m) = \sum_{i=1}^{r}\sum_{j=0}^{M-1} C_{ij} Ri^m m^j = c_{10}\,1^m + c_{21}1^m m$$

Ahora **deshacemos el cambio de variable** para volver al espacio de tiempos inicial:

$$T(n) = c_{10} + c_{21}\log(n)$$

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(n))**.

4. Implementación de los métodos y ejecuciones:

En el desarrollo de la práctica nos dimos cuenta de algo bastante curioso y es que el producto de 3 números consecutivos resultan ser siempre múltiplos de 6, con lo que recorrimos un bucle de 0 hasta 211, con un aumento del contador de 6 en cada iteración, para tratar de esta forma de ejecutar el máximo número de casos en los que se encuentra un número que se forma con 3 consecutivos, tanto en el algoritmo básico como en el divide y vencerás.

```cpp
int main() {
    cout << "Ejecución para algoritmo básico: " << endl;
    for (int i = 0; i < 211; i+=6) {
        if (existe(i)) cout << i << ": tiene 3 consecutivos que lo forman" << endl;
        if (!existe(i)) cout << i << ": NO tiene 3 consecutivos que lo forman" << endl;
    }

    cout << endl << "Ejecución para algoritmo DyV: " << endl;
    for (int i = 0; i < 211; i+=6) {
        if (existeDyV(i,0, i)) cout << i << ": tiene 3 consecutivos que lo forman" << endl;
        if (!existeDyV(i, 0, i)) cout << i << ": NO tiene 3 consecutivos que lo forman" << endl;
    }
}
```

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Documentos/Escritorio/Practica_2_ALG_DyV/ejercicios$ ./Ejercicio3
Ejecución para algoritmo básico:
0: NO tiene 3 consecutivos que lo forman
6: tiene 3 consecutivos que lo forman
12: NO tiene 3 consecutivos que lo forman
18: NO tiene 3 consecutivos que lo forman
24: tiene 3 consecutivos que lo forman
30: NO tiene 3 consecutivos que lo forman
36: NO tiene 3 consecutivos que lo forman
42: NO tiene 3 consecutivos que lo forman
48: NO tiene 3 consecutivos que lo forman
54: NO tiene 3 consecutivos que lo forman
60: tiene 3 consecutivos que lo forman
66: NO tiene 3 consecutivos que lo forman
72: NO tiene 3 consecutivos que lo forman
78: NO tiene 3 consecutivos que lo forman
84: NO tiene 3 consecutivos que lo forman
90: NO tiene 3 consecutivos que lo forman
96: NO tiene 3 consecutivos que lo forman
102: NO tiene 3 consecutivos que lo forman
108: NO tiene 3 consecutivos que lo forman
114: NO tiene 3 consecutivos que lo forman
120: tiene 3 consecutivos que lo forman
126: NO tiene 3 consecutivos que lo forman
132: NO tiene 3 consecutivos que lo forman
138: NO tiene 3 consecutivos que lo forman
144: NO tiene 3 consecutivos que lo forman
150: NO tiene 3 consecutivos que lo forman
156: NO tiene 3 consecutivos que lo forman
162: NO tiene 3 consecutivos que lo forman
168: NO tiene 3 consecutivos que lo forman
174: NO tiene 3 consecutivos que lo forman
180: NO tiene 3 consecutivos que lo forman
186: NO tiene 3 consecutivos que lo forman
192: NO tiene 3 consecutivos que lo forman
198: NO tiene 3 consecutivos que lo forman
204: NO tiene 3 consecutivos que lo forman
210: tiene 3 consecutivos que lo forman
```

```
Ejecución para algoritmo DyV:
0: NO tiene 3 consecutivos que lo forman
6: tiene 3 consecutivos que lo forman
12: NO tiene 3 consecutivos que lo forman
18: NO tiene 3 consecutivos que lo forman
24: tiene 3 consecutivos que lo forman
30: NO tiene 3 consecutivos que lo forman
36: NO tiene 3 consecutivos que lo forman
42: NO tiene 3 consecutivos que lo forman
48: NO tiene 3 consecutivos que lo forman
54: NO tiene 3 consecutivos que lo forman
60: tiene 3 consecutivos que lo forman
66: NO tiene 3 consecutivos que lo forman
72: NO tiene 3 consecutivos que lo forman
78: NO tiene 3 consecutivos que lo forman
84: NO tiene 3 consecutivos que lo forman
90: NO tiene 3 consecutivos que lo forman
96: NO tiene 3 consecutivos que lo forman
102: NO tiene 3 consecutivos que lo forman
108: NO tiene 3 consecutivos que lo forman
114: NO tiene 3 consecutivos que lo forman
120: tiene 3 consecutivos que lo forman
126: NO tiene 3 consecutivos que lo forman
132: NO tiene 3 consecutivos que lo forman
138: NO tiene 3 consecutivos que lo forman
144: NO tiene 3 consecutivos que lo forman
150: NO tiene 3 consecutivos que lo forman
156: NO tiene 3 consecutivos que lo forman
162: NO tiene 3 consecutivos que lo forman
168: NO tiene 3 consecutivos que lo forman
174: NO tiene 3 consecutivos que lo forman
180: NO tiene 3 consecutivos que lo forman
186: NO tiene 3 consecutivos que lo forman
192: NO tiene 3 consecutivos que lo forman
198: NO tiene 3 consecutivos que lo forman
204: NO tiene 3 consecutivos que lo forman
210: tiene 3 consecutivos que lo forman
```