



Instituto Politécnico Nacional

Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2021-1, 3CV1, Práctica 1,
25-10-2020

Práctica 2: Funciones recursivas vs iterativas.

De la Cruz Sierra Diana Paola

Raya Chávez Samuel Antonio

dianapaodcs@gmail.com, samiraya1323@gmail.com

Resumen. Se presentan diferentes algoritmos (cálculo de un producto y cálculo de un cociente entre dos números positivos) donde todos ellos llegan al mismo resultado, y se hará un análisis para identificar cuál es el mejor algoritmo.

Palabras clave. Producto, Cociente, A-priori, A-posteriori, C++

1. Introducción

La recursividad es un concepto fundamental en matemáticas y en computación. Es una alternativa diferente para implementar estructuras de repetición (ciclos). Los módulos se hacen llamadas recursivas.

Definir una función recursiva consiste en:

- caso base:
una solución simple para un caso particular (puede haber más de un caso base).
- caso general o recursivo:
una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al caso base

Para la verificación de funciones y procedimientos recursivos hay un método llamado: el método de las tres preguntas que consta en:

- La pregunta Caso-Base: ¿Existe una salida no recursiva o caso base del subalgoritmo? Además, ¿el subalgoritmo funciona correctamente para ella?
- La pregunta Más-pequeño: ¿Cada llamada recursiva se refiere a un caso más pequeño del problema original?
- La pregunta Caso-General: ¿es correcta la solución en aquellos casos no base?

Cuando un procedimiento incluye una llamada a sí mismo se conoce como recursión directa. Y cuando un procedimiento llama a otro procedimiento y éste causa que el procedimiento original sea invocado, se conoce como recursión indirecta.

La mayoría de los algoritmos pueden expresarse tanto de forma iterativa como de forma recursiva. En general la versión iterativa de una función siempre es más eficiente que la versión recursiva, tanto en términos de velocidad de ejecución como de memoria utilizada. En la versión recursiva, la llamada a la función requiere del uso de la pila para almacenar los parámetros pasados (si los hay), dirección de retorno, resultado, etc. Si el nivel de recursión puede ser muy elevado, el uso de memoria puede ser considerable o incluso prohibitivo.

Razones para escribir programas recursivos:

- Son mas cercanos a la descripción matemática.

- Se adaptan mejor a las estructuras de datos recursivas.
- Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.
- a veces son más fáciles de entender e implementar

2. Conceptos básicos

De aquí en adelante, y para estos algoritmos, se estudiarán 2 posibles casos: el mejor y el peor caso.

- **Peor caso.** Corresponde a la traza del algoritmo que realiza más instrucciones.
- **Mejor caso.** Corresponde a la traza (secuencia de sentencias) del algoritmo que realiza menos instrucciones.

A la hora de medir el tiempo, siempre se hará en función del número de operaciones elementales que realiza dicho algoritmo, entendiendo por operaciones elementales aquellas que el ordenador realiza en un tiempo acotado por una constante.

En general, es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, aquellas que caracterizan que el algoritmo sea lento o rápido en el sentido que nos interesa (Universidad de Málaga, 2009)

A continuación se muestran los 3 pseudo-códigos de algoritmos para calcular el producto de dos números.

Algorithm 1 Pseudocódigo: prod1

```

1: procedure PROD1(int  $m$ , int  $n$ )
2:    $r \leftarrow 0$ 
3:   while  $n \neq 0$  do
4:      $r \leftarrow r + m$ 
5:      $n \leftarrow n - 1$ 
6:   end while
7:   return  $r$ 
8: end procedure

```

Algorithm 2 Pseudocódigo: prod2

```
1: procedure PROD2(int  $m, int n$ )
2:    $n \leftarrow 0$ 
3:   while  $n \neq 0$  do
4:     if  $n \leq 1$  then
5:        $r \leftarrow r + m$ 
6:     end if
7:      $m \leftarrow 2 * m$ 
8:      $n \leftarrow n / 2$ 
9:   end while
10:  return  $r$ 
11: end procedure
```

Algorithm 3 Pseudocódigo: prod3

```
1: procedure PROD3(int  $a, int b$ )
2:   if  $b == 1$  then
3:     return  $a$ 
4:   else
5:     return  $a + prod3(a, b - 1)$ 
6:   end if
7: end procedure
```

Ahora se presentarán 3 diferentes pseudo-códigos para obtener el cociente de una división.

Algorithm 4 Pseudocódigo: Obtención de un cociente 1

```
1: procedure DIV1( $n, div, *r$ )
2:    $q \leftarrow 0$ 
3:   while  $n \geq div$  do
4:      $n \leftarrow n - div$ 
5:      $q++$ 
6:   end while
7:    $*r \leftarrow n$ 
8:   return  $q$ 
9: end procedure
```

Algorithm 5 Pseudocódigo: Obtención de un cociente 2

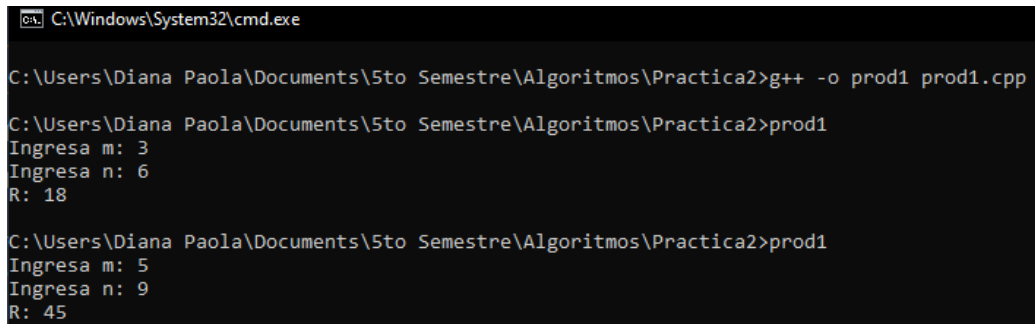
```
1: procedure DIV2( $n, div, *r$ )
2:    $dd \leftarrow div$ 
3:    $q \leftarrow 0$ 
4:    $*r \leftarrow n$ 
5:   while  $dd \leq n$  do
6:      $dd \leftarrow 2dd$ 
7:   end while
8:   while  $dd > div$  do
9:      $dd \leftarrow dd/2$ 
10:     $q \leftarrow 2 * q$ 
11:    if  $dd \leq *r$  then
12:       $*r \leftarrow *r - dd$ 
13:       $q \leftarrow q + 1$ 
14:    end if
15:  end while
16:  return  $q$ 
17: end procedure
```

Algorithm 6 Pseudocódigo: Obtención de un cociente 3

```
1: procedure DIV3( $n, div, *r$ )
2:   if  $div > n$  then
3:     return 0
4:   else
5:      $*r \leftarrow n - div$ 
6:   end if
7: end procedure
```

3. Experimentación y resultados

3.1. Algoritmo prod1



```
C:\Windows\System32\cmd.exe

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>g++ -o prod1 prod1.cpp

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>prod1
Ingresa m: 3
Ingresa n: 6
R: 18

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>prod1
Ingresa m: 5
Ingresa n: 9
R: 45
```

Figura 1: Pruebas de 2 ejemplos en ejecución del algoritmo1

En la figura 1 se observan dos ejemplos del producto de dos números utilizando el algoritmo 1.

Para este algoritmo el mejor caso es cuando $n \leq m$, ya que se ejecutará menos cantidad de veces que si fuera al revés.

Haciendo un análisis del peor caso se obtiene que el número de pasos ejecutados es: 1 para la asignación de n , el for se divide en inicialización, condición que se ejecutan 1, $n+1$ y n veces respectivamente, el cuerpo del while se ejecuta $2*(n-1)$ veces y el último return 1 vez. Obteniendo el siguiente resultado: $T(n) = 1 + (n+1) + n + 2*(n-1) + 1 = 4n + 1$

n	t(n) = 4n+1
3	13
7	29
15	61
22	89
30	121

Cuadro 1: Análisis A posteriori del peor caso

Graficando los puntos en la tabla 1 y analizando los puntos, podría proponerse que una gráfica que se ajusta a estos puntos por encima es $T(n) = 4n$ como puede observarse en la figura 2 y 3 respectivamente.

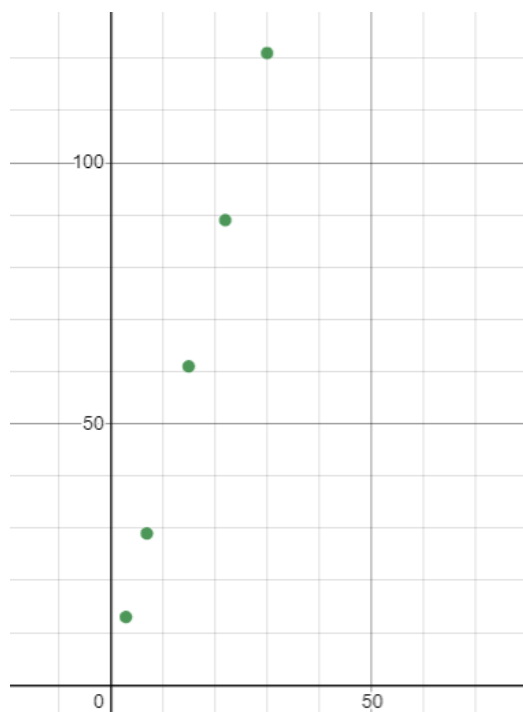


Figura 2: Puntos de n vs $T(n)$

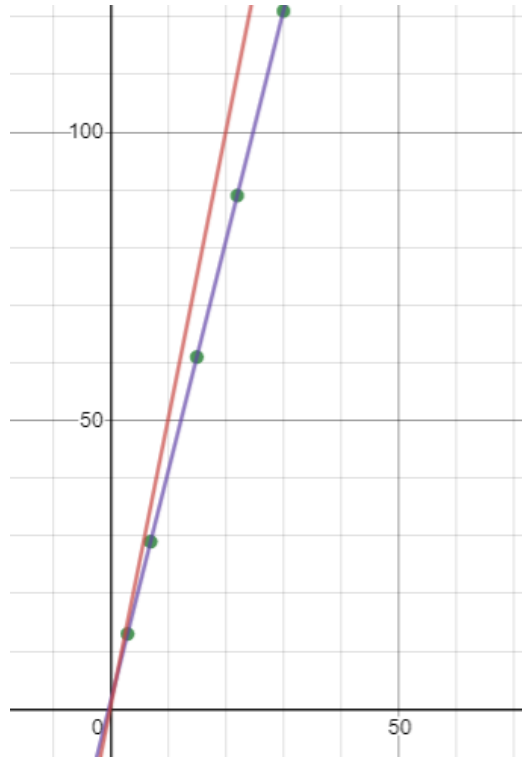


Figura 3: ecuación propuesta $T(n)=5n$ y $T(n)=4n+1$

A continuación se muestra un análisis A-priori del algoritmo 1:

	costo	# pasos ejecutados
$r=0$	$c1$	1
$\text{while}(n>0)$	$c2$	n
$r = r+m;$	C	$n-1$
$n-1$		
$\text{return } r$	$c4$	1

Cuadro 2: Tiempo computacional 1

Análizando la tabla 2 se observa que:

$$T(n) = C1 + C2 * n + C(n - 1) + C4 = n(C2 + C) + (C1 - C3 + C5) = an + b$$

Viendo el análisis A priori y A posteriori se observa que este algoritmo tiene un orden de complejidad: $T(n) \in O(n)$

3.2. Algoritmo prod2

```
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>g++ -o prod2 prod2.cpp
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>prod2
Ingresa m: 4
Ingresa n: 6
R: 24

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>prod2
Ingresa m: 7
Ingresa n: 2
R: 14
```

Figura 4: Pruebas de 2 ejemplos en ejecución del algoritmo2

En la figura 4 se observan dos ejemplos del producto de dos números utilizando el algoritmo 2

El mejor de los casos, al igual que en el primer algoritmo, se da cuando $n < m$, ya que se ejecutaría una menor cantidad de veces.

Haciendo un análisis del peor caso, este se da cuando n es un número en potencia de 2 menos 1. Por ejemplo: $2^{10} - 1 = 1023$ esto debido a que este número tiene todos sus bits prendidos y el if entra la cantidad de bits prendidos que tenga n . Así se ejecuta 1 vez la declaración del r , el while se ejecuta $2^{\lfloor \log_2(n) \rfloor}$ y el return una vez, siendo un aproximado:

$$T(n) = 2 + \log_2(n)$$

En la figura 5 se muestran los puntos n vs $t(n)$ del peor caso que se tabulan en la tabla 3

n	$f(n)$
1	2
5	4
10	5
16	6
25	6
32	7

Cuadro 3: puntos algoritmo 2

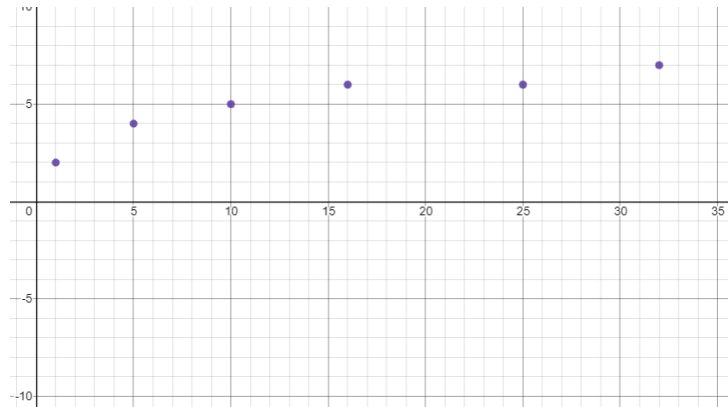


Figura 5: Puntos de n vs T(n)

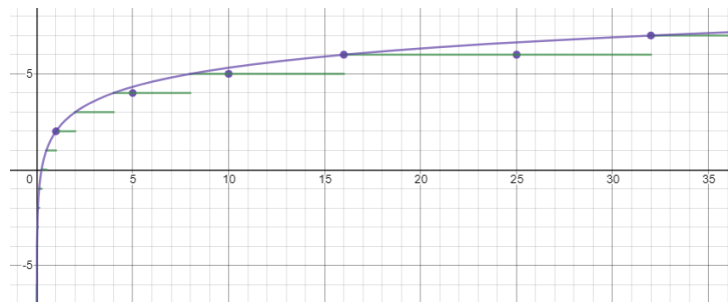


Figura 6: Ecuación propuesta y T(n)

como se puede observar en la imagen 6 podría proponerse $T(n) = 2 + \log_2(n)$ ya que se ajusta por arriba a los puntos graficados.

Haciendo un análisis A priori se observa lo siguiente: Las líneas fuera del while son constantes, mientras que el while se ejecuta mientras $n > 0$, analizando mejor a detalle se observa que se ejecuta $2^{\lfloor \log_2(n) \rfloor}$, lo cual implica que eso tiene complejidad logarítmica.

int prod2(int m, int n)	Costo	No. pasos ejecutados
r = 0	c1	1
while n>0	C	$\log_2(n)$
if(n&1)		
r = r+m		
m = 2*m		
n = n/2		
return r	c2	1

Cuadro 4: Analisis A priori del algoritmo 2

Observando la tabla 4 que es un análisis a priori, que si lo analizamos por bloques domina la complejidad logaritmica y contemplando el análisis a posteriori se observa que el orden de complejidad de $prod2 \in O(\log_2(n))$

3.3. Algoritmo prod3

```
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>g++ -o prod3 prod3.cpp
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>prod3
Ingresa a: 2
Ingresa b: 22
R: 44
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica2>prod3
Ingresa a: 9
Ingresa b: 6
R: 54
```

Figura 7: Pruebas de 2 ejemplos en ejecución del algoritmo3

En la figura 7 se observan dos ejemplos del producto de dos números utilizando el algoritmo 3

Para calcular el orden de complejidad se utilizara el método de sustitución hacia atras. Comenzamos definiendo la función recursiva que dice:

$M(b) = [a \text{ si } b=1, M(b-1)+1 \text{ si } b>1]$

$$\begin{aligned}
 M(b) &= M(a,b-1)+1 \\
 &= [M(b-2)+1]+1 &= M(b-2)+2 \\
 &= [M(b-3)+1]+2 &= M(b-3)+3 \\
 &\vdots \\
 &: \\
 M(i) &= M(b-i)+i \\
 M(b-1) &= M(b-b+1)+(b-1) &= M(1)+(b-1) \\
 &= a + (b-1) &= b + (a-1)
 \end{aligned}$$

Cuadro 5: sustitución hacia atras

En la tabla 5 puede observarlos los pasos para ver que la complejidad es lineal, es decir $M(b) \in O(n)$ mostrandose su la grafica de la complejidad en la imagen 8

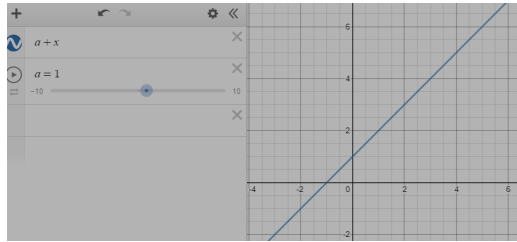


Figura 8: Complejidad de prod3

Analizando los algoritmos 1, 2 y 3 se concluye que 1 y 3 tienen ambas complejidad lineal, mientras que 2 tiene complejidad algorítmica, por lo que el algoritmo 2 es más eficiente mientras que el 1 y 2 al tener la misma complejidad es indistinto.

3.4. Algoritmo div1

```

C:\Users\samir\Desktop\div1.exe
Ingrese n
10
Ingrese div
2
El cociente es: 5
-----
Process exited after 3.234 seconds with return value 0
Presione una tecla para continuar . . .

C:\Users\samir\Desktop\div1.exe
12
El cociente es: 7
-----
Process exited after 3.117 seconds with return value 0
Presione una tecla para continuar . . .

```

Figura 9: Pruebas experimentales del algoritmo 1: div1

En la figura 9 se observan algunos ejemplos de la división de dos números utilizando el algoritmo 4.

En este algoritmo el mejor caso será si n es menor que div , omitiendo así lo que está adentro del ciclo *While*.

Mientras que el peor caso ocurre cuando $div=1$ y que n sea mucho mayor a div , haciendo que el ciclo *While* se ejecute muchas veces.

Haciendo un análisis del peor caso se obtiene que el número de pasos ejecutados es:

- > 1 para la inicialización de la variable q .
- $> n+1$ para el ciclo *While*.
- $> 2n$ para todo el cuerpo del ciclo *While*.
- ¡Y 2 más por la penúltima condición y el return.

Obteniendo el siguiente resultado: $T(n) = 1 + (n+1) + 2n + 2 = 3n + 4$

Y con algunos datos arbitrarios se obtiene la siguiente tabla.

n	$T(n) = 3n+4$
22	70
54	166
87	265
123	373
158	477

Cuadro 6: Análisis A posteriori del peor caso

Viendo la gráfica que se forma al unir los puntos en la tabla 6, podría proponerse que una gráfica que se ajusta a estos puntos por encima que es $T(n) = 4n$ como puede observarse en la figura 10.

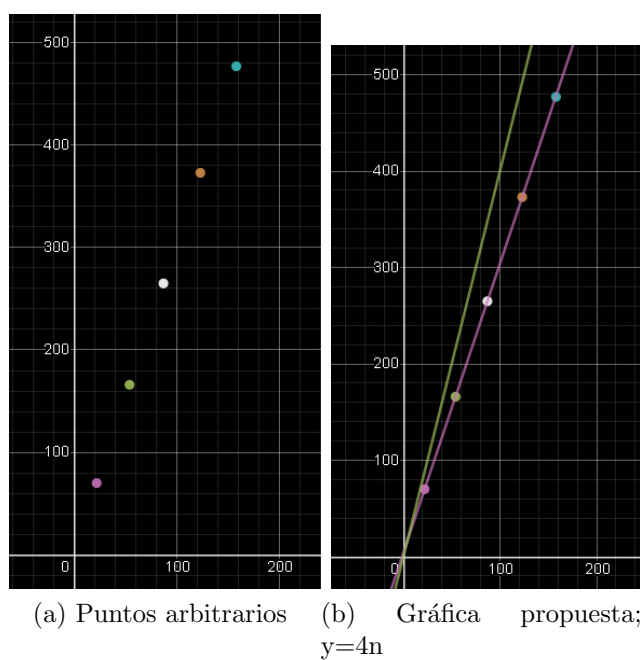


Figura 10: Gráficas de div1

Ahora se muestra un análisis A-priori del algoritmo 4:

	costo	# pasos ejecutados
q=0	c1	1
while(n ≥ div)	c2	n+1
n = n-div;	c3	n
q++	c4	
*r = n	c5	1
return q	c6	1

Cuadro 7: Tiempo computacional del algoritmo 4

Viendo la tabla 7 se observa que:

$$T(n) = c1 + c2 * (n + 1) + (c3 + c4)(n) + c5 + c6 = n(c2 + c3 + c4) + (c1 + c2 + c5 + c6) = an + b$$

Viendo el análisis A priori y A posteriori se observa que este algoritmo tiene un orden de complejidad: $T(n) \in O(n)$

3.5. Algoritmo div2

```

C:\Users\samir\Desktop\div2.exe
Ingrese n
80
Ingrese div
9
El cociente es: 8
-----
Process exited after 6.594 seconds with return value 8
Presione una tecla para continuar . . .

C:\Users\samir\Desktop\div2.exe
Ingrese n
135
Ingrese div
62
El cociente es: 2
-----
Process exited after 11.14 seconds with return value 2
Presione una tecla para continuar . . .

```

Figura 11: Pruebas experimentales del algoritmo 2: div2

En la figura 11 se observan algunos ejemplos de la división de dos números utilizando el algoritmo 5.

En este algoritmo el mejor caso será si div es mayor que n, de esta forma se omite ambos ciclos *While*.

Mientras que el peor caso ocurre div es igual a 1 y n es mucho mayor a div, haciendo que los ciclos *While* y la condicional *If* se ejecuten varias veces.

Haciendo un análisis del peor caso se obtiene que el número de pasos ejecutados es:

- + 3 para las inicializaciones.
- + $\log_2(n)$ para el primer ciclo *While*.

+ $\log_2(n)$ para el segundo ciclo *While*.

+ Y 1 más por el return.

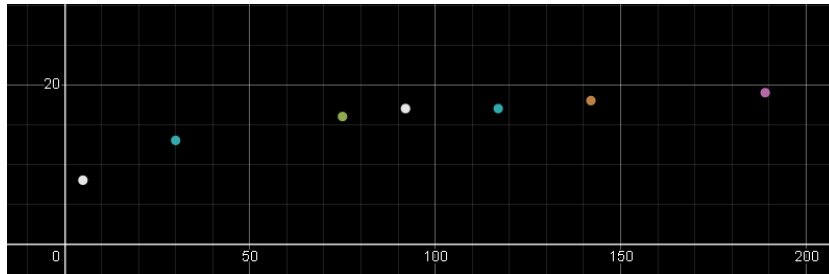
Obteniendo el siguiente resultado: $T(n) = 4 + 2\log_2(n)$

Y con algunos datos arbitrarios se obtiene la siguiente tabla.

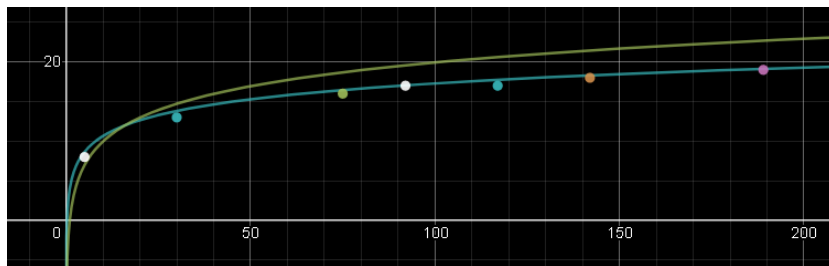
n	$T(n) = 4 + 2\lfloor \log_2(n) \rfloor$
5	8
30	13
75	16
92	17
117	17
142	18
189	19

Cuadro 8: Análisis A posteriori del peor caso

Viendo la gráfica que se forma al unir los puntos en la tabla 8, podría proponerse una gráfica que se ajusta a estos puntos por encima que es $T(n) = 3\lfloor \log_2(n) \rfloor$ como puede observarse en la figura 12.



(a) Puntos arbitrarios



(b) Gráfica propuesta; $T(n) = 3\lfloor \log_2(n) \rfloor$

Figura 12: Gráficas de \log_2

Ahora se muestra un análisis A-priori del algoritmo 5:

	costo	# pasos ejecutados
int dd=div	c1	1
int q=0	c2	1
*r=n	c3	1
while(dd ≤ n) n = n-div;	c4 c5	$\log_2(n)$
while dd<div dd=dd/2 q = 2*q if dd ≤ *r *r=*r-dd q++	c6 c7 c8 c9 c10 c11	$\log_2(n)$
return q	c12	1

Cuadro 9: Tiempo computacional del algoritmo 5

Viendo la tabla 9 se observa que:

$$T(n) = (c1 + c2 + c3) + (c4 + c5) * (\log_2(n)) + (c6 + c7 + c8 + c9 + c10 + c11) * (\log_2(n)) + c12 = \log_2(n) * (c4 + c5 + c6 + c7 + c8 + c9 + c10 + c11) + (c1 + c2 + c3 + c12) = a * \log_2(n) + b$$

Viendo el análisis A priori y A posteriori se observa que este algoritmo tiene un orden de complejidad: $T(n) \in o(\log_2(n))$

3.6. Algoritmo div3

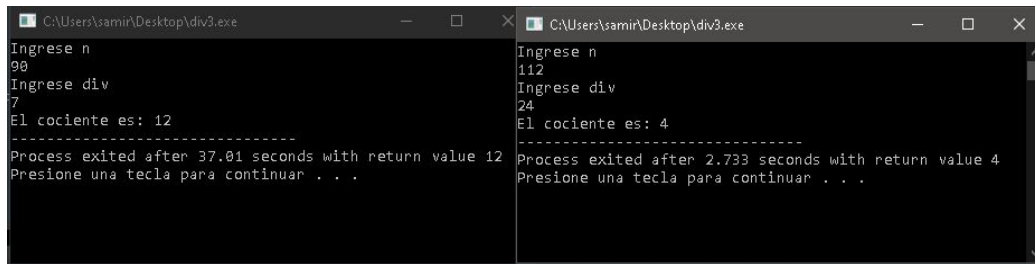


Figura 13: Pruebas experimentales del algoritmo 3: div3

En la figura 13 se observan algunos ejemplos de la división de dos números utilizando el algoritmo 6.

Analizando los algoritmos 4, 5 y 6 se observa que el algoritmo 4 y 6 tienen ambas complejidad lineal, mientras que el algoritmo 5 tiene complejidad logarítmica, por ello, es que el algoritmo 4 y 6 al tener la misma complejidad son indistintos, mientras que el algoritmo 5 resulta ser el más eficiente. Gráficamente se puede ver de la siguiente forma.

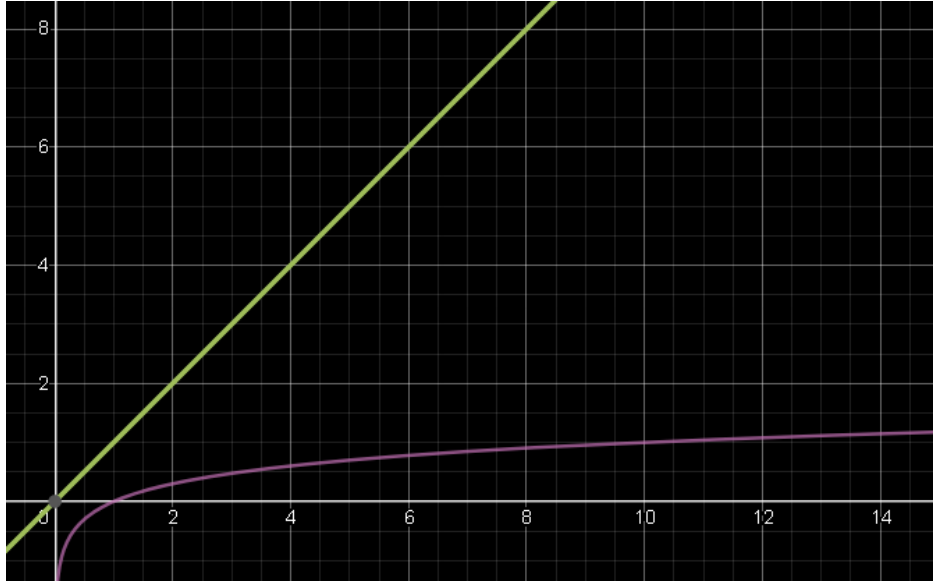


Figura 15: $\text{div1} \in O(n)$
 $\text{div2} \in O(\log(n))$
 $\text{div3} \in O(n)$

4. Conclusiones

- Conclusiones generales:

Hay ciertos aspectos que se podrían considerar en los algoritmos de producto, por ejemplo que sería mejor que $n < m$, ya que se ejecutaría menos veces, pero eso es independiente de lo que hace la función principal donde ya se ha establecido un valor de n , independientemente si es mayor, menor o igual a m .

Tomando a las divisiones, su peor caso en todos los algoritmos es si $\text{div}=1$, ya que el algoritmo se repetirá n veces, y peor aún si n resulta ser un número muy grande.

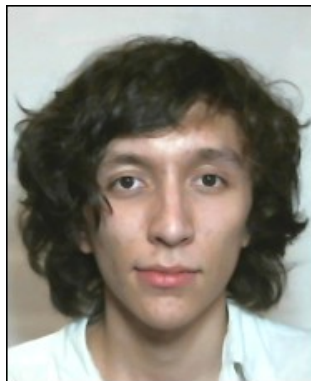
- Diana Paola De la Cruz Sierra:

Para decidir si usar un algoritmo recursivo o lineal depende del algoritmo programado, ya que a pesar de ser más sencillo de implementar una opción recursiva muchas veces es más lento en tiempo, además de ocupar un mayor espacio en memoria por la pila auxiliar utilizada.



- Raya Chávez Samuel Antonio:

Es curioso, uno creería que el algoritmo recursivo sería el más rápido al tener una menor cantidad de líneas de código, pero siendo que el segundo algoritmo pertenece a los logarítmicos, este va a ser el más rápido para los tres casos. Es por ello que los análisis a priori realizados nos ayudan a comprender cuál será el algoritmo más eficiente mientras que los análisis a posteriori nos ayuda a visualizar esto ya mencionado.



4.1. Fuentes de información:

- Castillo, O. (S.F). *Recursividad*. [PDF]. Disponible en:
<https://www.uv.mx/personal/ocastillo/files/2011/04/Recursividad.pdf>

- Alvarez, P (S.F). *Tema 1: recursividad*. [PDF]. Disponible en:
<http://www.lcc.uma.es/alvarezp/pm/recursividad.pdf>
- desarrollando.net (S.F). *Unidad 3: algoritmos recursivos*. [PDF]. Disponible en:
<http://formacion.desarrollando.net/cursosfiles/formacion/curso-454/deda-03.pdf>
- Universidad de Málaga. (2009). *LA COMPLEJIDAD DE LOS ALGORITMOS*. Octubre 27, 2020, de Universidad de Málaga Sitio web:
<http://www.lcc.uma.es/av/Libro/CAP1.pdf>