

PRACTICA 2: FUNCIONES RECURSIVAS VS ITERATIVAS.

Mendoza Martínez Eduardo, Aguilar Gonzalez Daniel.

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
edoomm8@gmail.com, daquilarglz97@gmail.com

Resumen: La siguiente practica pretende mostrar el comportamiento de dos algoritmos (Calculo del Producto y Cociente de dos numeros) implementados mediante funciones recursivas e iterativas y comparando sus tiempos de ejecucion, asi como la complejidad de estos.

Palabras Clave: Complejidad algoritmica, funciones iterativas, funciones recursivas.

1 Introducción

Anteriormente ya habiamos hablado de la existencia de diversos tipos de algoritmos, y sus complejidades y como estos factores influyen en la eficiencia de estos. En esta practica, nos encargaremos de mostrar las diferencias (desde numero de lineas de codigo, pasando por tiempo invertido en la ejecucion, hasta llegar a la complejidad algoritmica) de dos algoritmos: Producto y Cociente de dos numeros, implementandolos mediante funciones recursivas e iterativas, para visualizar las diferencias de costo computacional que requiere cada uno en cada metodo, y la complejidad que conllevan.

2 Conceptos Básicos

Se requiere conocer algunos conceptos relacionados a la implementacion de los algoritmos estudiados en esta practica y los metodos a traves de los cuales se desarrollaron. Se presenta a continuacion la informacion correspondiente.

2.1 Funcion recursiva.

Una funcion recursiva es aquella que se llama asi misma para resolverse. Dicho de otra manera, una funcion recursiva se resuelve con una llamada asi misma,

cambiando el valor de un parametro en la llamada a la funcion. A traves de las sucesivas llamadas recursivas a la funcion se van obteniendo valores que, sirven para obtener el valor de la funcion llamada originalmente. El proceso de llamadas recursivas siempre tiene que acabar en una llamada a la funcion que se resuelve de manera directa, sin necesidad de invocar de nuevo la funcion. Esto sera siempre necesario, para que llegue un momento que se corten las llamadas reiterativas a la funcion y no se entre en un bucle infinito de invocaciones.

2.2 Funcion iterativa

Iteración es un vocablo que tiene su origen en el término latino *iteratio*. Se trata de una palabra que describe el acto y consecuencia de iterar, un verbo que se emplea como sinónimo de reiterar o repetir (entendidos como volver a desarrollar una acción o pronunciar de nuevo lo que ya se había dicho). Cabe mencionar que en informática se asocia una iteración con los términos bucle y estructura de control, que hacen referencia a las palabras reservadas *while* y *for*, entre otras. Básicamente, se suele establecer una condición que se debe cumplir para que las líneas de código dentro de dichos bucles se ejecuten. Sin embargo, en muchos casos es necesario realizar al menos una vez dichas acciones antes de la comprobación, para lo cual se usa un modelo diferente, contemplado en algunos lenguajes con estructuras como *do while*.

Todo algoritmo recursivo puede expresarse como iterativo y viceversa. Sin embargo, según las condiciones del problema a resolver podrá ser preferible utilizar la solución recursiva o la iterativa.

2.3 Producto de dos enteros

Se le conoce como producto al resultado al que se llega tras multiplicar un numero por otro.

2.4 Cociente de dos enteros

Se le conoce como cociente al resultado al que se llega tras dividir un número por otro. En este sentido, el cociente sirve para indicar qué cantidad de veces el divisor está contenido en el dividendo.

2.5 Complejidad Temporal

Se denomina complejidad temporal a la función $T(n)$ que mide el número de instrucciones realizadas por el algoritmo para procesar los n elementos de entrada. Cada instrucción tiene asociado un costo temporal. Afecta al tiempo

de ejecución el orden en que se procesen los elementos de entrada. Podría considerarse que los valores de los n casos que se presentan como entrada son los correspondientes: a un caso típico, o a un caso promedio, o de peor caso. El peor caso es el más sencillo de definir (el que demore más para cualquier entrada), pero si se desea otros tipos de entrada habría que definir qué se considera típico, o la distribución de los valores en el caso promedio.

3 Experimentación y Resultados

3.1 Producto de dos enteros positivos

Para esta parte de la practica se implementaron 3 algoritmos (iterativos y recursivos) los cuales son capaces de calcular el producto de dos enteros positivos.

3.1.1 Primer algoritmo:

```
int prod1(int m, int n)
1--      r=0
2--      while n>0
3--          r=r+m
4--          n--
5--      return r
```

a) Mejor y Peor Caso.

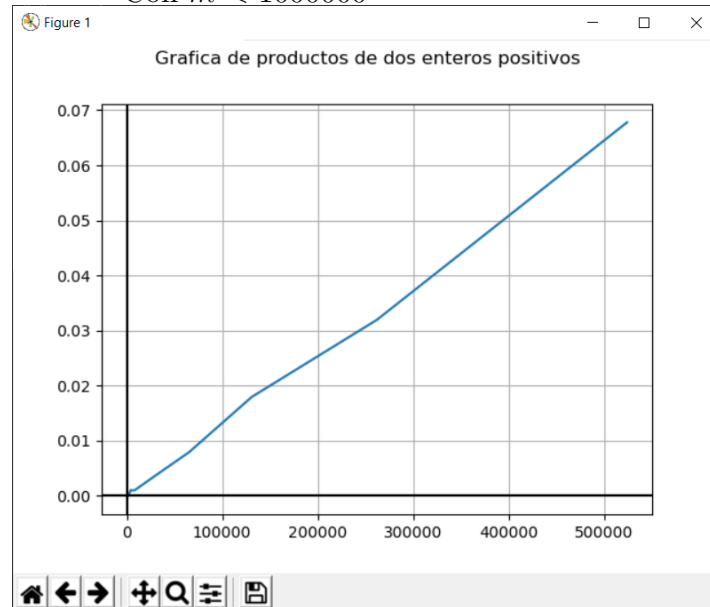
Al observar el algoritmo podemos notar que "El mejor caso" se presenta cuando n no es mayor a 0 es decir es igual ya que si es $=0$ no entra dentro del while y solo nos devuelve el valor inicial de r que es 0, esto es correcto ya que cualquier numero multiplicado por $0 = 0$.

"El peor caso" se presenta cuando n (numero ingresado) es un numero muy grande ya que al ser mayor que 0 entra al ciclo while y ejecuta las lineas de codigo dentro de el, se considera el peor caso ya que al ser muy grande este se decrementara de 1 en 1 y esto puede tardar mucho tiempo de ejecucion. Por lo tanto $prod1 \in O(n)$.

b) Propuesta de Orden de complejidad.

Para el peor caso, se graficó *tiempo* vs *números que fueratomando*, por otro lado m siempre se mantuvo en 1. A continuación se muestran algunos resultados.

Con $m < 1000000$



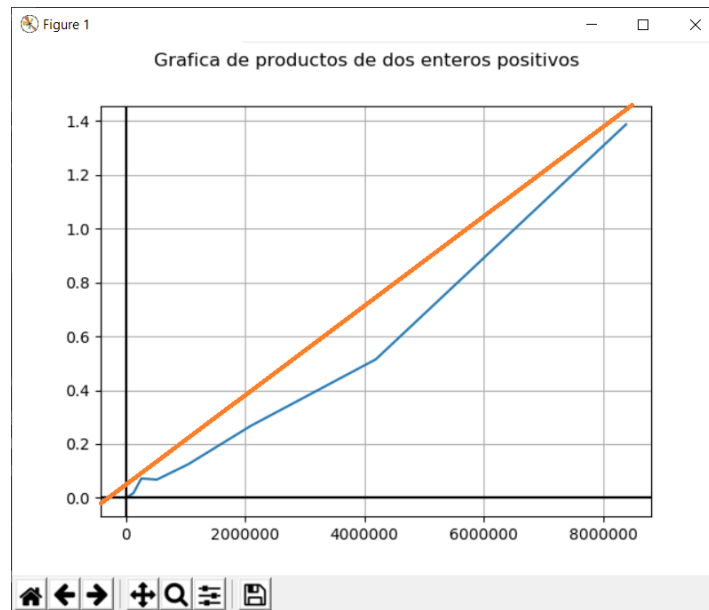
Resultado en consola

```
C:\Windows\System32\cmd.exe - py producto.py

C:\Users\52556\Documents\ESCOM\Análisis de algoritmos\P2>py pro
ducto.py
2 * 1, calculado en: 0.0 segundos --
4 * 1, calculado en: 0.0 segundos --
8 * 1, calculado en: 0.0 segundos --
16 * 1, calculado en: 0.0 segundos --
32 * 1, calculado en: 0.0 segundos --
64 * 1, calculado en: 0.0 segundos --
128 * 1, calculado en: 0.0 segundos --
256 * 1, calculado en: 0.0 segundos --
512 * 1, calculado en: 0.0 segundos --
1024 * 1, calculado en: 0.0 segundos --
2048 * 1, calculado en: 0.0 segundos --
4096 * 1, calculado en: 0.0010266304016113281 segundos --
8192 * 1, calculado en: 0.0009942054748535156 segundos --
16384 * 1, calculado en: 0.001994609832763672 segundos --
32768 * 1, calculado en: 0.003993988037109375 segundos --
65536 * 1, calculado en: 0.007946252822875977 segundos --
131072 * 1, calculado en: 0.01795196533203125 segundos --
262144 * 1, calculado en: 0.03194236755371094 segundos --
524288 * 1, calculado en: 0.06779170036315918 segundos --

C:\Users\52556\Documents\ESCOM\Análisis de algoritmos\P2>py pro
```

Con $m < 10000000$ y proponiendo $f(n) = \frac{1}{10^6}n + 0.05$ que acote nuestra función



Resultado en consola

```

C:\Windows\System32\cmd.exe - py producto.py
2 * 1, calculado en: 0.0 segundos --
4 * 1, calculado en: 0.0 segundos --
8 * 1, calculado en: 0.0 segundos --
16 * 1, calculado en: 0.0 segundos --
32 * 1, calculado en: 0.0 segundos --
64 * 1, calculado en: 0.0 segundos --
128 * 1, calculado en: 0.0 segundos --
256 * 1, calculado en: 0.0 segundos --
512 * 1, calculado en: 0.0 segundos --
1024 * 1, calculado en: 0.0009970664978027344 segundos --
2048 * 1, calculado en: 0.0 segundos --
4096 * 1, calculado en: 0.0 segundos --
8192 * 1, calculado en: 0.0009975433349609375 segundos --
16384 * 1, calculado en: 0.002017974853515625 segundos --
32768 * 1, calculado en: 0.002991914749145508 segundos --
65536 * 1, calculado en: 0.008003950119018555 segundos --
131072 * 1, calculado en: 0.01792120933532715 segundos --
262144 * 1, calculado en: 0.0718069076538086 segundos --
524288 * 1, calculado en: 0.06782174110412598 segundos --
1048576 * 1, calculado en: 0.12469816207885742 segundos --
2097152 * 1, calculado en: 0.2673208713531494 segundos --
4194304 * 1, calculado en: 0.5146243572235107 segundos --
8388608 * 1, calculado en: 1.3872900009155273 segundos --

```

c) Orden de complejidad de manera analítica

Procederemos a calcular el orden de complejidad mediante el metodo por bloques. Observando el algoritmo notamos que las lineas 1,3 y 4 son $O(1)$, al mismo tiempo que la línea 2 es $O(n)$

Por lo que
 $prod1 \in O(1) + O(n)$
 $\therefore prod1 \in O(n)$

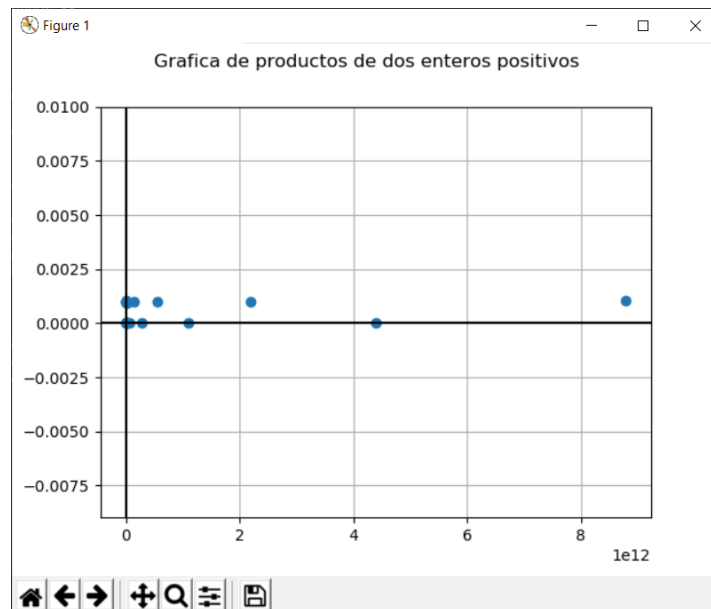
3.1.2 Segundo algoritmo:

```
int prod2(int m, int n)
1--      r=0
2--      while n>0
3--          if n & 1
4--              r=r+m
5--          m=2*m
6--          n=n/2
7--      return r
```

a) Mejor y Peor Caso.

Este algoritmo tiene un mejor y peor caso como el anterior, cuando $n < 0$, no se entrará al while y por tanto $prod2 \in \Omega(1)$. Mientras que cuando n sea un número muy grande, la función realizará más pasos por entrar al *while*, así $prod2 \in O(\log_2 n)$

b) Propuesta de Orden de complejidad



Con $m < 10000000$

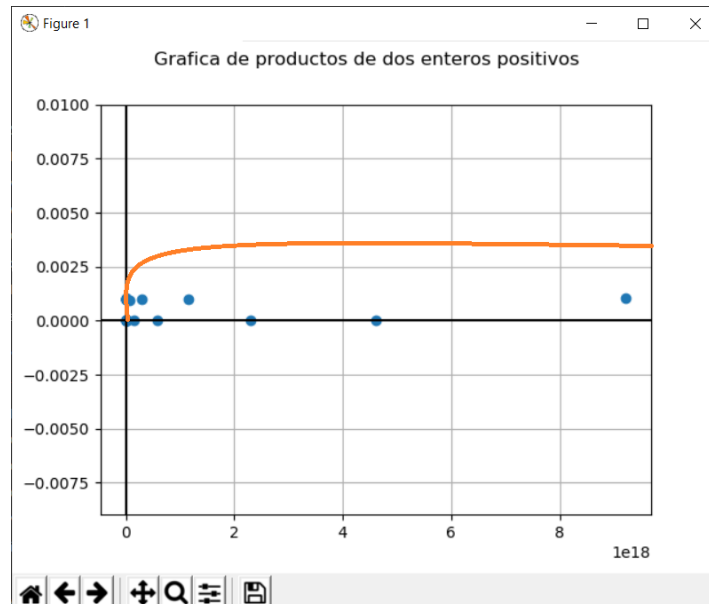
Resultado en consola

```

C:\Windows\System32\cmd.exe
1048576 * 1, calculado en: 0.000997304916381836 segundos --
2097152 * 1, calculado en: 0.0 segundos --
4194304 * 1, calculado en: 0.000997304916381836 segundos --
8388608 * 1, calculado en: 0.0 segundos --
16777216 * 1, calculado en: 0.0009975433349609375 segundos --
33554432 * 1, calculado en: 0.0 segundos --
67108864 * 1, calculado en: 0.0009975433349609375 segundos --
134217728 * 1, calculado en: 0.0 segundos --
268435456 * 1, calculado en: 0.0010013580322265625 segundos --
536870912 * 1, calculado en: 0.0 segundos --
1073741824 * 1, calculado en: 0.0009932518005371094 segundos --
2147483648 * 1, calculado en: 0.0 segundos --
4294967296 * 1, calculado en: 0.0009970664978027344 segundos --
8589934592 * 1, calculado en: 0.0 segundos --
17179869184 * 1, calculado en: 0.0010073184967041016 segundos --
34359738368 * 1, calculado en: 0.0009646415710449219 segundos --
68719476736 * 1, calculado en: 0.0 segundos --
137438953472 * 1, calculado en: 0.0009970664978027344 segundos --
274877906944 * 1, calculado en: 0.0 segundos --
549755813888 * 1, calculado en: 0.000997304916381836 segundos --
1099511627776 * 1, calculado en: 0.0 segundos --
2199023255552 * 1, calculado en: 0.000997304916381836 segundos --
4398046511104 * 1, calculado en: 0.0 segundos --
8796093022208 * 1, calculado en: 0.0010285377502441406 segundos --

```

Con $m < 10000000$, proponiendo un $f(n)$ que acote nuestra función se propuso $f(n) = \frac{1}{10^{3.9}} \log_2 n$



Resultado en consola

```

C:\Windows\System32\cmd.exe
1099511627776 * 1, calculado en: 0.0 segundos --
2199023255552 * 1, calculado en: 0.0009975433349609375 segundos --
4398046511104 * 1, calculado en: 0.0 segundos --
8796093022208 * 1, calculado en: 0.000997304916381836 segundos --
17592186044416 * 1, calculado en: 0.0 segundos --
35184372088832 * 1, calculado en: 0.000997304916381836 segundos --
70368744177664 * 1, calculado en: 0.0009975433349609375 segundos --
140737488355328 * 1, calculado en: 0.0 segundos --
281474976710656 * 1, calculado en: 0.000997304916381836 segundos --
562949953421312 * 1, calculado en: 0.0 segundos --
1125899906842624 * 1, calculado en: 0.000997304916381836 segundos --
2251799813685248 * 1, calculado en: 0.0 segundos --
4503599627370496 * 1, calculado en: 0.0010297298431396484 segundos --
9007199254740992 * 1, calculado en: 0.0 segundos --
18014398509481984 * 1, calculado en: 0.0010008811950683594 segundos --
36028797018963968 * 1, calculado en: 0.0 segundos --
72057594037927936 * 1, calculado en: 0.0009620189666748047 segundos --
144115188075855872 * 1, calculado en: 0.0 segundos --
288230376151711744 * 1, calculado en: 0.000997304916381836 segundos --
576460752303423488 * 1, calculado en: 0.0 segundos --
1152921504606846976 * 1, calculado en: 0.000997304916381836 segundos --
2305843009213693952 * 1, calculado en: 0.0 segundos --
4611686018427387904 * 1, calculado en: 0.0 segundos --
9223372036854775808 * 1, calculado en: 0.0010313987731933594 segundos --

```

Bien se puede observar que los puntos graficados pueden ser acotados superiormente por una función $\log n$

c) Orden de complejidad de manera analítica

Procederemos a calcular el orden de complejidad mediante el método por bloques. Observando el algoritmo notamos que las líneas 1,3,4 y 5 son $O(1)$, al mismo tiempo que la línea 2 es $O(n)$ y la línea 6 es $O(\log_2 n)$. Por lo que

$$\text{prod1} \in O(1) + O(n) + O(\log_2 n)$$

$$\therefore \text{prod1} \in O(\log_2 n)$$

3.1.3 Tercer algoritmo (recursivo):

```

int prod3(int a, int b)
1--      if b==1
2--                return a
3--      else
4--                return a+prod3(a,b-1)

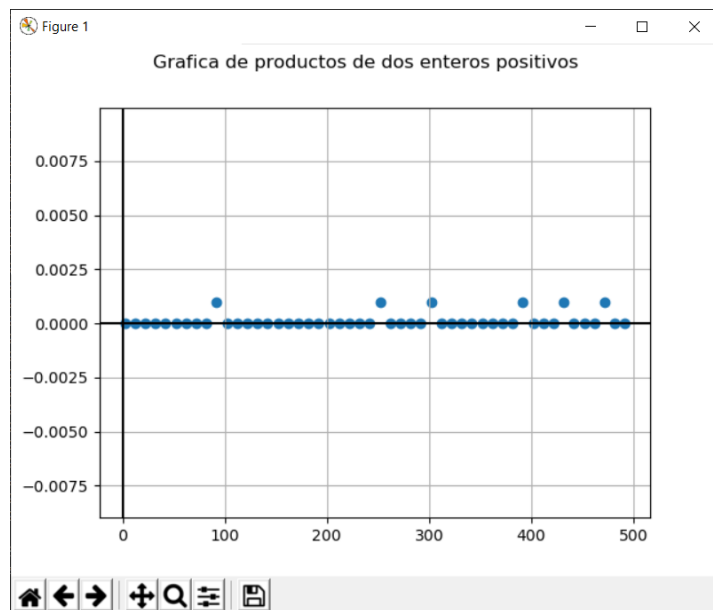
```

a) Mejor y Peor Caso.

Al analizar el algoritmo presentado nos podemos dar cuenta que "El mejor caso" se presenta cuando el número asignado como b es $= 1$ ya

que solo entrará a la sentencia if y nos retornará el valor de nuestro otro numero, esto es correcto debido a que cualquier numero a multiplicado por 1 nos da como resultado nuestro mismo numero a . "El peor caso" se presenta cuando los dos numeros ingresados son diferentes a 1 ya que entra a nuestro caso else y nos retorna la funcion recursiva la cual requiere mayor tiempo y uso de memoria que el mejor caso antes presentado.

b) Propuesta de Orden de Complejidad



Con $b < 500$

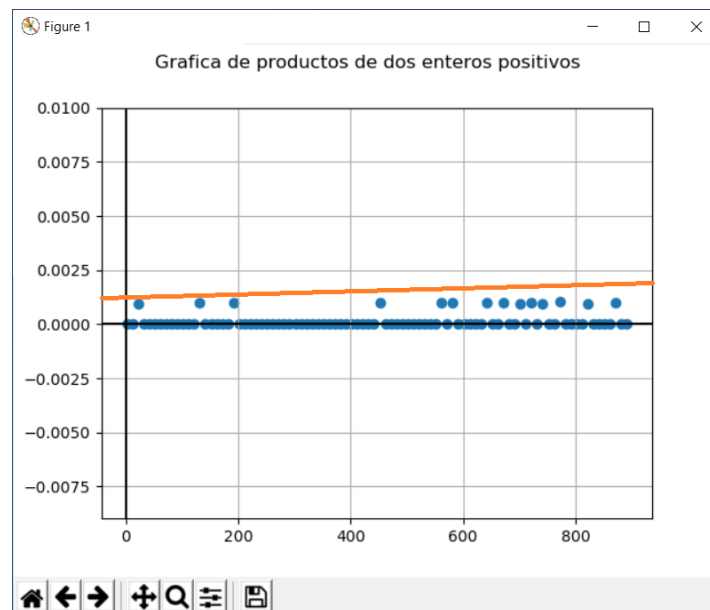
Resultado en consola

```

C:\Windows\System32\cmd.exe
262 * 1, calculado en: 0.0 segundos --
272 * 1, calculado en: 0.0 segundos --
282 * 1, calculado en: 0.0 segundos --
292 * 1, calculado en: 0.0 segundos --
302 * 1, calculado en: 0.000997304916381836 segundos --
312 * 1, calculado en: 0.0 segundos --
322 * 1, calculado en: 0.0 segundos --
332 * 1, calculado en: 0.0 segundos --
342 * 1, calculado en: 0.0 segundos --
352 * 1, calculado en: 0.0 segundos --
362 * 1, calculado en: 0.0 segundos --
372 * 1, calculado en: 0.0 segundos --
382 * 1, calculado en: 0.0 segundos --
392 * 1, calculado en: 0.0009989738464355469 segundos --
402 * 1, calculado en: 0.0 segundos --
412 * 1, calculado en: 0.0 segundos --
422 * 1, calculado en: 0.0 segundos --
432 * 1, calculado en: 0.0009968280792236328 segundos --
442 * 1, calculado en: 0.0 segundos --
452 * 1, calculado en: 0.0 segundos --
462 * 1, calculado en: 0.0 segundos --
472 * 1, calculado en: 0.0009720325469970703 segundos --
482 * 1, calculado en: 0.0 segundos --
492 * 1, calculado en: 0.0 segundos --

```

Con $b < 900$ y $f(n) = \frac{1}{10^6} + 0.001$



Resultado en consola

```

C:\Windows\System32\cmd.exe
662 * 1, calculado en: 0.0 segundos --
672 * 1, calculado en: 0.0010044574737548828 segundos --
682 * 1, calculado en: 0.0 segundos --
692 * 1, calculado en: 0.0 segundos --
702 * 1, calculado en: 0.0009717941284179688 segundos --
712 * 1, calculado en: 0.0 segundos --
722 * 1, calculado en: 0.0009944438934326172 segundos --
732 * 1, calculado en: 0.0 segundos --
742 * 1, calculado en: 0.0009636878967285156 segundos --
752 * 1, calculado en: 0.0 segundos --
762 * 1, calculado en: 0.0 segundos --
772 * 1, calculado en: 0.0010294914245605469 segundos --
782 * 1, calculado en: 0.0 segundos --
792 * 1, calculado en: 0.0 segundos --
802 * 1, calculado en: 0.0 segundos --
812 * 1, calculado en: 0.0 segundos --
822 * 1, calculado en: 0.0009636878967285156 segundos --
832 * 1, calculado en: 0.0 segundos --
842 * 1, calculado en: 0.0 segundos --
852 * 1, calculado en: 0.0 segundos --
862 * 1, calculado en: 0.0 segundos --
872 * 1, calculado en: 0.0009958744049072266 segundos --
882 * 1, calculado en: 0.0 segundos --
892 * 1, calculado en: 0.0 segundos --

```

Es importante destacar que este algoritmo al ser recursivo esta limitado por la capacidad de la pila que se tiene. Aún así, las gráficas muestran que se puede acotar estas funciones por una función lineal n

c) Orden de complejidad de manera Analitica.

Para calcular el orden de complejidad usaremos el método por bloques, observando podemos deprecia la primer sentencia if y solamente analizar la sentencia del else ya que es aquí donde se implementa la recursividad. A continuacion observamos que la recurrencia es

$$T(n) = \begin{cases} c & \text{si } n = 0 \\ T(n-1) + 1 & \text{si } n > 0 \end{cases} \quad (1)$$

A continuacion la resolveremos mediante el metodo decremento hacia atras

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 2 \end{aligned}$$

.

.

.

$$= T(n-i) + i$$

.

.

$$\begin{aligned}
 &= T(0) + n = c + n \\
 \therefore \text{div3} &\in O(n)
 \end{aligned}$$

3.1.4 Determinación del mejor algoritmo

Una vez analizado todos los 3 algoritmos, podemos concluir el que más eficiente es, es el que tiene un orden de complejidad menor, es decir, el segundo algoritmo tuvo $O(\log_2 n)$ y claramente se ve en las gráficas el tiempo que se tarda para hacer los cálculos. Mientras que el peor, puede ser el recursivo, es lineal como el primero, i.e.: $O(n)$, sin embargo con este algoritmo se tiene una gran desventaja, ya que está restringido a la pila que se usa en la recursividad, y es casi imposible calcular multiplicaciones grandes con este algoritmo.

3.2 Cociente de dos enteros positivos

3.2.1 Primer algoritmo

El primer algoritmo de esta práctica se muestra a continuación

```

int div1 (int n, int div, int *r)
1-- q = 0
2--     while n>=div
3--         n = n-div
4--         q++
5-- *r=n
6-- return q

```

3.2.2 Mejor y peor caso

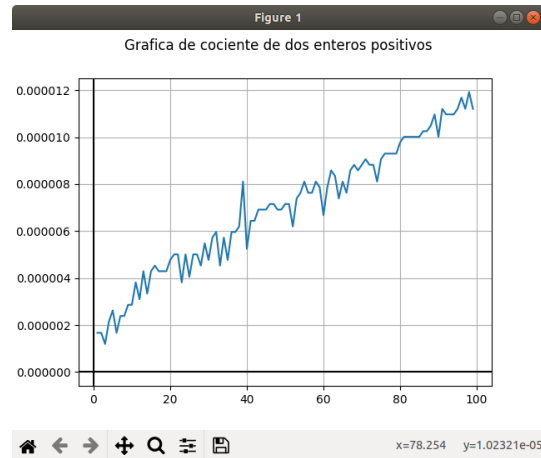
Como se puede observar **el mejor caso** es cuando n es mayor a div , es decir, cuando el numerador es mayor que el denominador, en tal caso, no se entrará al ciclo *while* y como se observa nuestra función $div1 \in \Omega(1)$

Por otro lado en **el peor de los casos** se tiene cuando nuestra div , es decir el numerador, es la mínima unidad de los enteros, en otras palabras, cuando $div = 1$, ya que en este caso, se recorrerá el while de manera lineal por div hasta que $n \geq div$ y como se dijo antes, este se recorrerá de uno en uno. Así, $div1 \in O(n)$

3.2.3 Propuesta de orden de complejidad a partir de datos experimentales

Con datos experimentales podemos observar una situación similar, las siguientes gráficas que se mostrarán a continuación serán todas usando en nuestro $div = 1 = \text{dividendo}$ y con diferentes valores de $n = \text{numerador}$

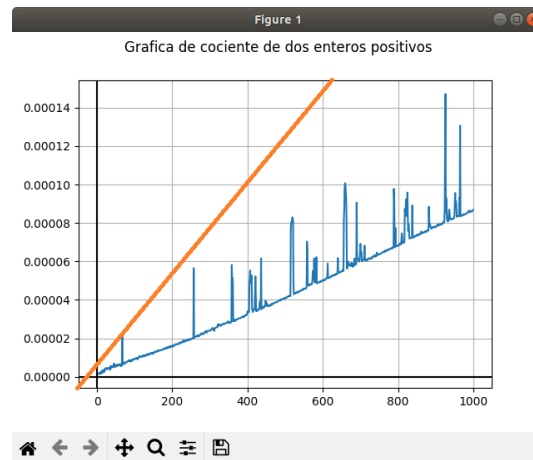
Con div hasta 100



Resultado en consola:

```
edoomm@Satellite-L735: ~/Documents/ESCOM/Analisis de algo...
File Edit View Search Terminal Help
76 / 1, calculado en: 5.7220458984375e-06 segundos --
77 / 1, calculado en: 5.9604644775390625e-06 segundos --
78 / 1, calculado en: 5.9604644775390625e-06 segundos --
79 / 1, calculado en: 5.9604644775390625e-06 segundos --
80 / 1, calculado en: 6.198883056640625e-06 segundos --
81 / 1, calculado en: 6.4373016357421875e-06 segundos --
82 / 1, calculado en: 6.4373016357421875e-06 segundos --
83 / 1, calculado en: 6.4373016357421875e-06 segundos --
84 / 1, calculado en: 6.198883056640625e-06 segundos --
85 / 1, calculado en: 6.9141387939453125e-06 segundos --
86 / 1, calculado en: 6.67572021484375e-06 segundos --
87 / 1, calculado en: 6.9141387939453125e-06 segundos --
88 / 1, calculado en: 7.152557373046875e-06 segundos --
89 / 1, calculado en: 7.152557373046875e-06 segundos --
90 / 1, calculado en: 6.67572021484375e-06 segundos --
91 / 1, calculado en: 7.3909759521484375e-06 segundos --
92 / 1, calculado en: 7.62939453125e-06 segundos --
93 / 1, calculado en: 7.3909759521484375e-06 segundos --
94 / 1, calculado en: 7.3909759521484375e-06 segundos --
95 / 1, calculado en: 7.62939453125e-06 segundos --
96 / 1, calculado en: 7.62939453125e-06 segundos --
97 / 1, calculado en: 7.62939453125e-06 segundos --
98 / 1, calculado en: 7.867813110351562e-06 segundos --
99 / 1, calculado en: 7.867813110351562e-06 segundos --
```

Ahora con div hasta 1000, se obtiene una gráfica muy similar, lo que claramente nos indica que se trata, en su peor caso, que es lineal. Y se propone una $f(n) = \frac{1}{10^{6.5}}n$



Resultado en consola

```

edoomm@Satellite-L735: ~/Documents/ESCOM/Analisis de alg...
File Edit View Search Terminal Help
953 / 1, calculado en: 8.845329284667969e-05 segundos --
954 / 1, calculado en: 9.131431579589844e-05 segundos --
955 / 1, calculado en: 8.96453857421875e-05 segundos --
956 / 1, calculado en: 8.320808410644531e-05 segundos --
957 / 1, calculado en: 8.344650268554688e-05 segundos --
958 / 1, calculado en: 8.344650268554688e-05 segundos --
959 / 1, calculado en: 8.344650268554688e-05 segundos --
960 / 1, calculado en: 8.368492126464844e-05 segundos --
961 / 1, calculado en: 8.392333984375e-05 segundos --
962 / 1, calculado en: 8.344650268554688e-05 segundos --
963 / 1, calculado en: 9.322166442871094e-05 segundos --
964 / 1, calculado en: 8.726119995117188e-05 segundos --
965 / 1, calculado en: 0.00013065338134765625 segundos --
966 / 1, calculado en: 8.392333984375e-05 segundos --
967 / 1, calculado en: 8.344650268554688e-05 segundos --
968 / 1, calculado en: 8.368492126464844e-05 segundos --
969 / 1, calculado en: 8.392333984375e-05 segundos --
970 / 1, calculado en: 8.392333984375e-05 segundos --
971 / 1, calculado en: 8.416175842285156e-05 segundos --
972 / 1, calculado en: 8.392333984375e-05 segundos --
973 / 1, calculado en: 8.416175842285156e-05 segundos --
974 / 1, calculado en: 8.416175842285156e-05 segundos --
975 / 1, calculado en: 8.463859558105469e-05 segundos --
976 / 1, calculado en: 8.440017700195312e-05 segundos --

```

Los picos que se pueden producir son tal vez a que la computadora estaba recién empezando a prenderse y este puede ser ese factor que origina esos picos inesperados, por los procesos que está iniciando el sistema operativo. Aún así, claramente se puede observar que es lineal este algoritmo en su peor caso. Por lo que, a partir de datos experimentales se llega a que $div1 \in O(n)$

3.2.4 Cálculo analítico del orden de complejidad

Calcularemos el orden de complejidad por bloques, ya que este resulta un poco más fácil de entender y de analizar.

Observando el algoritmo, se ve que las líneas 1, 3, 4, 5 y 6 son $O(1)$, mientras que la línea 2 es $O(n)$

Por lo que

$$\begin{aligned} div1 &\in O(1) + O(n) \\ \therefore div1 &\in O(n) \end{aligned}$$

3.2.5 Segundo algoritmo

Para este segundo algoritmo se tuvo el siguiente pseudocódigo

```
int div2(int n, int div, int *r)
1-- int dd=div
2-- int q=0
3-- *r=n
4-- while dd<=n
5--     dd=2*dd
6-- while dd>div
7--     dd=dd/2
8--     q=2*q
9--     if (dd<=*r)
10--         *r=*r-dd
11--         q++
12-- return q
```

3.2.6 Mejor y peor caso

Como se ha observado en algoritmos anteriores el mejor caso será cuando recorre menos pasos nuestros algoritmos. En este caso, en el algoritmo que se está analizando vemos que el menor número de pasos se dará cuando $div > n$, i.e., cuando el *denominador* $>$ *numerador*, ya que no entrará a los *while* de las líneas 4 y 6. Por lo que:

$$div2 \in \Omega(1)$$

Mientras que por otro lado, para el peor caso se tiene que cumplir la misma condición que el primer algoritmo, que nuestro *denominador* $= 1$, en algunos casos, este algoritmo, sigue la misma cantidad de pasos con *denomindaor* $= 2$ o hasta a veces con *denomindaor* $= 3$, pero siempre se observa que el peor caso, con el mayor número de pasos se observa cuando $div = 1$. En tal caso, observamos que, siendo $div = 1$, la variable *dd* también será 1 al inicio, y esta se irá incrementando por 2 hasta que $dd > n$, por lo que desde ahí ya se puede

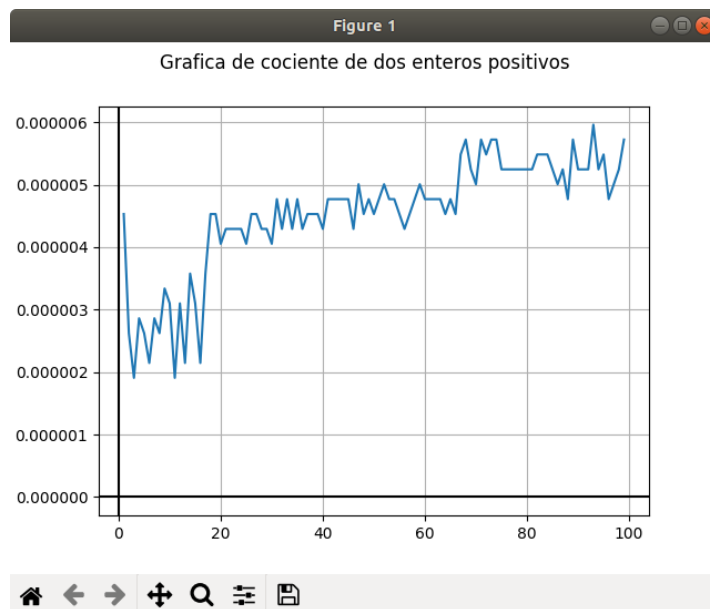
observar la complejidad que tendrá nuestro algoritmo, dado que el siguiente ciclo va decrementando entre 2 y se puede ver que los dos ciclos tienen la misma complejidad en el peor de los casos. Así:

$$\text{div2} \in O(\log_2 n)$$

3.2.7 Propuesta de orden de complejidad a partir de datos experimentales

Para el peor caso, se graficó *tiempo* vs *números que el numerador fue tomando*, el *denominador* siempre se mantuvo en 1. A continuación se muestran algunos resultados.

Con *numerador* < 100



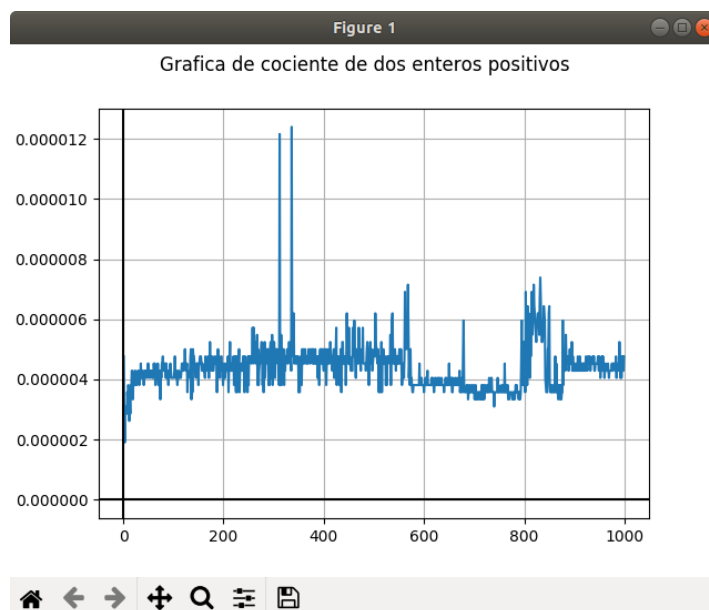
Resultado en consola


```

edoomm@Satellite-L735: ~/Documents/ESCOM/)Análisis de al...
File Edit View Search Terminal Help
74 / 1, calculado en: 5.7220458984375e-06 segundos --
75 / 1, calculado en: 5.245208740234375e-06 segundos --
76 / 1, calculado en: 5.245208740234375e-06 segundos --
77 / 1, calculado en: 5.245208740234375e-06 segundos --
78 / 1, calculado en: 5.245208740234375e-06 segundos --
79 / 1, calculado en: 5.245208740234375e-06 segundos --
80 / 1, calculado en: 5.245208740234375e-06 segundos --
81 / 1, calculado en: 5.245208740234375e-06 segundos --
82 / 1, calculado en: 5.4836273193359375e-06 segundos --
83 / 1, calculado en: 5.4836273193359375e-06 segundos --
84 / 1, calculado en: 5.4836273193359375e-06 segundos --
85 / 1, calculado en: 5.245208740234375e-06 segundos --
86 / 1, calculado en: 5.0067901611328125e-06 segundos --
87 / 1, calculado en: 5.245208740234375e-06 segundos --
88 / 1, calculado en: 4.76837158203125e-06 segundos --
89 / 1, calculado en: 5.7220458984375e-06 segundos --
90 / 1, calculado en: 5.245208740234375e-06 segundos --
91 / 1, calculado en: 5.245208740234375e-06 segundos --
92 / 1, calculado en: 5.245208740234375e-06 segundos --
93 / 1, calculado en: 5.9604644775390625e-06 segundos --
94 / 1, calculado en: 5.245208740234375e-06 segundos --
95 / 1, calculado en: 5.4836273193359375e-06 segundos --
96 / 1, calculado en: 4.76837158203125e-06 segundos --
97 / 1, calculado en: 5.0067901611328125e-06 segundos --

```

Con $numerador < 1000$



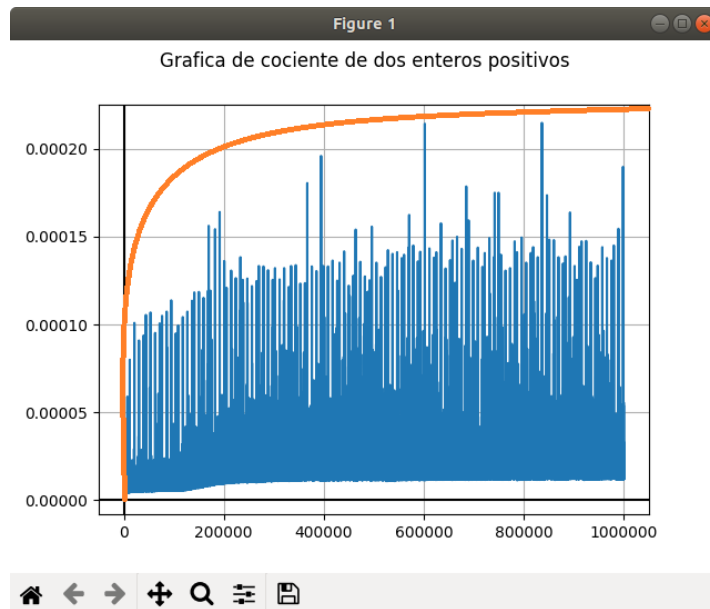
Resultado en consola

```

edoomm@Satellite-L735: ~/Documents/ESCOM/)Análisis de al...
File Edit View Search Terminal Help
982 / 1, calculado en: 4.291534423828125e-06 segundos --
983 / 1, calculado en: 4.76837158203125e-06 segundos --
984 / 1, calculado en: 4.0531158447265625e-06 segundos -
-
985 / 1, calculado en: 4.5299530029296875e-06 segundos -
-
986 / 1, calculado en: 4.291534423828125e-06 segundos --
987 / 1, calculado en: 4.291534423828125e-06 segundos --
988 / 1, calculado en: 4.5299530029296875e-06 segundos -
-
989 / 1, calculado en: 4.291534423828125e-06 segundos --
990 / 1, calculado en: 5.245208740234375e-06 segundos --
991 / 1, calculado en: 4.5299530029296875e-06 segundos -
-
992 / 1, calculado en: 4.0531158447265625e-06 segundos -
-
993 / 1, calculado en: 4.76837158203125e-06 segundos --
994 / 1, calculado en: 4.0531158447265625e-06 segundos -
-
995 / 1, calculado en: 4.5299530029296875e-06 segundos -
-
996 / 1, calculado en: 4.76837158203125e-06 segundos --
997 / 1, calculado en: 4.291534423828125e-06 segundos --
998 / 1, calculado en: 4.291534423828125e-06 segundos --

```

Finalmente con $numerador < 1000000$ y con $f(n) = \frac{1}{10^5} \log_2(n)$



Resultado en consola

```

edoomm@Satellite-L735: ~/Documents/ESCOM/Analisis de algoritmo...
File Edit View Search Terminal Help
999975 / 1, calculado en: 1.33514404296875e-05 segundos --
999976 / 1, calculado en: 1.239776611328125e-05 segundos --
999977 / 1, calculado en: 1.4543533325195312e-05 segundos --
999978 / 1, calculado en: 1.5497207641601562e-05 segundos --
999979 / 1, calculado en: 1.2636184692382812e-05 segundos --
999980 / 1, calculado en: 1.2159347534179688e-05 segundos --
999981 / 1, calculado en: 1.2874603271484375e-05 segundos --
999982 / 1, calculado en: 1.5497207641601562e-05 segundos --
999983 / 1, calculado en: 1.3828277587890625e-05 segundos --
999984 / 1, calculado en: 1.3113021850585938e-05 segundos --
999985 / 1, calculado en: 1.2636184692382812e-05 segundos --
999986 / 1, calculado en: 1.3589859008789062e-05 segundos --
999987 / 1, calculado en: 1.33514404296875e-05 segundos --
999988 / 1, calculado en: 1.33514404296875e-05 segundos --
999989 / 1, calculado en: 1.3113021850585938e-05 segundos --
999990 / 1, calculado en: 1.3589859008789062e-05 segundos --
999991 / 1, calculado en: 2.8848648071289062e-05 segundos --
999992 / 1, calculado en: 1.3589859008789062e-05 segundos --
999993 / 1, calculado en: 1.3589859008789062e-05 segundos --
999994 / 1, calculado en: 1.33514404296875e-05 segundos --
999995 / 1, calculado en: 1.5497207641601562e-05 segundos --
999996 / 1, calculado en: 1.239776611328125e-05 segundos --
999997 / 1, calculado en: 1.2874603271484375e-05 segundos --
999998 / 1, calculado en: 1.3589859008789062e-05 segundos --

```

Así se observa que efectivamente el algoritmo tiene una complejidad de $\log n$, en la última prueba con 1,000,000 se puede observar que donde más intenso es el azul de los puntos de donde cayeron el tiempo y los números, es efectivamente donde yace la gráfica $f(x) = \log n$. Por lo tanto

$$\text{div2} \in O(\log_2 n)$$

3.2.8 Cálculo analítico del orden de complejidad

Para este algoritmo de igual manera su orden de complejidad se calculará por bloques.

De la línea 1 a la 3 y la 12 también, se observa que tienen orden de complejidad $O(1)$. Mientras que la 4 y 5 tienen orden de complejidad $O(\log_2 n)$, ya que la 5 tiene $O(1)$, mientras que la 4, se podría decir que es dependiente de lo que pase en la línea 5, y se tiene una expresión donde la variable dd incrementará por dos su valor hasta llegar a la variable n . Por lo que las líneas 4 y 5 son $O(\log_2 n)$. Después se observa que las líneas 7 hasta la 11 tienen orden de complejidad $O(1)$. Mientras que la línea 6, cumple el mismo caso que la línea 4, así, de la línea 6 a la 11 tienen un orden de complejidad $O(\log_2 n)$. Por lo tanto:

$$\begin{aligned} \text{div2} &\in O(1) + O(\log_2 n) + O(\log_2 n) \\ \therefore \text{div2} &\in O(\log_2 n) \end{aligned}$$

3.2.9 Tercer algoritmo

El último algoritmo de la práctica fue el siguiente, un algoritmo recursivo para el cálculo de una división.

```
int div3 (int n, int div)
1-- if div>n
2--     return 0
3-- else
4--     return 1 + div3(n-div, div)
```

A simple vista puede parecer un mejor algoritmo, por las pocas líneas de código que tiene, sin embargo, es recursivo, y se tendrá que calcular su recurrencia para saber si sí es un mejor algoritmo o no.

3.2.10 Mejor y peor caso

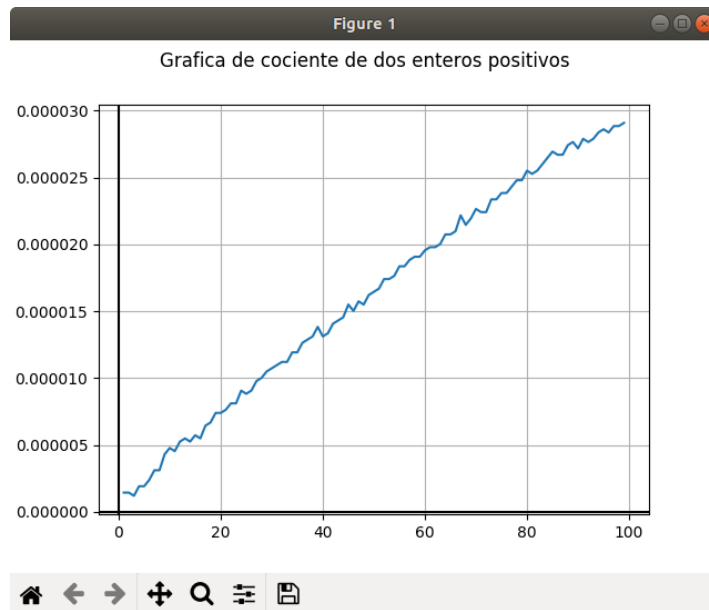
El mejor caso, otra vez se observa que se da cuando $div > n$, i.e., *denominador* > *numerador*. En dado caso esta función entrará al primer *if* y retornará un 0. Así,

$$div3 \in \Omega(1)$$

Por otro lado, para el peor caso, se observa que se da como en las otras funciones, cuando $div = denominador = 1$, ya que en este caso irá haciendo sumas de 1 en 1 hasta que $div > n$. Sin embargo para calcular O , necesitaremos calcular su recurrencia, misma que se verá más adelante en el punto 3.2.12.

3.2.11 Propuesta de orden de complejidad a partir de datos experimentales

De igual manera, se siguieron la misma serie de pasos para observar la gráfica *tiempo* vs *números*, con las que se usaron en los dos algoritmos anteriores. Con *numerador* < 100



Resultado en consola

```

edoomm@Satellite-L735: ~/Documents/ESCOM/Analisis de al...
File Edit View Search Terminal Help
77 / 1, calculado en: 2.4318695068359375e-05 segundos --
78 / 1, calculado en: 2.47955322265625e-05 segundos --
79 / 1, calculado en: 2.47955322265625e-05 segundos --
80 / 1, calculado en: 2.5510787963867188e-05 segundos --
81 / 1, calculado en: 2.5272369384765625e-05 segundos --
82 / 1, calculado en: 2.5510787963867188e-05 segundos --
83 / 1, calculado en: 2.5987625122070312e-05 segundos --
84 / 1, calculado en: 2.6464462280273438e-05 segundos --
85 / 1, calculado en: 2.6941299438476562e-05 segundos --
86 / 1, calculado en: 2.6702880859375e-05 segundos --
87 / 1, calculado en: 2.6702880859375e-05 segundos --
88 / 1, calculado en: 2.7418136596679688e-05 segundos --
89 / 1, calculado en: 2.765655517578125e-05 segundos --
90 / 1, calculado en: 2.7179718017578125e-05 segundos --
91 / 1, calculado en: 2.7894973754882812e-05 segundos --
92 / 1, calculado en: 2.765655517578125e-05 segundos --
93 / 1, calculado en: 2.7894973754882812e-05 segundos --
94 / 1, calculado en: 2.8371810913085938e-05 segundos --
95 / 1, calculado en: 2.86102294921875e-05 segundos --
96 / 1, calculado en: 2.8371810913085938e-05 segundos --
97 / 1, calculado en: 2.8848648071289062e-05 segundos --
98 / 1, calculado en: 2.8848648071289062e-05 segundos --
99 / 1, calculado en: 2.9087066650390625e-05 segundos --

```

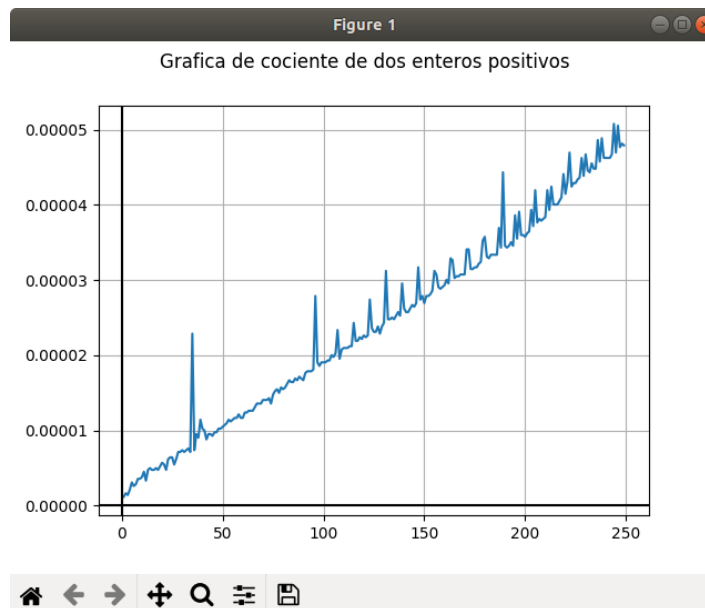
Con $numerador < 1000$ se quiso probar, sin embargo, a partir de $numerador = 998$, la pila excede su capacidad y no es posible hacer divisiones de tales magnitudes con este algoritmo.

```

edoomm@Satellite-L735: ~/Documents/ESCOM/)Análisis de al...
File Edit View Search Terminal Help
992 / 1, calculado en: 0.0003800392150878906 segundos --
993 / 1, calculado en: 0.0003783702850341797 segundos --
994 / 1, calculado en: 0.0003762245178222656 segundos --
995 / 1, calculado en: 0.0003743171691894531 segundos --
996 / 1, calculado en: 0.00037860870361328125 segundos -
-
997 / 1, calculado en: 0.00036978721618652344 segundos -
-
Traceback (most recent call last):
  File "E21.py", line 47, in <module>
    div3(i, 1)
  File "E21.py", line 40, in div3
    return 1 + div3(n - div, div)
  File "E21.py", line 40, in div3
    return 1 + div3(n - div, div)
  File "E21.py", line 40, in div3
    return 1 + div3(n - div, div)
  [Previous line repeated 995 more times]
  File "E21.py", line 37, in div3
    if div > n:
RecursionError: maximum recursion depth exceeded in compa
rison
edoomm@Satellite-L735:~/Documents/ESCOM/)Análisis de algo
ritmos/P2$

```

Entonces se probó con *numerador* < 250



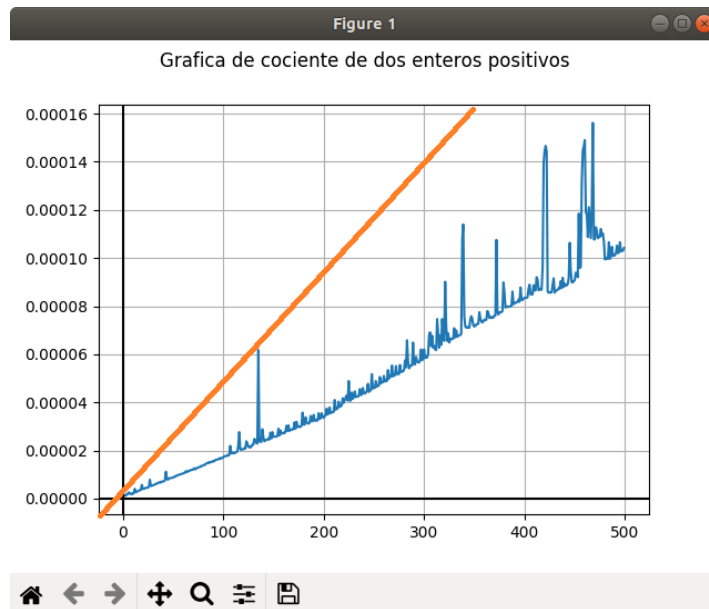
Resultado en consola

```

edoomm@Satellite-L735: ~/Documents/ESCOM/)Análisis de al...
File Edit View Search Terminal Help
-
233 / 1, calculado en: 4.553794860839844e-05 segundos --
234 / 1, calculado en: 4.482269287109375e-05 segundos --
235 / 1, calculado en: 4.482269287109375e-05 segundos --
236 / 1, calculado en: 4.863739013671875e-05 segundos --
237 / 1, calculado en: 4.57763671875e-05 segundos --
238 / 1, calculado en: 4.887580871582031e-05 segundos --
239 / 1, calculado en: 4.6253204345703125e-05 segundos --
-
240 / 1, calculado en: 4.6253204345703125e-05 segundos -
-
241 / 1, calculado en: 4.6253204345703125e-05 segundos -
-
242 / 1, calculado en: 4.6253204345703125e-05 segundos -
-
243 / 1, calculado en: 4.673004150390625e-05 segundos --
244 / 1, calculado en: 5.078315734863281e-05 segundos --
245 / 1, calculado en: 4.696846008300781e-05 segundos --
246 / 1, calculado en: 5.054473876953125e-05 segundos --
247 / 1, calculado en: 4.76837158203125e-05 segundos --
248 / 1, calculado en: 4.8160552978515625e-05 segundos -
-
249 / 1, calculado en: 4.792213439941406e-05 segundos --

```

Y finalmente con $numerador < 500$ y $f(n) = \frac{1}{10^{6.5}}n$



Resultado en consola

```

edoomm@Satellite-L735: ~/Documents/ESCOM/)Análisis de al...
File Edit View Search Terminal Help
-
488 / 1, calculado en: 0.00010085105895996094 segundos -
-
489 / 1, calculado en: 0.00010132789611816406 segundos -
-
490 / 1, calculado en: 0.0001010894775390625 segundos --
491 / 1, calculado en: 0.00010204315185546875 segundos -
-
492 / 1, calculado en: 0.00010514259338378906 segundos -
-
493 / 1, calculado en: 0.00010204315185546875 segundos -
-
494 / 1, calculado en: 0.00010228157043457031 segundos -
-
495 / 1, calculado en: 0.00010657310485839844 segundos -
-
496 / 1, calculado en: 0.000102996826171875 segundos --
497 / 1, calculado en: 0.00010275840759277344 segundos -
-
498 / 1, calculado en: 0.00010323524475097656 segundos -
-
499 / 1, calculado en: 0.00010418891906738281 segundos -
-

```

Con las gráficas mostradas se pueden observar que son muy similares a las del primer algoritmo. Entonces, se puede concluir que a partir de los datos experimentales,

$$\text{div3} \in O(n)$$

3.2.12 Cálculo analítico del orden de complejidad

Calculando por bloques esta función, se observa que se deprecia la primera sentencia del *if* y únicamente se analiza, la sentencia del *else* donde se implementa la recursividad.

Para esto se observa que la recurrencia es de la siguiente manera,

$$T(n) = \begin{cases} c & \text{si } n = 0 \\ T(n-1) + 1 & \text{si } n > 0 \end{cases} \quad (2)$$

Resolviendo esta recurrencia se tiene

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 2 \end{aligned}$$

.

.

.

$$= T(n-i) + i$$

.

.

$$\begin{aligned}
 &= T(0) + n = c + n \\
 \therefore \text{div3} &\in O(n)
 \end{aligned}$$

3.2.13 Determinación de mejor algoritmo

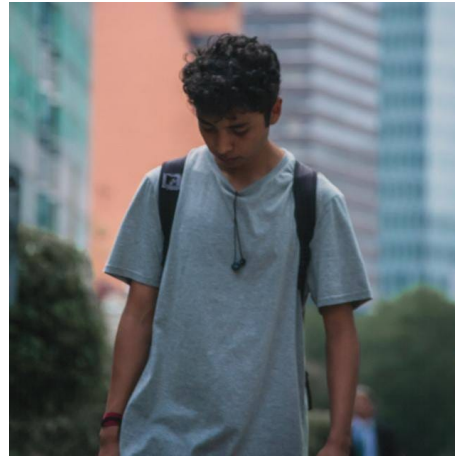
Una vez visto y analizado los algoritmos, podemos llegar a la conclusión que el algoritmo más eficiente es aquel que tuvo un orden de complejidad $O(\log n)$, dicho algoritmo fue *div2*, los otros tuvieron orden de complejidad $O(n)$, y el recursivo resulto ser el menos eficiente, por dos factores: ser de complejidad $O(n)$ y tener una capacidad limitada para realizar divisiones. Como se observó en la experimentación, este algoritmo, en su peor caso, no podía dividir números entre 1, mayores a 998, puede ser que en otras computadoras se permitieran números más grandes, pero al final siempre este algoritmo estará restringido.

4 Conclusiones

Eduardo Mendoza Martínez

Como se vió en esta práctica, es importante conocer que algoritmo puede ser mejor y saber como compararlos entre sí. Cuando se empieza a programar es común creer que menos es mejor, pero esto no ocurre en el mundo de la computación, la mayor parte del tiempo más código es mejor, puede ser más rápido y hacer una tarea con mayor eficiencia. Ejemplos que se me pueden venir a la mente, son todos los métodos matemáticos que se computan a través de algoritmos medianamente grandes, para conocer números primos, integrar, transformar regiones, etc.

Todos estos algoritmos llegan a usar muchas líneas de código dado que en una computadora es demasiado rápido hacer operaciones aritméticas básicas, mientras que algunos de esos procesos a nosotros nos pueden parecer tediosos, para una computadora no es nada y lo hace muy rápido. Claramente esto se observó en el algoritmo del cociente, se nos dieron 3 algoritmos y sorpresivamente el que tenía más líneas de código fue el que resulto ser el mejor. Así con un simple vistazo, se pudo haber dicho que la



función recursiva podría ser mejor que cualquier de las otros dos, sin embargo, no fue el caso. A través del análisis a priori y a posteriori vimos como que el que tenía más líneas de código resultó ser mejor, e inclusive el recursivo resultó ser un poco peor ya que no acepta números muy grandes por depender de la pila que se usa en funciones recursivas, esta excede el límite que se tiene. A partir de esto he llegado a esta conclusión, que muchas veces en la programación más es mejor, mientras nuestros algoritmos usen al máximo los recursos computacionales de una computadora mejor, para nosotros puede resultar tedioso seguir una serie de pasos muy larga, pero para una computadora no resulta nada complicado y nos permite realizar y calcular cosas complicadas.

Daniel Aguilar Gonzalez

El desarrollo de esta práctica me sirvió para poder comprender un poco más a fondo como es que se realiza un análisis a un algoritmo para así poder determinar orden de complejidad, tiempo de ejecución, etc. Cómo pudimos observar en la durante la práctica existen más de un algoritmo los cuales nos pueden ayudar a solucionar el mismo problema planteado, es por esto que se debe llevar a cabo un análisis a cada uno para poder determinar qué algoritmo es el más eficiente, hablando en tiempo y uso de recursos (memoria) todo esto para darnos la solución correcta, rápida y Eficaz a la que queremos llegar. Como también se pudo observar en ella un algoritmo con menos líneas de código no siempre es el más eficiente, como ejemplo de esto se mostraron 3 algoritmos los cuales calculan el producto de dos enteros positivos pero en uno de ellos se usa una función recursiva y hace que el algoritmo tenga muchas menos líneas de ejecución más sin embargo este algoritmo es el que más requiere del uso de recursos y es de los que más tiempo de ejecución requiere.



5 Anexo

5.1 Cálculo del orden de complejidad del BubbleSort

Para calcular el orden de complejidad de este algoritmo, se obtuvo lo siguiente.

Línea	Costo
C_1	n
C_2	$\sum_{j=i}^{n-1} t_j$
C_3	$\sum_{j=i}^{n-1} (t_j - 1)$
C_4	$\sum_{j=i}^{n-1} (t_j - 1)$

Así, para t_j se tiene lo siguiente.

j	t_j
1	n
2	n - 1
...	...
j	n-j+1

Por lo que nuestra $T(n)$ quedaría conformada de la siguiente forma.

$$T(n) = C_1 n + C_2 \sum_{j=i}^{n-1} t_j + (C_3 + C_4) \sum_{j=i}^{n-1} (t_j - 1)$$

$$T(n) = C_1 n + C_2 \sum_{j=i}^{n-1} (n - j + 1) + (C_3 + C_4) \sum_{j=i}^{n-1} (n - j)$$

$$T(n) = C_1 n + \frac{1}{2} C_2 (i - n - 2)(i - n - 1) + \frac{1}{2} (C_3 + C_4) (i - n - 1)(i - n)$$

i.e.,

$$T(n) = an^2 + bn + c$$

$$\therefore T(n) \in O(n^2)$$

5.2 Problemas 24 de la lista de problemas

5.2.1 Primer problema

$$x(n) = \begin{cases} x(n-1) + 5 & , n > 1 \\ 0 & , n = 1 \end{cases} \quad (3)$$

$$\begin{aligned} x(n) &= x(n-1) + 5 \\ &= x(n-2) + 5 + 5 \\ &= x(n-3) + 5 + 5 + 5 \\ &\vdots \\ &= x(n-i) + 5i \end{aligned}$$

$$\begin{aligned}
 & \cdot \\
 & \cdot \\
 & = x(1) + 5(n+1) = 5n + 5
 \end{aligned}$$

5.2.2 Segundo problema

$$x(n) = \begin{cases} 3x(n-1) & , n > 1 \\ 4 & , n = 1 \end{cases} \quad (4)$$

$$\begin{aligned}
 x(n) &= 3x(n-1) \\
 &= 3(3x(n-2)) = 3^2x(n-2) \\
 &= 3^2(3x(n-3)) = 3^3x(n-3) \\
 &\cdot \\
 &\cdot \\
 &\cdot \\
 &= 3^i x(n-i) \\
 &\cdot \\
 &\cdot \\
 &\cdot \\
 &= 3^{n+1}x(1) = 4(3^{n+1})
 \end{aligned}$$

5.2.3 Tercer problema

$$x(n) = \begin{cases} x(n/2) + n & , n > 1 \\ 1 & , n = 1 \end{cases} \quad (5)$$

Sea $n = 2^k$, con $k = \log_2 n$, se tiene

$$\begin{aligned}
 x(n) &= x(2^{k-1}) + 2^k \\
 &= x(2^{k-2}) + 2^{k-1} + 2^k \\
 &= x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\
 &\cdot \\
 &\cdot \\
 &\cdot \\
 &= x(2^{k-i-1}) + 2^{k-i} + 2^{k-i+1} + \dots + 2^{k-1} + 2^k \\
 &\cdot \\
 &\cdot \\
 &\cdot \\
 &= x(2^0) + 2^0 + 2^1 + \dots + 2^{k-1} + 2^k \\
 &= 1 + 2^{k+1} - 1 = 2(n)
 \end{aligned}$$

5.2.4 Cuarto problema

$$x(n) = \begin{cases} x(n/3) + 1 & , n > 1 \\ 1 & , n = 1 \end{cases} \quad (6)$$

Sea $n = 3^k$, con $k = \log_3 n$, se tiene

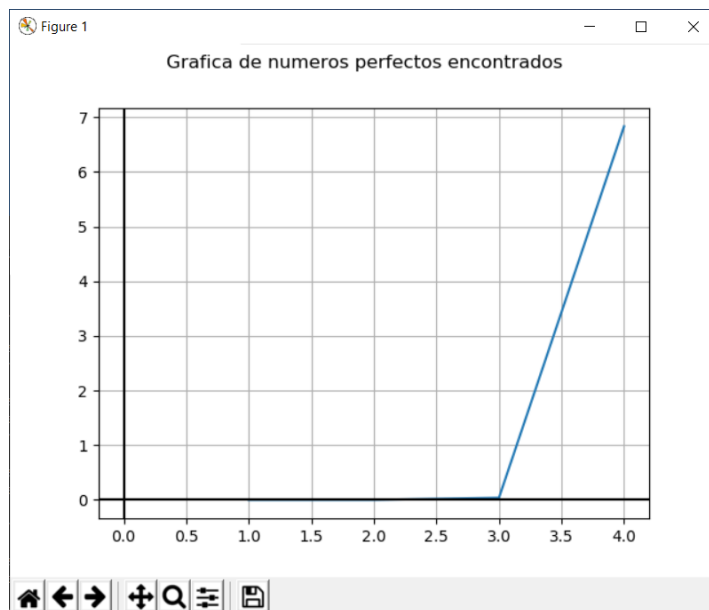
$$\begin{aligned} x(n) &= x(3^{k-1}) + 1 \\ &= x(3^{k-2}) + 1 + 1 = x(3^{k-2}) + 2 \\ &= x(3^{k-3}) + 3 \\ &\quad \cdot \\ &\quad \cdot \\ &\quad \cdot \\ &= x(3^{k-i}) + i \\ &\quad \cdot \\ &\quad \cdot \\ &\quad \cdot \\ &= x(3^0) + k = 1 + \log_3 n \end{aligned}$$

5.3 Números perfectos

El siguiente algoritmo, irá mostrando números perfectos conforme el usuario le diga, es decir el n -ésimo número perfecto será el último que encuentre dado un número d dado por el usuario.

```
encontrarNumeroPerfecto(int c)
    n = 1
    i = 0
    while i < c
        sum = 0
        divisor = 1
        while divisor < n
            if not n % divisor
                sum = sum + divisor
            divisor = divisor + 1
        if sum == n
            print(n + " es un numero perfecto")
            i = i + 1
        n = n + 1
```

Primeramente se probó con que encontrará 4 números perfectos, y el resultado de nuestra gráfica fue el siguiente.

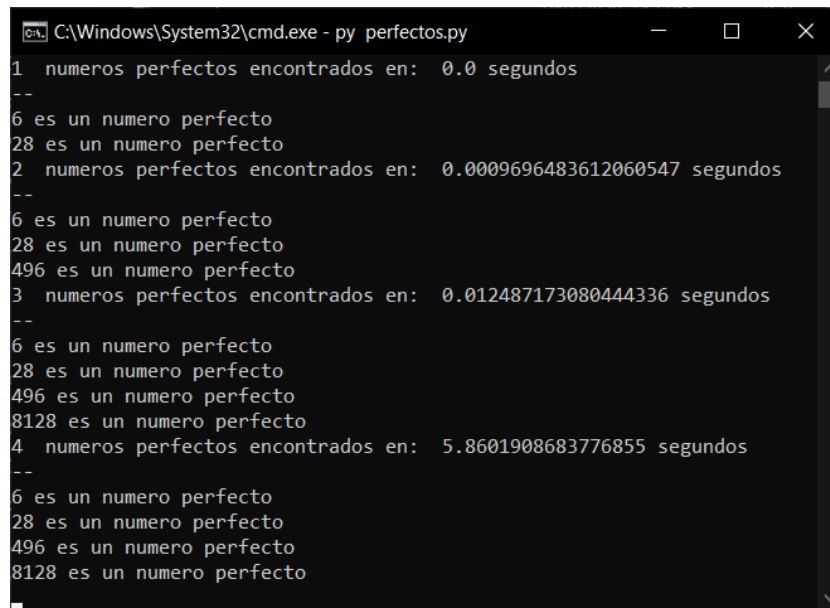


Resultado en consola

```
C:\Windows\System32\cmd.exe - py perfectos.py
C:\Users\52556\Documents\ESCOM\Análisis de algoritmos\P2>py perfectos.py
6 es un numero perfecto
1  numeros perfectos encontrados en:  0.0 segundos
--
6 es un numero perfecto
28 es un numero perfecto
2  numeros perfectos encontrados en:  0.0 segundos
--
6 es un numero perfecto
28 es un numero perfecto
496 es un numero perfecto
3  numeros perfectos encontrados en:  0.03690195083618164 segundos
--
6 es un numero perfecto
28 es un numero perfecto
496 es un numero perfecto
8128 es un numero perfecto
4  numeros perfectos encontrados en:  6.835728883743286 segundos
--
```

Como se puede estar observando, a medida que se quiera encontrar otro número perfecto, el tiempo de ejecución es más tardado.

Seguidamente, se probó con el 5to número perfecto, donde el tiempo aumento significativamente y esto se ve reflejado claramente en la gráfica y en la consola. Para el quinto número perfecto la computadora tardó mucho en calcularlo, llevó más de 2 horas y se suspendió, solo quedó trabajando pero nunca apareció ese quinto número perfecto.



```
C:\Windows\System32\cmd.exe - py perfectos.py
1 numeros perfectos encontrados en: 0.0 segundos
--
6 es un numero perfecto
28 es un numero perfecto
2 numeros perfectos encontrados en: 0.0009696483612060547 segundos
--
6 es un numero perfecto
28 es un numero perfecto
496 es un numero perfecto
3 numeros perfectos encontrados en: 0.012487173080444336 segundos
--
6 es un numero perfecto
28 es un numero perfecto
496 es un numero perfecto
8128 es un numero perfecto
4 numeros perfectos encontrados en: 5.8601908683776855 segundos
--
6 es un numero perfecto
28 es un numero perfecto
496 es un numero perfecto
8128 es un numero perfecto
```

6 Bibliografía

- [1] "Análisis de Algoritmos: Complejidad", Lab.dit.upm.es, 1997. [Online]. Disponible: <http://www.lab.dit.upm.es/lprg/material/apuntes/o/index.html>. [Acceso: 26- Feb- 2018]. [2] R. Wachenchauzer, M. Manterola, M. Curia, M. Medrano and N. Paez, Algoritmos de Programacion con Python.