Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2021-1, 3CV1, Práctica 2, 26/10/20

Práctica 2: Funciones Recursivas vs Iterativas

Valle Martínez Luis Eduardo, Rivero Ronquillo Omar Imanol

lvalle212@gmail.com, imanol.rivero7@gmail.com

Resumen: En el presente documento se presenta una comparación del análisis de los algoritmo recursivos frente a los iterativos
Palabras Clave: Algoritmo iterativos, Algoritmos Recursivos, Java.

Introducción

En el area de las ciencias de la computación se tiene una gran diversidad de algoritmos. Estos pueden ser clasificados con patrones bien definidos que los distinguen unos de otros.
Los algoritmos iterativos se caracterizan por ejecutarse mediante ciclos. Este tipo de algoritmos son muy útiles al momento de realizar tareas repetitivas. Por otro lado, los algoritmos recursivos son aquellos que buscan resolver un problema sustituyéndolo por otros problemas de la misma categoria, pero más simples. Se dice que un algoritmo es recursivo si dentro del cuerpo del algoritmo y de forma directa o indirecta se realiza una llamada a sí mismo.
Para esta práctica se desarrolla un analisis de complejidad de los algoritmos propuestos, comprobando de forma analitica y de forma experimental la eficiencia de los algorimos en cuestión.

Conceptos Básicos

Método de sustitución hacia atrás
Se realizará el análisis de 3 algoritmos diferentes, que toman enfoques distintos para otorgar el mismo resultado. La primer tercia, realiza el producto de 2 números enteros:

Producto de 2 enteros

Los 3 algoritmos reciben como argumento a los 2 enteros y regresan el valor del producto también como un entero.

Prod1

De naturaleza iterativa, el algoritmo **prod1** realiza n veces la suma de m mediante un bucle while que por cada iteración disminuirá en la unidad el valor de n hasta que sea 0.

```
int prod1(m,n)
    r=0
    while n>0
        r = r+m
        n--
    return r
```

- **¿Cuándo se obtiene el mejor caso?**
  Este caso lo encontramos cuando $n \leq 0$ ya que solo verifica la condición del bloque while una sola vez sin ejecutar su contenido.
- **¿Cuándo se obtiene el peor caso?** Para este algoritmo, no se encuentra un peor caso, pero un caso general que se obtiene cuando $n > 0$ incrementando los pasos que tiene que ejecutar creciendo de forma lineal según sea el valor de n.
- **Cálculo analítico de orden de complejidad**

```
int prod1(m,n)
    r=0             C1    |   1
    while n>0       C2    |   n+1
        r = r+m     C3    |   n
        n--         C4    |   n
    return r        C5    |   1
```

Luego,

$$T(n) = C_1 + C_2(n+1) + C_3 n + C_4 n + C_5 = C_1 + C_2 n + C_2 + C_3 n + C_4 n + C_5 = (C_2 + C_3 + C_4)n + (C_1 + C_2 + C_5)$$

Entonces, nuestra $T(n)$ es de la forma $an+b$. Por lo que el mejor caso tendrá una complejidad constante: $T(n) \in \Omega(n)$
Para el caso general, tendremos la complejidad lineal: $T(n) \in \Theta(n)$

Prod2

Otro algoritmo de naturaleza iterativa. Funciona dentro de un ciclo while que se ejecuta mientras n sea mayor a 0. Para lograr que el algoritmo funcione, es necesario ocupar únicamente la parte entera de las operaciones que requieren división.

Se va a multiplicar el valor de m por una potencia de 2, que dependerá del número de la iteración dentro del ciclo. De esta forma, el valor de m:
Para la primer iteración

$$m = 2^1 * m$$

Para la segunda iteración

$$m = 2^2 * m$$

...
Entonces para la k iteración

$$m = 2^k * m$$

Donde k se defina a partir de n:

$$\frac{n}{2^k} = 1$$

$$2^k = n$$
$$klog(2) = log(n)$$
$$k = log(n)$$

Mientras n será decrementada al ser dividida por un potencia de 2 en cada iteración.

Para obtener el producto resultante, se realiza la condición mediante una comparación a nivel bit de n con 1 ($n\&1$), resultando cierta la condición cuando el valor de n es impar, así el valor que conserva m de la iteración pasada se irá sumando a la variable auxiliar r.

```
int prod2(m,n)
    r=0
    while n>0
        if n & 1
            r = r+m
        m = 2*m
        n = n/2
    return r
```

- **¿Cuándo se obtiene el mejor caso?**
  Corresponde al caso en que $n \leq 0$, ya que en este caso tampoco ejecutaria el contenido del while, devolviendo directamente $r = 0$.
- **¿Cuándo se obtiene el peor caso?**
  El peor caso puede ser considerado como aquel que es el general para este alogoritmo en particular.Se obtendrá el caso general cuando el valor de $n > 0$
- **Cálculo analítico de orden de complejidad**

```
int prod2(m,n)
    r=0                 C1 | 1
    while n>0           C2 | k = log(n)+1
        if n & 1        C3 | log(n)
            r = r+m     C4 | constante
        m = 2*m         C5 | log(n)
        n = n/2         C6 | log(n)
    return r            C7 | 1
```

Luego,

$T(n) = C_1 + C_2(log(n) + 1) + C_3log(n) + C_4 + C_5log(n) + C_6log(n) + C_7 = Clog(n) + C$

De esta forma obtenemos para el mejor caso un complejidad constante: $T(n) \in \Omega(1)$
Y en el caso promedio una complejidad logarítmica: $T(n) \in \Theta(log(n))$

Prod3

El último de los algoritmos para obtener el producto de 2 números enteros, será recursivo.

La lógica detrás de este algoritmo es igual a la de **prod1**, donde se irá sumando b veces el número a.
El caso base para esta recursión es cuando **b** es igual a 1, por lo que solamente se regresa el valor de a. En caso contrario, al valor de a se le deberá sumar el valor de la función con *b-1*.

```
int prod3(a,b)
    if b =1
        return a
    else
        return a+prod3(a,b-1)
```

- **¿Cuándo se obtiene el mejor caso?**
  Le corresponde al caso en que nuestra variable de entrada a la primera iteración es $b = 1$.

- **¿Cuándo se obtiene el peor caso?**
  Este se obtiene cuando encontramos desde la primera iteración un $b > a$ que aumenta las veces que vuelve a ejecutar el algoritmo recursivo sin importar si desde el inicio la primera iteración tiene $a = 0$. Además no se considera el caso en que directamente $b = 0$.

- **Cálculo analítico de orden de complejidad** Del algoritmo obtenemos la condición base y el caso general:

$$prod3(a,b) = \begin{cases} a & si \quad b = 1 \\ a + prod3(a, b-1) & si \quad otro \end{cases}$$

  Sea $M(n)$ el número de sumas que realiza $prod3(a,b)$ para calcular el producto final. Entonces:

$$M(n) = M(n-1) + 1$$

  Ya que se requieren $M(n-1)$ sumas para calcular $prod3(a, b-1)$ y una más para sumar con a. Además $M(0) = a$ ya que no se ejecuta ninguna suma. Por lo tanto nuestra ecuación de recurrencia para este algoritmo, se define como:

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

  Resolviendo mediante el método de sustitución hacia atrás. Se tiene:

Si $n > 0$	Para i, $= M(n-i) + i$
$M(n) = M(n-1) + 1$	...
$= M(n-2) + 2$	...
$= M(n-3) + 3$	Nos detenemos cuando $n - i = 0$, entonces $n = i$
...	$= M(0) + n$
...	$= n$
...	$\Rightarrow M(n) \in \Theta(n)$

Cociente de 2 enteros

Se tiene ahora, una tercia de algoritmos que calculan el cociente de 2 números enteros.

Div1

El primer algoritmo, será una solución iterativa al problema. Se aceptan ahora 3 argumentos de la función, el primero será el entero dividendo(n), el segundo el divisor(div) y el último será un apuntador para poder obtener el residuo(r).

Este parte del principio que la división es la operación inversa de la multiplicación, por lo que ahora en vez de realizar sumas, se debera restar.

Se ejecuta dentro de un bucle while, que iterará hasta que el valor de n sea menor al de div(divisor). En cada iteración, se restara 1 vez div de n, y al cociente, representado por q, se le sumará la unidad.

Al final del ciclo, al apuntador r se le asigna el valor de n, es decir, el valor restante de la división o residuo.

```
int div1(n,div,*r)
    q = 0
    while n>=div
        n=n-div
        q++
    *r = n
    return q
```

- **¿Cuándo se obtiene el mejor caso?**
  Se distingue como mejor caso, a las entradas donde el divisor *div* será mayor que el dividendo n, evitando la ejecución de las lineas dentro del while,tan solo se realizará la comprobación del ciclo while en una ocasión.

- **¿Cuándo se obtiene el peor caso?**
  El peor caso será cuando el divisor *div*, sea ingresado con el valor de 1.

- **Cálculo analítico de orden de complejidad**
  Ahora se realizará el cálculo de complejidad considerando el orden de complejidad por linea.

```
int  div1(n,div,*r)
     q  =  0              O(1)
     while  n>=div        O(n)
          n=n-div         O(1)
          q++             O(1)
     *r  =  n             O(1)
return  q                 O(1)
```

  Se observa que el número de veces que se ejecuta el bucle while corresponde con el valor de q, más no nos es útil la variable a retornar para analizar la ejecución del bucle. Se plantea la siguiente ecuación donde el valor derecho de la igualdad debería llegar a ser menor al valor de *div*:

$$div = n - kdiv$$

$$div - n = -kdiv$$

$$k = \frac{n - div}{div}$$

  Donde será k el número de iteraciones que se realizan hasta que n tome el valor de *div*, y para poder terminar la ejecución se le suma 1 a k, quedándo finalmente como:

$$k = \frac{n - div}{div} + 1$$

  Para el peor caso se va considerar el valor de *div* como 1, por tal razón vamos a obtener una complejidad $O(n)$. Ésta será la complejidad máxima de las líneas conformando a la función, de tal manera que finalmente $T(n) \in O(n)$.

Div2

El segundo algoritmo para obtener el cociente también será recursivo y tendrá los mismos argumento de función que el primero.

Este en cambio, iniciará asignando de entrada el residuo con el valor de n, y a una variable auxilir *dd*, que tomará el valor de *div*.

Se inicia el primer ciclo de iteraciones que cambiará el valor de *dd* al multiplicarlo por una potencia de 2 $dd = 2^j dd$, se apunta que el valor de *dd* se obtiene del argumento de función *div*, permitiéndonos sustituir la en la ecuación cuando el valor sea igual a n: $2^j * div = n$ $j = log(\frac{n}{div})$ En la ecuación, j será el número de iteraciones, y a este resultado, se le sumará 1 por la última iteración que se realiza cuando *dd* sigue siendo menor o igual a n, pero que al final del bucle lo volverá mayor a n.

En el segundo bucle se inicia el procedimiento para el cálculo del cociente. Este bucle iterará mientras el valor calculado anteriormente de *dd* sea mayor al del divisor(*div*). Para esto, *dd* será dividido por una potencia de 2, como proceso inverso,

y q también será multiplicado por la misma potencia de 2, pero su valor ira cambiando según el condicionamiento dentro del bucle.

La condición restará el valor de dd al residuo r, y se aumenta en la unidad el cociente.

Finalmente, el número de iteraciones que realiza, al ser un proceso inverso, en lo que respecta al cambio de los valores de dd, seguirá siendo $j = log(\frac{n}{div})$

```
int div2(n,div,*r)
    dd = div
    q = 0
    *r = n
    while dd<=n
        dd = 2*dd
    while dd>div
        dd = dd/2
        q = 2*q
        if dd <= *r
            *r = *r-dd
            q++
    return q
```

- **¿Cuándo se obtiene el mejor caso?**
  El mejor caso, al igual que con *div1*, será cuando el divisor div, es mayor al dividendo n.
- **¿Cuándo se obtiene el peor caso?**
  El peor caso para este algoritmo lo encontramos cuando $div = 1$. con esta configuración de valores, se ejecutaria más veces las restas para cualquier valor de n.
- **Cálculo analítico de orden de complejidad**
  Para el cálculo de la complejidad de este algoritmo, consideramos las ecuaciones explicadas anteriormente, y rescatamos la ecuación $j = log(\frac{n}{div})$ que expone la complejidad del primer bucle, se transcribe de la siguiente manera: $\Theta(log(\frac{n}{div}))$, pero la complejidad es una función que depende de la variable n y div se considera constante, quedándonos simplemente:

$$\Theta(log(n))$$

`dd = div`	$\mathcal{O}(1)$
`q = 0`	$\mathcal{O}(1)$
`*r = n`	$\mathcal{O}(1)$
`while dd<=n`	$\mathcal{O}(log(n))$
`    dd = 2*dd`	$\mathcal{O}(1)$
`while dd>div`	$\mathcal{O}(log(n))$
`    dd = dd/2`	$\mathcal{O}(1)$
`    q = 2*q`	$\mathcal{O}(1)$
`    if dd <= *r`	$\mathcal{O}(1)$
`        *r = *r-dd`	$\mathcal{O}(1)$
`        q++`	$\mathcal{O}(1)$
`return q`	$\mathcal{O}(1)$

Siendo $\Theta(log(n))$ la complejidad mayor, la función tendrá complejidad $T(n) \in \Theta(log(n))$

Div3

El último algoritmo de la tercia será el de naturaleza recursiva. Este algoritmo también aplicará la lógica de **div1**, donde se suman las veces que div puede restársele a n.

A diferencia de los algoritmo anteriores, este solo dispone de 2 argumentos de función, n o dividendo, y div o divisor, por lo que no se conoce el residuo al terminar la función, pero que facilmente podría incorporársele para conocerlo.

El caso base para esta recursión es cuando el valor del divisor es mayor que n, por lo que el valor actual de n será ya el residuo y por lo tanto no se le suma la unidad al cociente, más se regresa el valor 0. En caso contrario, se sumará la unidad al resultado de la misma función para los valores del dividendo menos el divisor: $div3(n - div, div)$.

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

- **¿Cuándo se obtiene el mejor caso?**
  El mejor caso lo encontramos cuando el divisor *div* es mayor a *n*. Ya que evitamos el mayor numero de lineas con este valor como entrada.

- **¿Cuándo se obtiene el peor caso?**
  Al igual que en los algoritmo anteriores, el peor caso para este algoritmo será cuando el divisor es igual a 1, asegurando para cualquier valor de n el mayor número de restas posible.

- **Cálculo analítico de orden de complejidad** Del algoritmo obtenemos la condición base y el caso general:

$$div3(n, div) = \begin{cases} 0 & si \quad div > n \\ 1 + div3(n - div, div) & si \quad otro \end{cases}$$

  Sea $M(n)$ el número de sumas que realiza $div3(a, b)$ para calcular el cociente final. Entonces:

$$M(n) = M(n - div) + 1$$

  Ya que se requieren $M(n - div)$ sumas, con div constante, para calcular $div3(n - div, div)$ y una más para sumar con 1. Además $M(0) = 1$ ya que no se ejecuta ninguna suma. Por lo tanto nuestra ecuación de recurrencia para este algoritmo, se define como:

$$M(n) = \begin{cases} 1 & si \quad n < div \\ M(n-1) + 1 & si \quad otro \end{cases}$$

  Ahora, considerando para $n > 1$
  $M(n) = M(n-1) + 1 + \mathcal{O}(1)$
  $M(n) = M(n-1) + \mathcal{O}(1)$
  $M(n) = M(n-1) + C$, siendo c una constante.

  Resolviendo mediante el método de sustitución hacia atrás. Se tiene:

  $M(n) = M(n-1) + C$
  $= M(n-2) + 2C$
  $= M(n-3) + 3C$
  ...
  ...
  ...
  Para i, $= M(n-i) + iC$

  ...
  ...
  Nos detenemos cuando $n - i = 0$, entonces $n = i$
  $= M(0) + nC = C + nC$
  Por lo tanto, $M(n) \in \mathcal{O}(n)$

Experimentación y Resultados

Producto 3 enteros

A continuación se muestran las gráficas generadas de los pares ordenadas que corresponden al valor de la n, eje de las abscisas, y el tiempo de procesamiento t en μ seg, para cada uno de los 3 algoritmos del producto de 2 números enteros.

Gráficas de tamaño vs tiempo

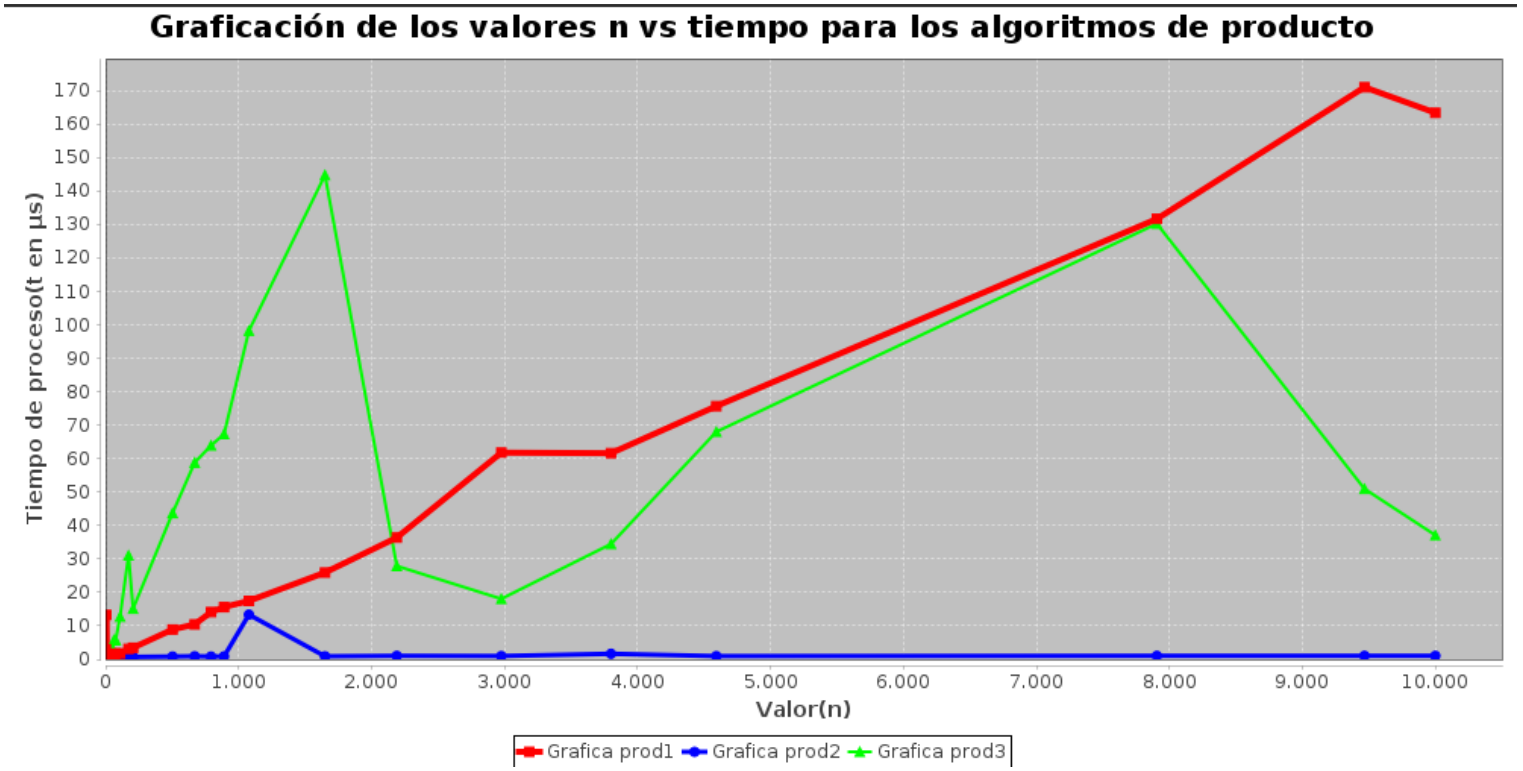

Figura 1: Gráficas de los puntos ordenados obtenidos del registro de tiempo de procesamiento para una n incremental en cada algoritmo de producto de números enteros

Análisis Asintótico

Ahora realizamos la comparación de las complejidades obtenidas *a priori* en la sección anterior, y el análisis *a posteriori* observable con las gráficas.

Primer algoritmo

El primer algoritmo se muestra en la gráfica como una curva de color rojo. Es facilmente identificable que es una recta, representando una complejidad lineal para el primer algoritmo.
Se recupera la complejidad calculada de forma analítica para el primer algoritmo de producto, y se muestran las complejidades según el método en la siguiente tabla:

Método análisis	Complejidad
A priori	O(n)
A posteriori	$\Theta(n)$

De forma unánime, es correcto concluir que para el primer algoritmo, de naturaleza iterativa: $prod1 \in \Theta(n)$

Segundo algoritmo

El segundo algoritmo tiene de curva representativa una de color azul. Recurriendo al análisis visual de esta curva y realizando la comparación que las otras 2 en el plano, erroneamente puede sugerirse que la recta no creciente, asemeja al comportamiento de una función igualada a una constante.
Se recupera una vez más, la complejidad calculada de forma analítica para el segundo algoritmo de producto, y se muestran las complejidades según el método en la siguiente tabla:

Método análisis	Complejidad
A priori	O(1)
A posteriori	$\Theta(log(n))$

Con la base del análisis teórico de la complejidad del algoritmo, es seguro relacionar la lentitud de crecimiento de la curva para valores ascendentes de n, a las propiedades de una curva descrita por una función logarítmica, como lo sugiere la complejidad *a priori*. De esta forma se concluye que: $prod2 \in \Theta(log(n))$

Tercer algoritmo

El tercer algoritmo es la única solución de forma iterativa, su gráfica se mostrará de un color verde y es fácilmente identificable por su irregularidad. El análisis de esta curva es más complicado con el método de observación *a posteriori* por su irregularidad. Da la impresión de mostrar 2 comportamientos diferentes, el primero creciendo parecido a una función exponencial o cuadrada, más sin embargo, a medida que crece el valor de n, podemos ver que corrige este comportamiento pareciendose más a una curva que crece de forma lineal.
Para concluir una sola complejidad de este algoritmo, se opta por describir el comportamiento de la curva como una recta. Dado que este último se da cuando los valores son mayores para n, se intuye que para valores mayores seguirá con esta tendencia.
Finalmente se muestra la tabla comparando las 2 complejidades obtenidas de los 2 métodos:

Método análisis	Complejidad
A priori	O(n)
A posteriori	$\Theta(n)$

Apoyando la complejidad convencida para el análisis *a posteriori*, la complejidad analítica nos afirma que su complejidad será lineal. De esta forma concluimos que: $prod3 \in \Theta(n)$

Eficiencia de los algoritmo

Determinada la complejidad de los 3 algoritmos, se puede señalar la eficiencia de estos en comparación entre ellos mismos.
Salta a la vista gracias a la cruva gráficada, y respaldada por su análisis analítico, que el segundo algoritmo **prod2**, es el **más eficiente** de la tercia, probando un crecimiento de tiempo de procesamiento muy lento para valores que seguiran creciendo de n.
Nos queda entonces, indicar cual de los 2 restantes es el menos eficiente. Los resultados analíticos nos ofrecen una respuesta sencilla: "Los 2 se acotan por el mismo conjunto de ecuaciones, donde solo variará la constante que más concretamente los delimitará", más la representación gráfica no ofrece un resultado completamente claro, pero se realiza una interpretación de este y se concluye que, con una tendecia clara y pendiente mayor a la del algoritmo recursivo, el **prod2** iterativo se puede considerar el **menos eficiente** de entre los analizados.

Cociente de 2 enteros

Finalmente, se muestran las gráficas generadas de los pares ordenadas que corresponden al valor de la n, eje de las abscisas, y el tiempo de procesamiento t en μ seg, para cada uno de los 3 algoritmos del cociente de 2 números enteros.

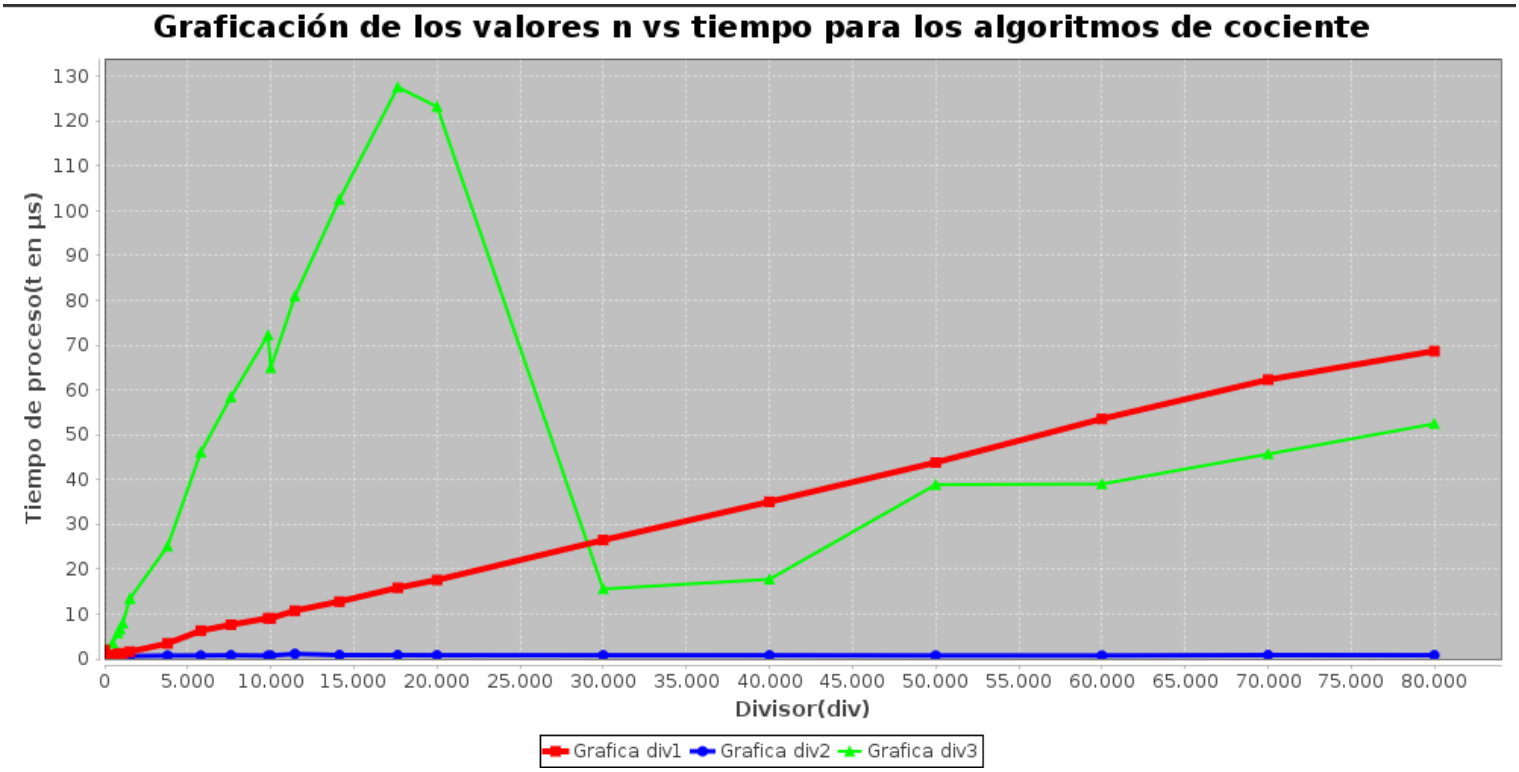

Figura 2: Gráficas de los puntos ordenados obtenidos del registro de tiempo de procesamiento para una n incremental en cada algoritmo de cociente de números enteros

Gráficas de tamaño vs tiempo

Análisis Asintótico

Ahora realizamos la comparación de las complejidades obtenidas *a priori* en la sección anterior, y el análisis *a posteriori* observable con las gráficas.

Primer algoritmo

El primer algoritmo se muestra en la gráfica como una curva de color rojo, al igual que el algoritmo del producto. La tendencia de comportamiento es clara y consistente con los valores evaluados, se define su comportamiento por una recta. Se toma la complejidad calculada analíticamente en el análisis *a priori* para este algoritmo, y se muestran las complejidades comparadas en la siguiente tabla:

Método análisis	Complejidad
A priori	O(n)
A posteriori	$O(n)$

Claramente su crecimiento es lineal y se concluye que: $div1 \in \Theta(n)$

Segundo algoritmo

El segundo algoritmo, de naturaleza iterativa, se representa con una curva de color azul. Una vez más, nos enfrentamos a una representación engañosa de la que necesitaremos el análisis *a priori* para no afirmar erróneamente que su descripción es la de una constante.
Se recupera una vez más, la complejidad calculada de forma analítica para el segundo algoritmo de divisiones, y se muestran las complejidades según el método en la siguiente tabla:

Método análisis	Complejidad
A priori	O(1)
A posteriori	$O(log(n))$

Advertido por el resultado obtenido para su algorimo inverso de los productos, su descripción no será la de una constante, pero la de curva de crecimiento algorítmico: $div2 \in \Theta(log(n))$

Tercer algoritmo

El último algoritmo se plantea de forma recursiva, su gráfica es la de un color verde y que también muestra un comportamiento errático como el algoritmo también recursivo para los productos. Tomando directamente en cuenta solo la tendencia de la gráfica para los valores mayores de n, podría sugerirse que se llega a comportar con un crecimiento lineal. Se eliminan las opciones que podría describir su comportamiento por debajo del de *div1*, que no llegaría a superar para valores mayores; Su comportamiento no podría ligarse al de una constante, ni a la de una función logarítmica ni de una raíz, por lo que nos queda otorgarle las cualidades de una función descrita por una recta.
Finalmente se muestra la tabla comparando las 2 complejidades obtenidas de los 2 métodos:

Método análisis	Complejidad
A priori	O(n)
A posteriori	$O(n)$

Concluimos finalmente entonces, que para este algoritmo recursivo, su complejidad es: $div3 \in \Theta(n)$

Eficiencia de los algoritmos

Rápidamente determinanos el algoritmo más eficiente del conjunto utilizado. El algoritmo de *div2*, tiene complejidad $\Theta(log(n))$, siendo la menor de entre los 3. Los 2 algoritmos restantes, una vez más tendrán la misma complejidad acotada por $\Theta(n)$, donde solo el manejo de las constantes nos determina, de la familia de rectas, cuál es la que los acota más justamente.
Se recurre a la gráfica para comprobar las tendencias de ambos logaritmos, y se identifica, aún con el comportamiento errático de su contrincante, que la gráfica de *div1* es la que seguirá creciendo con una pendiente mayor resultando en mayor tiempo de procesamiento y por lo tanto **menos eficientemente**.

Conclusiones

Rivero Ronquillo Omar Imanol

Al iniciar el análisis de algoritmos iterativos era sencillo concluir que aquellos que parecían tener una menor cantidad de lineas de código eran aquellos que potencialmente tenían una menor complejidad algorítmica, sin embargo, cuando se realizaba un análisis a profundidad (que probablemente no sea necesario para alguien con más experiencia en el cálculo de complejidades), no siempre era aquello que representaba inicialmente.

Resulta particularmente interesante aplicar un análisis *a priori* y *a posteriori* en algoritmos que finalmente están diseñados para resolver el mismo problema, sin embargo, gracias al análisis pudimos comprobar de manera teórica y posteriormente, de forma experimental, la diferencia fundamental entre estos algoritmos: su complejidad. Para aquellos que su complejidad era mayor era claro experimentar un consumo de tiempo de ejecución mas elevado, escalando progresivamente cuando se realizaban pruebas para valores muy grandes o para aquellos que, por su propia definición, le provocan caer en su peor caso al algoritmo.

Para estos algoritmos, una vez más queda claro que no hay que dejar engañarnos por las primeras impresiones. Aunque los algoritmos recursivos *prod3* y *div3*, considerablemente tenian una menor cantidad de líneas de código que sus contrapartes iterativas, en ningún caso probaron ser mejores. Aún así para algunos casos como los de *prod2()* y *div2*, fue necesaria la comparación de sus resultados experimentales con sus respectivos resultados teóricos, provocados por una representación engañosa.

Finalmente, me queda comentar que aunque un algoritmo recursivo sea mucho más simple de definir, no siempre será la mejor solución posible del problema.

Valle Martínez Luis Eduardo

En la presente práctica se desarrollaron implementaciones a un mismo problema a través de ópticas algoritmicas distintas. Se introdujo la recursividad como otro de los recursos disponibles para solucionar problemas, con códigos más reducidos y que en algunos casos, ofrecen métodos más sofisticados y elegantes.

Pero por tener una cantidad menor de líneas de codficación, esto implica directamente una disminución en la complejidad en su ejecución. Por esta razón se analizaron, utilizando ambos métodos, *a priori* y *a posteriori*, la eficacia de una alternativa recursiva, frente a la clásica solución iterativa.

Se comprobaron además, las ventajas y limitaciones de un método con perspectiva del otro, presentando también que en conjunto permiten clarificar las dudas posibles existentes y complementarse para hallar una respuesta correcta. Mientras el analítico requiere una completa y clara comprensión del funcionamiento y flujo del algoritmo para identificar la complejidad de cada bloque que lo integra, en el método experimental, si bien esta ímplicita la comprensión del funcionamiento del algoritmo, no es una limitante para sugerir, a través de la comparación de la tendencia que muestran los datos recopilados con el comportamiento de crecimiento de los diferentes tipos de funciones, la complejidad de este.

En el proceso de desarrollo no nos encontramos en fallas o errores que perjudicaran significativamente el avance en su elaboración, sin embargo, origen de un razonamiento poco minucioso, se habían generado originalmente, valores de prueba para la generación de las gráficas que no cumplían con su objetivo, mostrar el comportamiento cuando el valor n crecía gradualmente. Este error fue facilmente corregido manteniendo el otro parámetro de ingreso a la función constante, y variando únicamente el que correspondía al parámetro analizado para su recolección.

Los resultados que se esperaban en la sección experimental, se basaban en los obtenidos de teóricamente, de manera que el error explicado antes fue identificado de esta forma. Sin embargo al ser corregido, en su mayor parte, salvo algunos fragmentos con comportamiento errático que atribuimos a procesos internos del lenguaje y el sistema pero que no son directamente casuados por el algoritmo analizado, fueron congruentes con las expectativas generadas del análisis.

Bibliografía

[1] Luna,B.[Benjamín Luna].(2020, Octubre, 6). CLASE 7 - ANÁLISIS DE ALGORITMOS[Archivo de video]. Recuperado de https://www.youtube.com/watch?v=XFSsIYKG4IE