



Instituto Politécnico Nacional
Escuela Superior de Cómputo

PRÁCTICA 3: Funciones Recursivas vs. Iterativas

Análisis de algoritmos
3CV12

Reyes Rodríguez Enrique Abdiel
Chávez Hernández Juan Diego

abykings1@gmail.com
jdiegohdez0233@gmail.com

Octubre 2021

Resumen: En el siguiente trabajo se presenta el análisis a priori y posteriori de 2 ejercicios; donde el primero se trata sobre 3 métodos para obtener el cociente de una división y el segundo sobre la búsqueda ternaria con soluciones iterativas y recursivas.

Palabras clave: Algoritmo, complejidad, iteración , recursión, búsqueda ternaria, búsqueda binaria

1. Introducción

La complejidad temporal describe la cantidad de tiempo que lleva ejecutar un algoritmo. La complejidad temporal se estima comúnmente contando el número de instrucciones que ejecuta el algoritmo, también se estima mediante bloques de código.

Como objetivo de la práctica está el demostrar que el análisis de la complejidad temporal de un algoritmo, frente a diferentes tipos de metodos de solución, es decir, iterativos contra recursivos.[1]

2. Conceptos básicos

La **búsqueda ternaria** es un método ya conocido para encontrar elementos en arrays. Este se tiene cierto parecido en la búsqueda binaria, pero su complejidad es diferente. Siendo uno de los algoritmos más útiles cuando se intenta buscar un elemento dentro de una lista, array o vector. La única condición es que sus elementos deben estar previamente ordenados.[2]

La **recursividad** a un proceso mediante el que una función se llama a sí misma, hasta que se cumple con alguna condición. Se usa para generar nuevas soluciones a problemas iterativos, así mismo, provee de soluciones más cortas y elegantes, siendo su limitante, la pila de llamadas en ejecución. [3]

La **iteración** es el acto de repetir , con el propósito de generar resultado y con el objetivo de acercarse a una meta En el contexto de las matemáticas o la informática, la iteración es un bloque de construcción estándar de algoritmos.. [4]

Pseudocódigo del algoritmo División 1

```
func division1(n,div)
    q = 0
    while n>=div
        n=n-div
        q=q+1
```

Pseudocódigo del algoritmo División 2

```
func division2(n,div)
    q = 0
    dd=div
    r=n
    while dd<=n
        dd=2*dd
    while dd>div
        dd=dd/2
        q=2*q
    if dd<=r
        r=r-dd
        q=q+1
```

Pseudocódigo del algoritmo División 3

```
func division3(n,div)
    if div>n
        return 0
    else
        return 1 + division3(n-div,div)
```

Pseudocódigo del algoritmo de búsqueda ternaria iterativa

```
func busqueda(a[],left,right,x)
    while left <= right
        mid_a = left + ((right - left) / 3)
        mid_b = right - ((right - left) / 3)
        if (a[mid_a] == x)
            return mid_a
        if (a[mid_b] == x)
            return mid_b
        if (x < a[mid_a])
            right = mid_a - 1
        else if (x > a[mid_b])
            left = mid_b + 1
```

```

else
    left = mid_a + 1
    right = mid_a - 1
return - 1

```

Pseudocódigo del algoritmo de búsqueda ternaria recursiva

```

func busqueda(a[],left,right,x)
    if (left > right)
        return -1
    mid_a = left + ((right - left) / 3)
    mid_b = right - ((right - left) / 3)
    if (a[mid_a] == x)
        return mid_a
    if (a[mid_b] == x)
        return mid_b
    if (x < a[mid_a])
        return recursiveTernarySearch(a, left, (mid_a - 1), x)
    else if (x > a[mid_b])
        return recursiveTernarySearch(a, (mid_b + 1), right, x)
    else
        return recursiveTernarySearch(a,(mid_a + 1), (mid_b - 1), x)

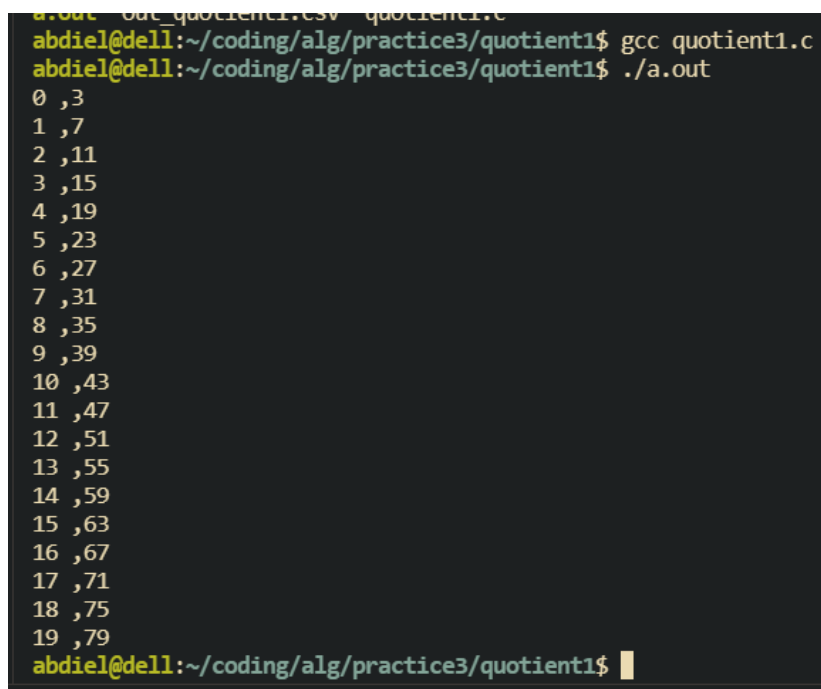
```

3. Experimentación y resultados

3.1. Algoritmo de división 1

En este caso, se experimentó con una solución iterativa, donde se observa que tiene un orden lineal dado el rango del único while que contiene, ya que en el cuerpo de este, solo hay instrucciones de orden constante.

Vemos que para este caso, se tiene que $T(n) \in O(n)$



```
abdiel@dell:~/coding/alg/practice3/quotient1$ gcc quotient1.c
abdiel@dell:~/coding/alg/practice3/quotient1$ ./a.out
0 ,3
1 ,7
2 ,11
3 ,15
4 ,19
5 ,23
6 ,27
7 ,31
8 ,35
9 ,39
10 ,43
11 ,47
12 ,51
13 ,55
14 ,59
15 ,63
16 ,67
17 ,71
18 ,75
19 ,79
abdiel@dell:~/coding/alg/practice3/quotient1$
```

Figura 1: Esta es la compilacion y ejecucion de codigo en consola

n	c
0	3
1	7
2	11
3	15
4	19
5	23
6	27
7	31
8	35
9	39
10	43
11	47
12	51
13	55
14	59
15	63

Cuadro 1: Estos son los resultados de la ejecución del programa. En la primer columna, tenemos el numero con el que se ingreso al la función `quotient()`, va del 0 al 15. En la primera columna tenemos la entrada. En la segunda columna tenemos el valor del contador, que muestra el numero de instrucciones ejecutadas. Vemos como el crecimiento es constante

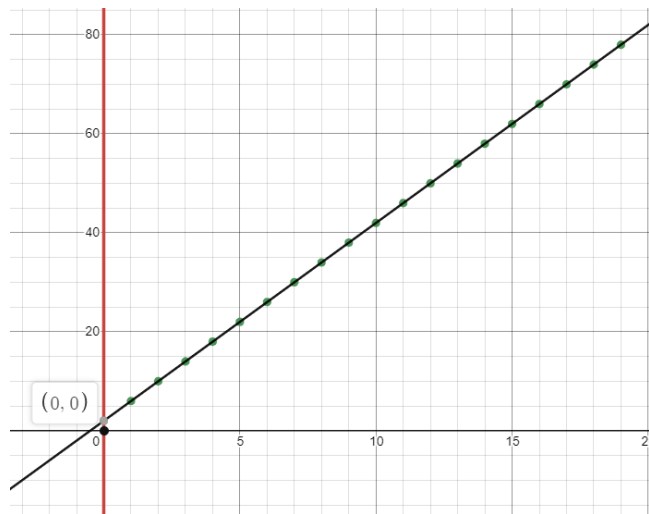


Figura 2: Esta es la gráfica que obtenemos de la tabla anterior. Los puntos son las coordenadas dadas por la tabla, la recta negra es la función $y = 4x + 2$, y n_0 está en el punto $(0, 0)$

3.2. Algoritmo de división 2

En este caso, se experimentó con una solución iterativa, donde se observa que tiene un orden $\log n$ dado el rango de los while que contiene, ya que en el cuerpo de estos, si se comportan diferente dado el incremento que tiene sus variables, crecen más rápido.

Vemos que para este caso, se tiene que $T(n) \in O(\log n)$

```
abdiel@dell:~/coding/alg/practice3/quotient2$ gcc quotient2.c
abdiel@dell:~/coding/alg/practice3/quotient2$ ./a.out
0 ,7
2 ,20
4 ,28
2 ,33
8 ,36
2 ,41
4 ,41
2 ,46
16 ,44
2 ,49
4 ,49
2 ,54
8 ,49
2 ,54
4 ,54
2 ,59
32 ,52
2 ,57
4 ,57
2 ,62
```

Figura 3: Esta es la compilacion y ejecucion de codigo en consola

n	c
0	7
1	20
2	28
3	33
4	36
5	41
6	41
7	46
8	44
9	49
10	49
11	54
12	49
13	54
14	54
15	59
16	52
17	57
18	57
19	62

Cuadro 2: Estos son los resultados de la ejecución del programa. En la primer columna, tenemos el numero con el que se ingreso al la función `quotient()`, va del 0 al 19. En la segunda columna tenemos el valor del contador, que muestra el numero de instrucciones ejecutadas. Vemos como el no es constante, sino que va decreciendo su incremento.

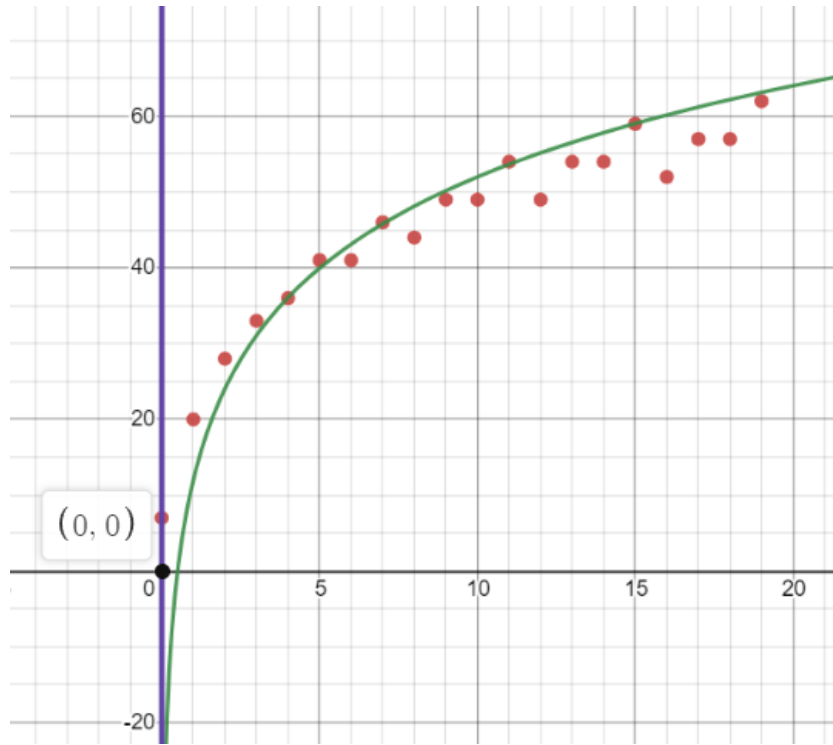


Figura 4: Esta es la gráfica que obtenemos de la tabla anterior. Los puntos son las coordenadas dadas por la tabla, la curva verde es la función $y = 40 \log x + 12$ obtenida jugando con la ecuación de la recta y el logaritmo. n_0 está en el punto $(0,0)$

3.3. Algoritmo de división 3

En este caso, el problema se solucionó usando recursión, siendo una manera más corta y elegante de escribir código, teniendo un comportamiento lineal, que es no tan común cuando se dan soluciones recursivas.

Vemos que para este caso, se tiene que $T(n) \in O(n)$

```
abdiel@del:~/coding/alg/practice3/quotient3$ gcc quotient3.c
abdiel@del:~/coding/alg/practice3/quotient3$ ./a.out
0 ,2
1 ,3
2 ,4
3 ,5
4 ,6
5 ,7
6 ,8
7 ,9
8 ,10
9 ,11
10 ,12
11 ,13
12 ,14
13 ,15
14 ,16
15 ,17
16 ,18
17 ,19
18 ,20
19 ,21
abdiel@del:~/coding/alg/practice3/quotient3$
```

Figura 5: Esta es la compilacion y ejecucion de codigo en consola

n	c
0	2
1	3
2	4
3	5
4	6
5	7
6	8
7	9
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17
16	18
17	19
18	20
19	21

Cuadro 3: Estos son los resultados de la ejecución del programa. En la primer columna, tenemos el numero con el que se ingreso al la función `quotient()`, van del 0 al 19. En la segunda columna tenemos el valor del contador, que muestra el numero de instrucciones ejecutadas. Vemos como el crecimiento es constante.

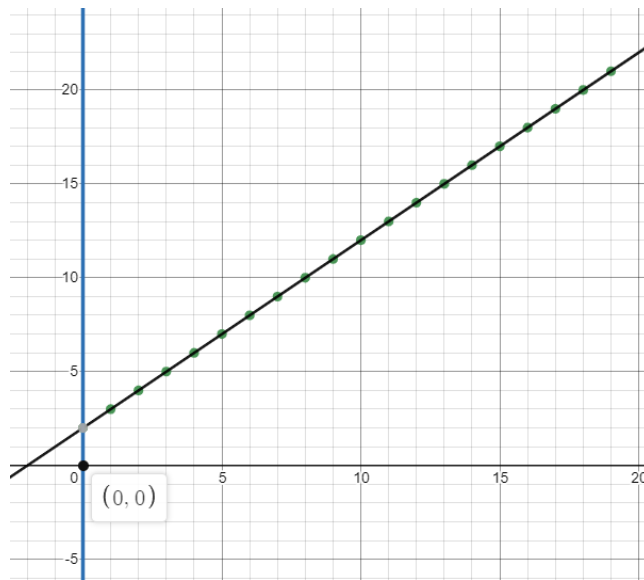
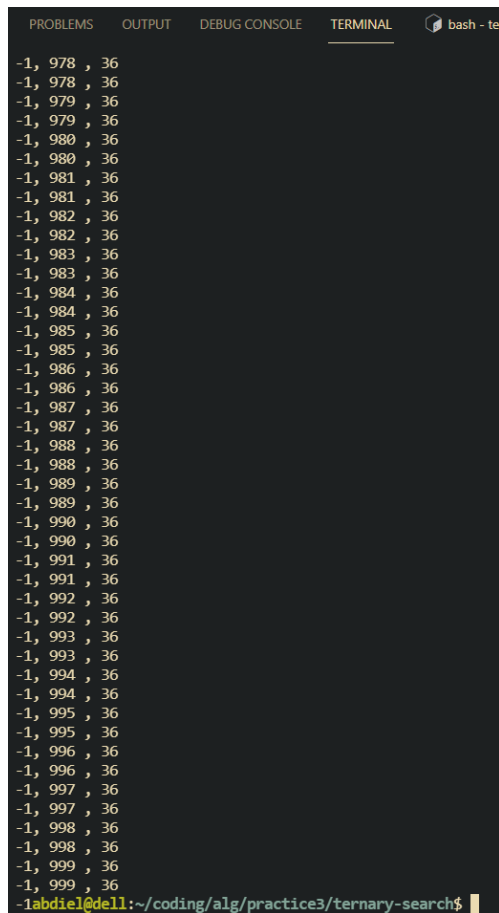


Figura 6: Esta es la gráfica que obtenemos de la tabla anterior. Los puntos son las coordenadas dadas por la tabla, la recta roja es la función $y = x + 2$, se observa que es muy parecida a $y = x$ y no está en el punto $(0,0)$

3.4. Algoritmo de búsqueda ternaria iterativo

Básandose en el algoritmo de búsqueda binaria, se tiene un comportamiento muy similar. Se observa como no tiene un crecimiento lineal, sino logarítmico. Vemos que para este caso, se tiene que $T(n) \in O(\log_3 n)$



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL bash - te
-1, 978 , 36
-1, 978 , 36
-1, 979 , 36
-1, 979 , 36
-1, 980 , 36
-1, 980 , 36
-1, 981 , 36
-1, 981 , 36
-1, 982 , 36
-1, 982 , 36
-1, 983 , 36
-1, 983 , 36
-1, 984 , 36
-1, 984 , 36
-1, 985 , 36
-1, 985 , 36
-1, 986 , 36
-1, 986 , 36
-1, 987 , 36
-1, 987 , 36
-1, 988 , 36
-1, 988 , 36
-1, 989 , 36
-1, 989 , 36
-1, 990 , 36
-1, 990 , 36
-1, 991 , 36
-1, 991 , 36
-1, 992 , 36
-1, 992 , 36
-1, 993 , 36
-1, 993 , 36
-1, 994 , 36
-1, 994 , 36
-1, 995 , 36
-1, 995 , 36
-1, 996 , 36
-1, 996 , 36
-1, 997 , 36
-1, 997 , 36
-1, 998 , 36
-1, 998 , 36
-1, 999 , 36
-1, 999 , 36
-1abdiel@de11:~/coding/alg/practice3/ternary-search$
```

Figura 7: Esta es la ejecución de código en consola, no se muestran todas las impresiones, ya que son 1000 líneas

n	c
1	6
2	6
3	11
4	11
5	11
6	11
7	11
8	11
9	16
10	11
11	16
12	16
13	16
14	16
15	16
16	16
17	16
18	16
19	16
20	16

Cuadro 4: Estos son los resultados de la ejecución del programa. En la primer columna, tenemos el numero con el que se ingreso al la función `iterativeTernarySearch()`, van del 1 al 20, no se muestran las 1000 lineas, por falta de espacio. En la segunda columna tenemos el valor del contador, que muestra el numero de instrucciones ejecutadas. Vemos como se comporta de manera no lineal.

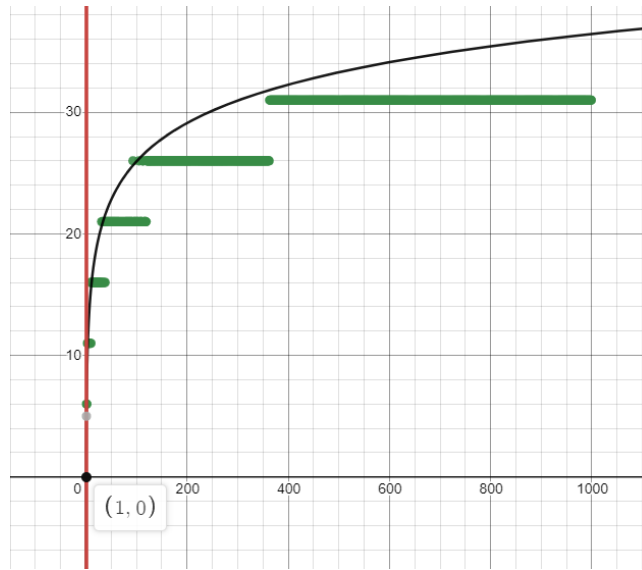
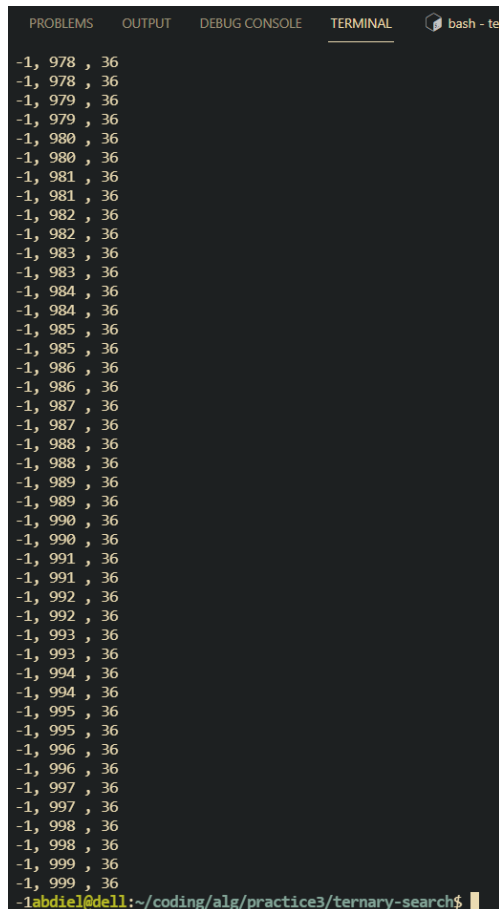


Figura 8: Esta es la gráfica que obtenemos de la tabla anterior. Los puntos son las coordenadas dadas por la tabla, la curva es la función $y = 5 \log_3 x + 5$ obtenida jugando con los valores, se observa que es muy parecida a $y = \log_3 x$ y no está en el punto $(1, 0)$

3.5. Algoritmo de búsqueda ternaria recursivo

Básandose en el algoritmo de búsqueda binaria, se tiene un comportamiento muy similar, inclusive a su contraparte iterativa. Se observa como no tiene un crecimiento lineal, sino logarítmico

Vemos que para este caso, se tiene que $T(n) \in O(\log_3 n)$



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL bash - te
-1, 978 , 36
-1, 978 , 36
-1, 979 , 36
-1, 979 , 36
-1, 980 , 36
-1, 980 , 36
-1, 981 , 36
-1, 981 , 36
-1, 982 , 36
-1, 982 , 36
-1, 983 , 36
-1, 983 , 36
-1, 984 , 36
-1, 984 , 36
-1, 985 , 36
-1, 985 , 36
-1, 986 , 36
-1, 986 , 36
-1, 987 , 36
-1, 987 , 36
-1, 988 , 36
-1, 988 , 36
-1, 989 , 36
-1, 989 , 36
-1, 990 , 36
-1, 990 , 36
-1, 991 , 36
-1, 991 , 36
-1, 992 , 36
-1, 992 , 36
-1, 993 , 36
-1, 993 , 36
-1, 994 , 36
-1, 994 , 36
-1, 995 , 36
-1, 995 , 36
-1, 996 , 36
-1, 996 , 36
-1, 997 , 36
-1, 997 , 36
-1, 998 , 36
-1, 998 , 36
-1, 999 , 36
-1, 999 , 36
-1abdiel@del1:~/coding/alg/practice3/ternary-search$
```

Figura 9: Esta es la ejecucion de codigo en consola, no se muestran todos las impresiones, ya que son 1000 lineas

n	c
1	7
2	7
3	12
4	12
5	12
6	12
7	11
8	12
9	16
10	12
11	16
12	17
13	17
14	17
15	17
16	17
17	17
18	17
19	17
20	17

Cuadro 5: Estos son los resultados de la ejecución del programa. En la primer columna, tenemos el numero con el que se ingreso al la función recursiveTernary-Search(), van del 1 al 20, no se muestran las 1000 lineas, por falta de espacio. En la segunda columna tenemos el valor del contador, que muestra el numero de instrucciones ejecutadas. Vemos como se comporta de manera no lineal.

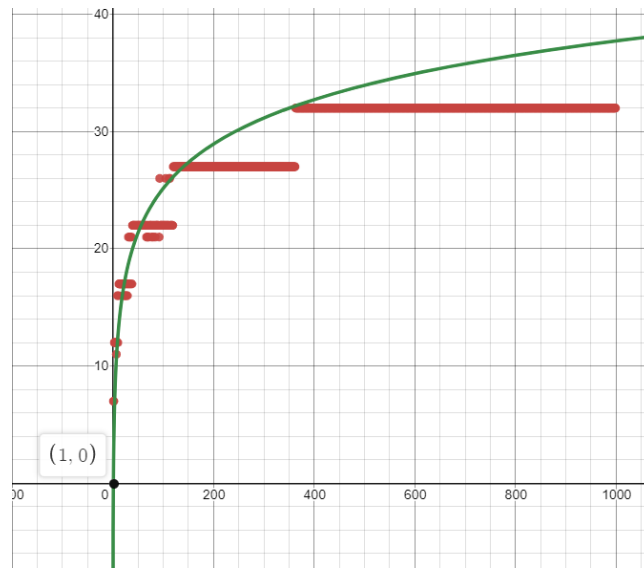


Figura 10: Esta es la gráfica que obtenemos de la tabla anterior. Los puntos son las coordenadas dadas por la tabla, la curva es la función $y = 6 \log_3 x$ obtenida jugando con los valores, se observa que es muy parecida a $y = \log_3 x$. $n0$ está en el punto $(1, 0)$

4. Conclusiones

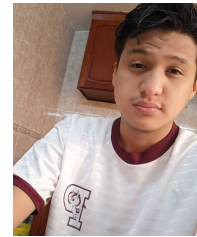


Enrique Abdiel Reyes Rodríguez:

Dada la basta cantidad de problemas que se tuvo para analizar se vió como la iteración no es solo otra manera de escribir código, sino de proveer nuevas soluciones. Creo que el algoritmo más complicado fue la búsqueda ternaria con su análisis a priori. Se observó que hay casos en los que la recursión es muy útil ahorrando líneas y hasta optimizando la complejidad temporal.

Juan Diego Chávez Hernández:

En los análisis a priori se vió como cambia el comportamiento dadas las instrucciones dentro de los bucles, ya no son algoritmos de crecimiento lineal como en las anteriores prácticas. También se observó como la búsqueda ternaria, se basa mucho en la búsqueda binaria, siendo que su complejidad cambiaba en la base del logaritmo.



5. Anexo

5.1. Análisis a priori del algoritmo de división 1

Por medio del análisis por bloques de código vamos analizando el orden de complejidad de adentro hacia afuera. Empezando dentro del ciclo while, vemos como su cuerpo tiene instrucciones de orden constante, realizamos la multiplicación de órdenes igual con la primera línea de la función, resultando que $T(n) \in O(n)$

```
func division1(n,div)
  q = 0-0(1)-----|
  while n>=div -- 0(n)--- 0(n)
    n=n-div-0(1)      |
    q=q+1---0(1)-----
```

Sabiendo que para los ciclos for cuyo incremento se da por $i = 2 * i$ el orden es de $O(\log n)$ para ese bucle. Este algoritmo se podría simplificar con un ciclo for, pero no es así. Está explícito en bucles while. Resultando que $T(n) \in O(\log n)$

```
func division2(n,div)
  q = 0
  dd=div
  r=n
  while dd<=n ---0(log n)---
    dd=2*dd          |---0(log n)
  while dd>div ---0(log n)--
    dd=dd/2
    q=2*q
    if dd<=r
      r=r-dd
      q=q+1
```

Pseudocódigo del algoritmo División 3

```
func division3(n,div)
  if div>n
    return 0
  else
    return 1 + division3(n-div,div)
```

5.2. Análisis a priori del algoritmo de búsqueda ternaria iterativa

Haciendo el análisis por bloques de código hay que tener una consideración: hay un decremento del tipo $\frac{1}{3}$ en `mid_a` y `mid_b` provocando que el ciclo `while` se comporte de manera no lineal, sino logarítmica, despreciando las demás líneas por su complejidad lineal, nos resulta que $T(n) \in O(\log_3 n)$

```
func busqueda(a[],left,right,x)
    while left <= right-----O(log_3 n)
        mid_a = left + ((right - left) / 3)--- Aqui se provoca el
        comportamiento logarítmico.
        mid_b = right - ((right - left) / 3)--- Aqui se provoca el
        comportamiento logarítmico.
        if (a[mid_a] == x)
            return mid_a
        if (a[mid_b] == x)
            return mid_b
        if (x < a[mid_a])
            right = mid_a - 1
        else if (x > a[mid_b])
            left = mid_b + 1
        else
            left = mid_a + 1
            right = mid_a - 1
    return - 1
```

5.3. Análisis a priori del algoritmo de búsqueda ternaria recursiva

El siguiente pseudo código es útil para obtener la ecuación de recurrencia

```
func busqueda(a[],left,right,x)
    if (left > right)
        return -1
    mid_a = left + ((right - left) / 3)
    mid_b = right - ((right - left) / 3)
    if (a[mid_a] == x)
        return mid_a
    if (a[mid_b] == x)
        return mid_b
    if (x < a[mid_a])
        return recursiveTernarySearch(a, left, (mid_a - 1), x)
    else if (x > a[mid_b])
        return recursiveTernarySearch(a, (mid_b + 1), right, x)
```

```

else
    return recursiveTernarySearch(a, (mid_a + 1), (mid_b - 1), x)

```

Se tienen 3 llamadas recursivas $T(\frac{n}{3})$, y C siendo todas las demás instrucciones

$$T(n) = c + T(\frac{n}{3}) \quad (1)$$

$$T(\frac{n}{3}) = C + T(\frac{n}{9}) \quad (2)$$

Sustituyendo 2 en 1

$$T(n) = 2C + T(\frac{n}{9}) \quad (3)$$

$$T(\frac{n}{9}) = 2C + T(\frac{n}{27}) \quad (4)$$

Sustituyendo 4 en 3

$$T(n) = T(\frac{n}{27}) + 3C \quad (5)$$

Se encuentra un patrón

$$T(ni) = T(\frac{n}{3^i}) + iC \quad (6)$$

$$T(\frac{n}{3^i}) = T(1) \quad (7)$$

$$\frac{n}{3^i} = 1 \quad (8)$$

$$\log_3 n = i \quad (9)$$

Sustituyendo 9 en 6

$$T(\frac{n}{3^{\log_3 n}}) + C \log_3 n \quad (10)$$

$$T(\frac{n}{n}) + C \log_3 n \quad (11)$$

$$T(1) + C \log_3 n \quad (12)$$

$$T(n) = 1 + C \log_3 n \quad (13)$$

Por lo tanto $T(n) \in O(\log_3 n)$.

```

abdiel@deli:~/coding/alg/practice3/quotient1$ neofetch
      .-/+oossssoo+/-.
      `:+ssssssssssssssss+`
      -+ssssssssssssssssyyssss+-
      .osssssssssssssssssdmmNnyssso.
      /ssssssssssshdmmNnmyNMMMyhssssss/
      +ssssssssshmydMMMyNddddyssssss++
      /ssssssshNMMMyhhyyyhmmNMMNhssssss/
      .ssssssssdMMNhsssssssssshNMMMdssssss.
      +ssshhhyNMMMysssssssssssyNMMMyssssss+
      ossyNMMMyMMhsssssssssssshmmhssssssso
      ossyNMMMyMMhsssssssssssshmmhssssssso
      +ssshhhyNMMMysssssssssssyNMMMyssssss+
      .ssssssssdMMNhsssssssssshNMMMdssssss.
      /ssssssshNMMMyhhyyyhdNMMNhssssss/
      +ssssssssshmydMMMyNddddyssssss++
      /ssssssssshdmmNnmyNMMMyhssssss/
      .osssssssssssssssssdmmNnyssso.
      -+ssssssssssssssssyyssss+-
      `:+ssssssssssssss+`
      .-/+oossssoo+/-.


abdiel@deli:~/coding/alg/practice3/quotient1$

```

```

OS: Ubuntu 20.04.2 LTS on Windows 10 x86
Kernel: 5.10.60.1-microsoft-standard-WSL
Uptime: 6 hours, 18 mins
Packages: 847 (dpkg)
Shell: bash 5.0.17
Theme: Adwaita [GTK3]
Icons: Adwaita [GTK3]
Terminal: vscode
CPU: Intel i7-8550U (8) @ 1.991GHz
GPU: 7a8c:00:00:00 Microsoft Corporation
Memory: 640MiB / 7900MiB

```



```

abdiel@deli:~/coding/alg/practice3/quotient1$

```

Figura 11: Estas son las especificaciones de la máquina en donde se ejecutó

Referencias

- [1] (s.f.). Recuperado el 10 de Octubre de 2021, de <https://ccia.ugr.es/jfv/ed1/c/cdrom/cap6/cap66.html>
- [2] (s.f.). Gardey., J. P. (s.f.). definicion.de. Recuperado el 06 de Octubre de 2021, de <https://conceptodefinicion.de/iteracion/>
- [3] MITCHELL, C. (19 de Julio de 2021). Investopedia. Recuperado el 5 de Octubre de 2021, de <https://www.investopedia.com/terms/f/fibonaccilines.asp>
- [4] Sipser, M. (2006). Introduction to the Theory of Computation. Course Technology. En M. Sipser, Introduction to the Theory of Computation. Course Technology. Recuperado el 05 de Octubre de 2021