



Instituto Politécnico Nacional  
Escuela Superior de Cómputo



Análisis de Algoritmos, Sem: 2022-2, 3CV11, Práctica 1, 28/02/2022

## PRÁCTICA 1: DETERMINACIÓN EXPERIMENTAL DE LA COMPLEJIDAD TEMPORAL DE UN ALGORITMO

Steiner Vázquez Anthony Francis

*asteinerv1700@alumno.ipn.mx*

**Resumen:** En el presente trabajo se presentan 2 algoritmos, de los cuales veremos su funcionamiento, además, haremos un análisis *a posteriori*, es decir, el análisis experimental, de la complejidad temporal de dichos algoritmos. Esto se hará mediante el uso de un contador, en el código implementado del algoritmo, para contar cada operación realizada por el algoritmo que se esté midiendo, más tarde, mediante el uso de gráficas obtendremos una representación del tiempo que tardan en ejecutarse dichos algoritmos. Para desarrollar cada gráfica, evaluaremos algunos puntos, y buscaremos una función que acote dichos puntos por arriba o, dependiendo del caso, por abajo. Estas funciones serán las funciones de complejidad temporal de los algoritmos.

**Palabras Clave:** C++, Algoritmo, Complejidad, Tiempo

### 1 Introducción

Esta práctica fue realizada para presentar una manera de medir el tiempo que tarda en ejecutarse un algoritmo, y poder apreciar, de manera gráfica, el comportamiento de dichos algoritmos, en algunos casos particulares. Para ello necesitamos, primero, de un algoritmo el cual será medido; segundo, necesitamos checar cuál caso nos interesa, puede ser el mejor caso, el peor, o ambos; tercero, implementar el algoritmo en algún lenguaje de programación, en este caso se usará C++; cuarto, agregaremos un contador al código, con lo cual cada vez que se ejecute una línea de código sumaremos 1 a dicho contador, así mediremos la cantidad de operaciones realizadas por el algoritmo; quinto y último, graficaremos los puntos que se obtienen de la entrada y el número de operaciones ejecutadas, para encontrar la función que lo acote, dependiendo del caso, por ejemplo: para el mejor caso deberá acotarlo por abajo, y para el peor caso, deberá acotarlo por arriba.

Ahora, surgen las preguntas ¿Por qué es importante un algoritmo? ¿Por qué es importante analizar un algoritmo? Para responder a la primera pregunta, diremos que los algoritmos son parte importante de la computación, prácticamente cualquier cosa en la computación fue diseñada e implementada por 1 o más algoritmos, sean para software o para hardware, como por ejemplo las interfaces gráficas, o un algoritmo para desarrollar hardware, y es por eso mismo que un algoritmo puede ser considerado como tecnología, haciendo de su uso, algo cotidiano; a pesar de no ser implementados directamente por uno, pues cada dispositivo que ocupamos ocupa alguno en su núcleo. Para la segunda pregunta hay que ver que un algoritmo es una herramienta que nos permite resolver problemas, y esto nos lleva a considerar cuáles son mejores dependiendo de la memoria y el tiempo que ocupen en resolver dicho problema, esto nos lleva a tener algoritmos que no son correctos, o que simplemente no resuelven un problema de manera eficiente (Cormen, 2001).

Con esto podemos decir que un algoritmo es un procedimiento para realizar una tarea, o en otras palabras, un algoritmo debe resolver un problema bien especificado. Un problema algorítmico es dado mediante la descripción de su conjunto de instancias, con las cuales trabajará, entonces al correr el algoritmo con alguna de estas instancias, la salida debe ser la correcta. A esto le llamamos un algoritmo correcto, ahora bien, no basta con que un algoritmo sea correcto, también debe ser eficiente (Skiena, 1998). Aunque ¿Por qué es tan importante que un algoritmo sea eficiente? Lo veremos con un ejemplo: supongamos que tenemos un algoritmo que para una entrada  $n$ , ejecuta 100,000,000 de operaciones, y otro algoritmo que hace 1,000,000 de operaciones, y ejecutamos estos algoritmos en una computadora que realiza 1,000,000 de instrucciones por segundo. Aunque ambos resuelven el mismo problema, la computadora tardaría 100 segundos en ejecutar el 1er algoritmo, mientras que solo 1 para el segundo. Si a esto le sumamos que los algoritmos son usados por grupos multidisciplinarios que resuelven problemas, y deben hacerlo de la manera más rápida, gastando la menor cantidad de recursos, podríamos decir que el segundo algoritmo es mejor que el primero, aunque hacerlo con este tipo de comparaciones no es lo ideal, nosotros buscaremos funciones que puedan describir el comportamiento, veremos que existen familias de funciones que describen comportamientos similares entre los algoritmos que pueden o no, resolver el mismo problema (Cormen, 2001).

Así, el objetivo de esta práctica es mostrar el funcionamiento de 2 algoritmos que resuelven 2 problemas diferentes, para medir el tiempo que tardan en realizar esta tarea de manera correcta y además darnos una idea de si son eficientes. Finalmente observaremos que el tiempo que tardan en ejecutarse estos algoritmos puede ser acotado por alguna función que dicta el comportamiento del algoritmo, separándolo por casos, para darnos una mejor idea de cuánto debe tardar en ejecutarse el algoritmo que estamos ocupando.

## 2 Conceptos Básicos

Una función  $f$  es una correspondencia que mapea un elemento  $x$  de un conjunto  $D$ , el dominio de la función, a un único elemento  $f(x)$  de un conjunto  $E$ , el rango de la función. Un método para visualizar una función, es a través de su gráfica, esto es, el conjunto de parejas ordenadas  $\{ (x, f(x)) \mid x \in D \}$ .

Un problema  $P$  es una función cuyo dominio es un conjunto de instancias  $I$ , y como contradominio(rango), un conjunto de soluciones  $S$  i.e.  $P : I \rightarrow S$ . Las instancias son una entrada en particular para el problema.

Un algoritmo es una secuencia ordenada de pasos para resolver un problema, este debe ser preciso, es decir, tiene pasos definidos de manera clara; debe ser finito, esto es, que termina en algún momento; y finalmente debe estar definido, lo que nos garantiza que siguiendo los pasos, se llega a la misma salida siempre. Estos se representan generalmente con pseudo-código.

Si dado un problema  $P$  existe un algoritmo  $A$  de tal manera que para cada elemento  $x \in I$ ,  $A$  produce como salida el elemento  $P(x) \in S$  entonces se dice que  $P$  es computable i.e.  $P : I \rightarrow S, \exists A \mid \forall x \in I, A(x) = P(x) \in S \rightarrow P$  es computable. Los problemas que se pueden resolver en una computadora son el conjunto de problemas computables.

Un algoritmo es correcto, si para cada instancia de entrada, el algoritmo se detiene con la salida correcta, en tal caso, decimos que el algoritmo resuelve el problema. Un algoritmo es eficiente cuando resuelve el problema con la menor cantidad de recursos, como por ejemplo el tiempo que tarda en ejecutarse, entonces podemos encontrar una función que nos diga el número de operaciones o pasos a ejecutar por un algoritmo para resolver un problema con una entrada de tamaño  $n$ . Se denota por  $T(n)$ , y se le conoce como función de complejidad temporal del algoritmo.

Al cálculo experimental en la que se recogen datos estadísticos del tiempo consumido por el algoritmo mientras se ejecuta en una computadora en particular, se le conoce como análisis a *posteriori*.

Después de hacer el análisis a *posteriori* de un algoritmo, obtendremos la función de complejidad temporal  $T(n)$ , la cual se puede acotar por arriba, y por abajo con alguna función, lo cual nos da una idea del orden de crecimiento de la función de complejidad, además, nos permite tener una caracterización simple de la eficiencia de un algoritmo, y también nos permite comparar el rendimiento de los algoritmos. A esto se le conoce como eficiencia asintótica de un algoritmo, y tenemos las siguientes notaciones:

$\Theta(g(n)) = \{f(n) : \exists n_0, c_1, c_2 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$ . Esto significa que  $f(n)$  está acotada por arriba y por abajo por  $g(n)$  y esto se cumple para toda  $n \geq n_0$ . Es decir  $g(n)$  es un ajuste asintótico para  $f(n)$ .

$O(g(n)) = \{f(n) : \exists n_0, c > 0 \mid 0 \leq f(n) \leq c g(n) \forall n \geq n_0\}$ . Es decir  $g(n)$  es un ajuste asintótico superior para  $f(n)$ . Se usa para denotar el peor caso de un algoritmo, es decir cuando hace la mayor cantidad de operaciones.

$\Omega(g(n)) = \{f(n) : \exists n_0, c > 0 \mid 0 \leq c g(n) \leq f(n) \forall n \geq n_0\}$ . Es decir  $g(n)$  es un ajuste asintótico inferior para  $f(n)$ . Se usa para denotar el mejor caso de un algoritmo, es decir cuando hace la menor cantidad de operaciones.

Si tenemos 2 funciones  $f(n)$  y  $g(n)$ ,  $f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n))$  y  $f(n) \in \Omega(g(n))$ .

Ya que estas notaciones son conjuntos podríamos escribir  $f(n) \in \Theta(g(n))$  para indicar que  $f(n)$  es un miembro de  $\Theta(g(n))$ . Aunque también podríamos escribir  $f(n) = \Theta(g(n))$  para expresar la misma noción.

Para darnos una idea de cómo se usan estas notaciones haremos una analogía entre la comparación asintótica de 2 funciones  $f$  y  $g$ , y la comparación de 2 números reales  $a$  y  $b$ :

$f(n) = O(g(n))$  es como  $a \leq b$ ,

$f(n) = \Omega(g(n))$  es como  $a \geq b$ ,

$f(n) = \Theta(g(n))$  es como  $a = b$ .

Los algoritmos que se presentan en este trabajo son los siguientes:

#### 1) Algoritmo de búsqueda en sub-arreglos

Este algoritmo checa si para un arreglo dado, existe un elemento igual en los subarreglos que se formarían de partir a la mitad el arreglo, y si existe un elemento así, muestra cuál fue ese elemento, y sus posiciones en el arreglo, en caso de que no exista, simplemente se detiene.

#### 2) Algoritmo de Euclides

Este algoritmo calcula el máximo común divisor de 2 números enteros positivos, mediante el método de divisiones.

A continuación veremos sus pseudo-códigos:

---

**Algorithm 1** BusquedaEnSubArreglo(*arr*):
 

---

```

  tam  $\leftarrow$  arr.size()
  valor  $\leftarrow$  -1
  posi  $\leftarrow$  -1
  posf  $\leftarrow$  -1
  if arr[0] = arr[tam/2] then
    valor  $\leftarrow$  arr
    posi  $\leftarrow$  0
    posf  $\leftarrow$  tam/2
  else
    hashtam  $\leftarrow$  hash.size()
    for i = 0 to i < hashtam do
      hash[i]  $\leftarrow$  -1
    end for
    for i = 0 to i < tam do
      if i < tam then
        hash[arr[i]]  $\leftarrow$  arr[i]
      else
        if hash[arr[i]]  $\neq$  -1 then
          valor  $\leftarrow$  arr[i]
          posf  $\leftarrow$  i
          break
        end if
      end if
    end for
    for i = 0 to i < tam/2 do
      if arr[i] = valor then
        posi  $\leftarrow$  i
        break
      end if
    end for
    if valor  $\neq$  -1 && posi  $\neq$  -1 && posf  $\neq$  -1 then
      print(*)
    end if
  end if

```

---

---

**Algorithm 2** AlgoritmoDeEuclides( $m, n$ ) :
 

---

```

while  $n \neq 0$  do
     $r \leftarrow m \bmod n$ 
     $m \leftarrow n$ 
     $n \leftarrow r$ 
end while
return  $m$ 

```

---

Ahora veremos su funcionamiento dando unos ejemplos de instancias: Para el algoritmo de BúsquedaEnSubArreglo daremos como instancia el arreglo  $A[4, 5, 0, 9, 4, 1, 2, 5, 3, 0]$  y la solución es que el valor repetido en los subarreglos  $A[0, \dots, n/2]$  y  $A[n/2 + 1, \dots, n - 1]$  es 5, en las posiciones [1] y [7]. El algoritmo guarda en *tam* el valor del tamaño del arreglo e inicializa las variables *valor*, *posi* y *posf* con -1. Checa si el valor del primer subarreglo es el mismo que el del segundo subarreglo, esto sería el mejor caso, de ser así, actualiza los valores de las variables *valor*, *posi* y *posf*; ya que no es el caso entonces creamos un arreglo auxiliar y en la variable *hashtam* guardamos el valor de su tamaño, ya que este arreglo será usado para hashear debe ser suficientemente grande, es decir, debe tener como tamaño el valor máximo contenido en nuestro arreglo original, como mínimo; ya que usaremos la función de hasheo identidad, es decir guardaremos el valor de nuestro arreglo original en la posición que sea igual a ese valor, y por tanto primero debemos inicializar todos sus elementos con -1. Luego itera sobre nuestro arreglo original y checa si estamos en la primera mitad, de ser así, hashea los elementos, ya en la 2da mitad checa si ese valor ya fue hasheado, pues esto quiere decir que encontramos un valor repetido, entonces actualizaremos las variables *valor* y *posf*, y romperemos el ciclo. Aunque nos falta encontrar el valor de *posi*, así que volveremos a iterar sobre nuestro arreglo pero esta vez solo en la primera mitad, y en el momento que encontremos el valor, actualizamos el valor de *posi*, y romperemos el ciclo. Finalmente si se ha encontrado un elemento en ambos subarreglos, lo muestra en pantalla, de lo contrario, simplemente se detiene.

Por otro lado el algoritmo de Euclides entra en un ciclo hasta que nuestra variable  $n$  sea 0, posteriormente guarda en una variable  $r$  el residuo de la división entre  $m$  y  $n$ , y después guarda el nuevo  $m$  que será  $n$ , y el nuevo  $n$ , que será  $r$ , es decir el residuo que ya habíamos calculado anteriormente, de esta manera siempre estaremos calculando el modulo, del modulo, y así sucesivamente desde la primera división hasta que nuestra  $n$ , que fue el último modulo, nos de 0, de tal manera en  $m$  habremos guardado el valor de nuestro máximo común divisor, que para el peor caso se trata de los primos relativos, que se dan en valores consecutivos de la sucesión de Fibonacci, es decir 2 números cuyo máximo común divisor  $mcd(m, n) = 1$ .

Por ejemplo con la instancia  $m = 34$ ,  $n = 21$ , la solución es 1, puesto que son primos relativos. Por lo tanto el algoritmo empieza en un ciclo donde inicialmente  $r = 0$ ,  $m = 34$ , y  $n = 21$ ; en la 1ra iteración  $r = 13$ ,  $m = 21$ , y  $n = 13$ , en la 2da iteración  $r = 8$ ,  $m = 13$ ,  $n = 8$ , en la 3ra iteración  $r = 5$ ,  $m = 8$ , y  $n = 5$ , en la 4ta iteración  $r = 3$ ,  $m = 5$ , y  $n = 3$ , en la 5ta iteración  $r = 2$ ,  $m = 3$  y  $n = 2$ , en la 6ta iteración  $r = 1$ ,  $m = 2$ , y  $n = 1$ , y por último en la 7ma iteración  $r = 0$ ,  $m = 1$ , y  $n = 0$ , por lo tanto deja de iterar, y nuestro  $mcd$  es igual a 1, recordemos que el máximo común divisor, es el número más grande que divide a ambos números sin dejar residuo.

Podemos observar que ambos algoritmos son correctos, así que procedemos a checar su eficiencia, haciendo un análisis *a posteriori* de los mismos.

### 3 Experimentación y Resultados

Para nuestro primer algoritmo el tamaño del problema es el tamaño del arreglo donde buscaremos al elemento en los subarreglos. Así que checaremos cuántas operaciones realiza nuestro algoritmo dependiendo del valor de  $tam$ , al que vamos a restringir a números pares, tanto para el mejor caso, como para el peor caso; como previamente discutimos, el mejor caso se encuentra cuando el valor en la primera posición de ambos subarreglos es igual, sin embargo, para poder definir cuál es el peor caso deberemos de checar más a detalle la implementación del algoritmo, puesto que existe la opción de que el peor caso sea que no exista un elemento que cumpla que esté en ambos subarreglos, sin embargo, también puede ser que el peor caso, sea cuando el valor que se repite está en las últimas posiciones de los subarreglos. Así que nos apoyaremos del análisis *a posteriori* para determinar cuál es el peor caso, y además saber cuál es la función que lo acota, es decir, su función de complejidad temporal.

Para el segundo algoritmo definiremos como tamaño del problema a nuestro número  $m$ , ya que en particular para el peor caso estaremos checando números consecutivos de la sucesión de Fibonacci, de tal manera que  $m$  será el mayor de estos 2 números, y vamos a checar cuántas operaciones realiza dadas estas condiciones.

Ahora pasaremos a ejecutar nuestros algoritmos, y contar el número de operaciones que ejecuta para realizar su tarea correctamente, para ello hemos agregado al código en C++ un contador, que aumentara por cada operación realizada por el algoritmo. Así bien procedemos a ejecutar nuestro primer algoritmo y analizar su comportamiento tanto en el mejor como en el peor caso.

Empezaremos con nuestro algoritmo de búsqueda en subarreglos.

```

> ./BusquedaEnSubArreglo 10
Arreglo A:
10 25 16 3 30 1 12 0 17 25
-----
dato: 25, posiciones: [1], [9].
El algoritmo hizo: 63 operaciones.

/mnt/c/Users/DELL/Documents/Algoritmos/P1

```

Figura 1. Ejecución del código de Búsqueda en subarreglo.

Para analizar a *posteriori* un algoritmo iremos aumentando el valor de  $n$  que es el tamaño del problema, entonces procedemos a ejecutar nuestro código aumentando de 10 en 10 el tamaño del arreglo que se guarda en *tam* puesto que esa es nuestra  $n$ , para obtener varias mediciones.

Así para el mejor caso tenemos:

m	# operaciones
10	11
20	11
30	11
40	11
50	11
60	11
70	11
80	11
90	11
100	11

Table 1: Mediciones para el mejor caso de Búsqueda en subarreglo

Después, para determinar el peor caso, se tuvieron que hacer mediciones para cuando se encuentran valores que cumplan las condiciones y para cuando no se cumple.



Así para el caso en que se encuentran los valores:

<b>m</b>	<b># operaciones</b>
10	55
20	86
30	178
40	221
50	152
60	196
70	269
80	318
90	345
100	366

Table 2: Mediciones para el caso en que se encuentran valores iguales

Ahora para cuando no se encuentran los valores:

<b>m</b>	<b># operaciones</b>
10	67
20	122
30	177
40	232
50	287
60	342
70	397
80	452
90	507
100	562

Table 3: Mediciones para el caso en que no se encontraron valores iguales

Así podemos observar que el peor caso es cuando no existen valores iguales en los subarreglos.

Con esta información vamos a proponer funciones que pasen por la mayor cantidad de puntos, o que los acoten, y diremos que esa es su  $T(n)$ , para posteriormente graficar el número de operaciones v.s.  $n$ .

Entonces las  $T(n)$  para el algoritmo de búsqueda en subarreglos quedan: para el mejor caso, propondremos la función  $f(n) = 11$ , y para el peor caso proponemos  $f(n) = 7n$ .

Veremos como se comporta de manera gráfica nuestro algoritmo en el mejor caso:

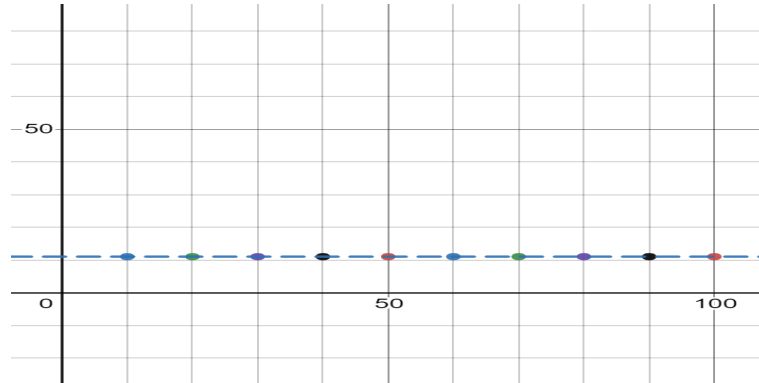


Figura 2. Gráfica de la función  $f(n) = 11$

Podemos observar que todos los puntos pasan por  $f(n) = 11$ . Ahora, de la definición de la notación  $\Omega$ :

Necesitamos encontrar un  $c_1 \leq 11 \forall n \geq n_1$ , de ahí que:

$10(1) \leq 11 \forall n \geq 1$ , entonces  $c_1 = 10$ ,  $g(n) = 1$ , y  $n_1 = 1 \therefore f(n) = 11 \in \Omega(1)$ .

Ahora veremos que pasa para el peor caso:

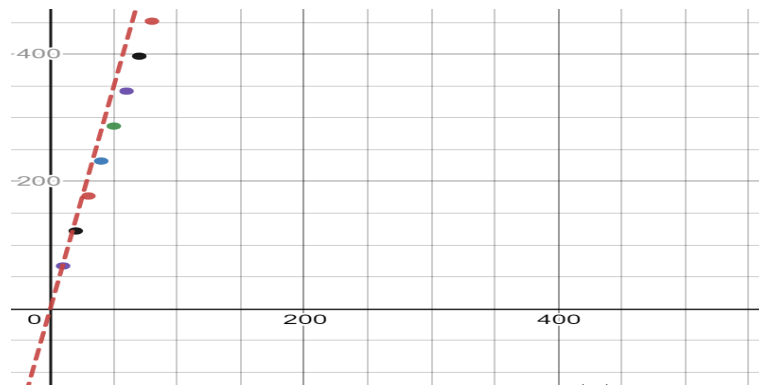


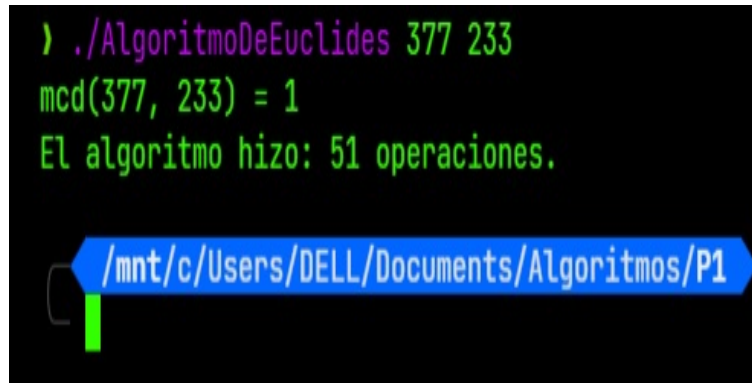
Figura 3. Gráfica de la función  $f(n) = 7n$

Podemos observar que algunos puntos pasan por  $f(n) = 7n$  y los demás quedan acotados por arriba. Ahora, de la definición de la notación  $O$ :

Necesitamos encontrar un  $c_1 g(n) \geq 7n \forall n \geq n_1$ , de ahí que:

$7n \leq 8n \forall n \geq 1$ , entonces  $c_1 = 8$ ,  $g(n) = n$ , y  $n_1 = 1 \therefore f(n) = 7n \in O(n)$ .

Por último veremos el funcionamiento del algoritmo de Euclides:



```

) ./AlgoritmoDeEuclides 377 233
mcd(377, 233) = 1
El algoritmo hizo: 51 operaciones.

/mnt/c/Users/DELL/Documents/Algoritmos/P1

```

Figura 4. Ejecución del código para Euclides.

Ahora analizaremos los resultados experimentales arrojados por el algoritmo de Euclides, para el peor caso:

m	# operaciones
1	7
2	7
3	11
5	15
8	19
13	23
21	27
34	31
55	35
89	39
144	43

Table 4: Mediciones para el peor caso de Euclides

Para el peor caso de euclides proponemos para  $T(n)$  la función  $f(n) = 18\log_2(n)$ .

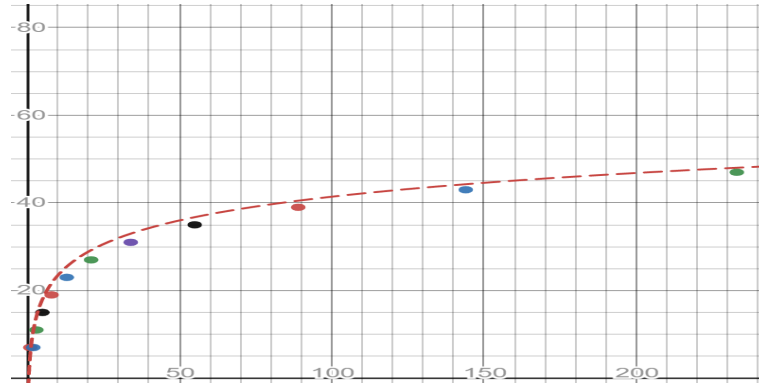


Figura 5. Gráfica de la función  $f(n) = 18\log_2(n)$

Podemos observar que algunos puntos pasan por  $f(n) = 18\log_2(n)$  y los demás quedan acotados por arriba. Ahora, de la definición de la notación  $O$ :

Necesitamos encontrar un  $c_1 g(n) \geq 18\log_2(n) \forall n \geq n_1$ , de ahí que:

$18\log_2(n) \leq 19\log_2(n) \forall n \geq 2$ , entonces  $c_1 = 19$ ,  $g(n) = \log_2(n)$ , y  $n_1 = 2 \therefore f(n) = 18\log_2(n) \in O(\log_2(n))$ .

## 4 Conclusiones

### Conclusión General

Uno de los problemas que se presentó fue que la complejidad para el 1er algoritmo era cuadrática, pero no se estuvo satisfecho con ese rendimiento, debido a esto, fue necesario checar si un mapeo hash resolvía el problema, y de esto surgió el problema de encontrar una función hash que se adecuara a nuestra situación, así que se optó por la función hash de identidad, aunque esta funciona bien solo para valores pequeños, así que quizás una manera de poder mejorar este algoritmo sería el de implementar una mejor función hash, que sea más general, aunque independientemente de eso, pudimos observar que nuestro algoritmo es correcto, por lo tanto se tuvieron los resultados esperados, y ahora con una complejidad lineal, así que se logró mejorar la eficiencia del algoritmo; aunque es posible que para cuando no se encuentren valores repetidos en los subarreglos, se podría aún mejorar la cantidad de operaciones, pero seguiría siendo lineal. Otra duda que surgió era el qué sucedía si nuestro algoritmo no encontraba un elemento con las características que nos solicitaban, así que se optó por inicializar todas nuestras variables auxiliares con -1, y si no se modificaban, entonces simplemente no mostraba nada, pero se detenía de todos modos. Para el segundo algoritmo surgieron dudas de realmente cuál era la función que acotaba por arriba nuestros puntos, pero después de analizar puntos fuera del peor caso, y checar más puntos, nos dimos cuenta de que la función tenía un comportamiento parecido al logarítmico.

### Conclusión de Anthony

En particular de los algoritmos que se checaron en esta práctica, del 1ro me llamó mucho la atención el poder analizar un algoritmo que aunque ya es correcto, no quedas satisfecho con su desempeño, y por lo tanto regresar a la fase de diseñar el algoritmo, y checar como optimizarlo, fue muy interesante. Además de poder pensarlo en casos, también ayudó a quitar complejidad innecesaria del algoritmo. Del segundo algoritmo, me llamó mucho la atención que justo el peor caso para calcular el mcd de 2 números sea cuando cumplan la propiedad de que sean primos relativos, y que justo 2 números consecutivos de la sucesión de Fibonacci cumplan esta propiedad, me pareció increíble.



## 5 Anexo

Para este caso no se realizaron problemas de tarea para anexar.

## 6 Bibliografía

Cormen, T. H., & Cormen, T. H. (2001). Introduction to algorithms. Cambridge, Mass: MIT Press.  
Skiena, S. S. (1998). The algorithm design manual. New York: Springer-Verlang.