# Informe Taller Complejidad

Simón Álvarez
David Madrid

August 2020

# 1 Max Value

El algoritmo se encarga de calcular el número máximo de elementos de un arreglo usando recursividad. Sea $n$ la longitud de los elementos, que incrementa en una unidad. Sea $c$ las constantes involucradas en el algoritmo (decisiones, declaraciones, retornos...). Sea $T(n)$ la función que describe la complejidad del algoritmo, entonces, se tiene que la complejidad del algoritmo está dada por:

$T(n) = c_1 + T(n+1)$, $0 \leq n$. Por tanto, $n$ está entre 0 y $m$, donde $m$ es la longitud del arreglo el cual se le busca el valor máximo. Otra forma de escribir la complejidad del algoritmo es: $T(n) = c_1 + T(n-1)$, lo cual tiene forma de sucesión aritmética y, por tanto, $T(n) = c_1 \cdot n + c_2$.

A continuación, hay una tabla de valores la cual se le intentó hallar el elemento máximo cuando este no hubiera uno mayor que los demás.

| Java | |
|---|---|
| Number of Elements | Time (s) |
| 800 | 2.00E-04 |
| 1600 | 4.36E-05 |
| 2400 | 5.64E-05 |
| 3200 | 7.56E-05 |
| 4000 | 7.00E-05 |
| 4800 | 7.88E-05 |
| 5600 | 1.17E-04 |
| 6400 | 9.05E-05 |
| 7200 | 9.27E-05 |
| 8000 | 1.03E-04 |
| 8800 | 1.17E-04 |
| 9600 | 3.66E-05 |
| 10400 | 3.88E-05 |
| 11200 | 4.17E-05 |
| 12000 | 7.69E-05 |
| 12800 | 1.21E-04 |
| 13600 | 5.54E-05 |
| 14400 | 5.55E-05 |
| 15200 | 8.66E-05 |
| 16000 | 9.33E-05 |

Table 1: Tiempo necesario para encontrar el elemento máximo según el tamaño del arreglo en Java.

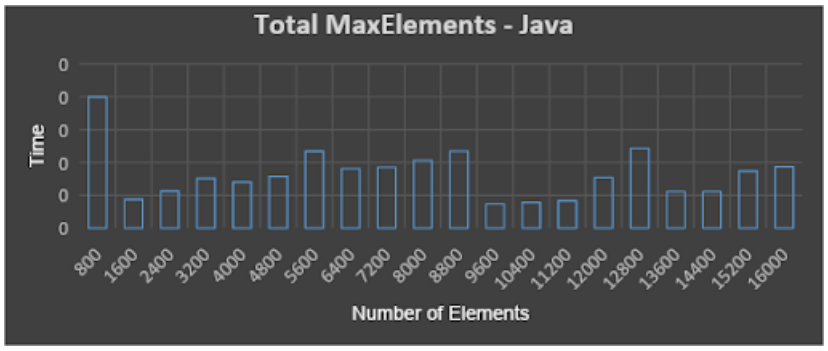


Figure 1: Gráfico de la tabla 2

A continuación, se presenta otra tabla de datos que hizo su entrada en el lenguaje de programación Python.

| Python | |
|---|---|
| Number of Elements | Time (s) |
| 1046 | 0.00116 |
| 2092 | 0.00342 |
| 3138 | 0.00181 |
| 4184 | 0.00253 |
| 5230 | 0.00587 |
| 6276 | 0.01413 |
| 7322 | 0.00851 |
| 8368 | 0.00720 |
| 9414 | 0.01456 |
| 10460 | 0.01176 |
| 11506 | 0.00978 |
| 12552 | 0.01615 |
| 13598 | 0.01131 |
| 14644 | 0.01670 |
| 15690 | 0.00970 |
| 16736 | 0.01191 |
| 17782 | 0.01360 |
| 18828 | 0.01349 |
| 19874 | 0.01342 |
| 20920 | 0.01960 |

Table 2: Tiempo necesario para encontrar el elemento máximo según el tamaño del arreglo en Python.

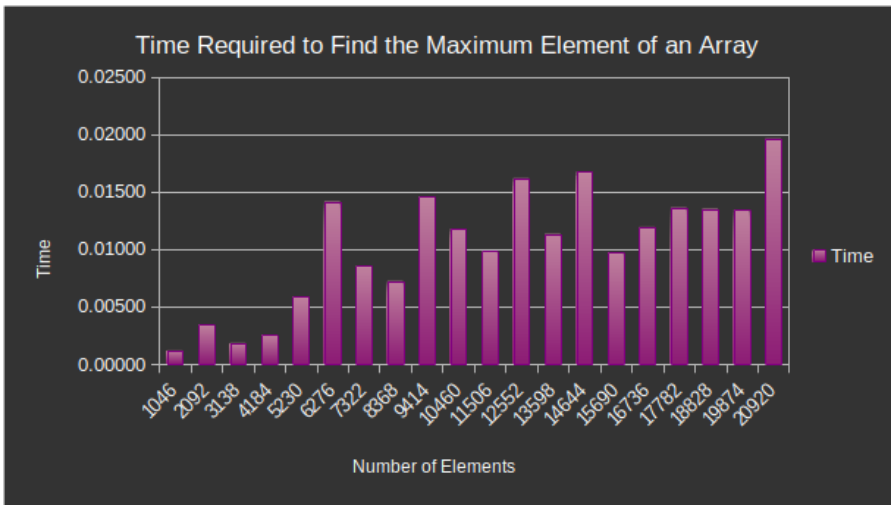


Figure 2: Gráfico de la tabla 2

La complejidad se definirá como $O(c_1 \cdot n + c_2)$. Por la regla de la suma, obtenemos $O(c_1 \cdot n)$. Por la regla del producto la complejidad final es $O(n)$.

# 2 Max Volume

El algoritmo se encarga de calcular si la suma de los volúmenes de una cantidad indicada de objetos puede ser igual al volumen máximo estipulado por el usuario.

Sea $n$ la cantidad de objetos con volúmenes. Sea $m$ la posición del volumen de algún objeto. Esta variable depende de la cantidad de los elementos $(n)$ y no puede ser menor a 0. Sean las $c_n$ las constantes involucradas en el algoritmo (decisiones, declaraciones, retornos...). Sea $T(n)$ la función que describe la complejidad del algoritmo. Así, se tiene que la complejidad del algoritmo está dada por:

$T(n,m) = c_1 + T(n,m+1) + T(n–1,m)$, $0 \leq m < n$. Como la distancia entre las variables en los cambios del lado derecho $(n,m)$ es igual, o sea, $m+1-n = m-n+1 := m-(n-1) = m-n+1$ y la forma en la que crece o disminuyen las variables es simétrica y sus restricciones no varían en ninguna de las dos funciones, entonces la fórmula se puede reducir como: $T(n) = 2 \cdot T(n-1) + c_1$. Así, la fórmula general será: $T(n) = c_1 \cdot (2^n - 1) + c_2 \cdot 2^(n-1)$, $c_1, c_2$ constantes.

La siguiente tabla muestra los datos introducidos y los resultados entregados en Java.

| Java | |
|---|---|
| Number of Elements | Time (s) |
| 225 | 2.07E-05 |
| 450 | 4.89E-05 |
| 675 | 1.12E-04 |
| 900 | 5.84E-05 |
| 1125 | 6.74E-05 |
| 1350 | 7.60E-05 |
| 1575 | 8.49E-05 |
| 1800 | 1.33E-04 |
| 2025 | 9.65E-05 |
| 2250 | 1.03E-04 |
| 2475 | 1.12E-04 |
| 2700 | 1.57E-04 |
| 2925 | 1.28E-04 |
| 3150 | 1.37E-04 |
| 3375 | 1.44E-04 |
| 3600 | 1.53E-04 |
| 3825 | 1.62E-04 |
| 4050 | 2.13E-04 |
| 4275 | 1.79E-04 |
| 4500 | 2.25E-04 |

Table 3: Tiempo necesario para hallar una combinación de elementos que se ajuste al volumen máximo en Java.

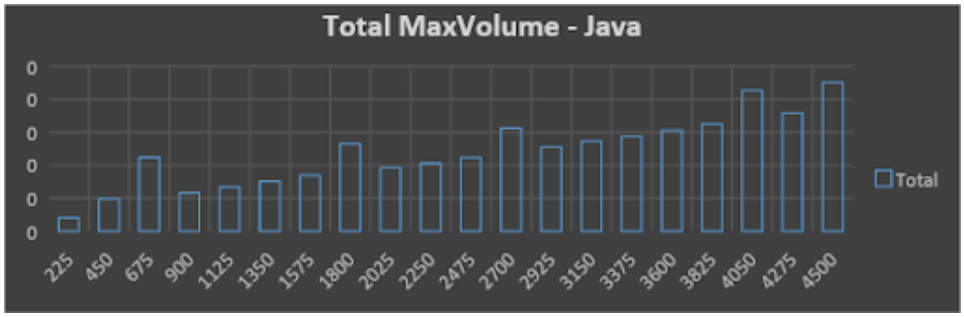


Figure 3: Gráfica de la tabla 3

Ahora, se presenta una tabla de datos que se introdujo en el lenguaje de Python.

| Python | |
|---|---|
| Number of Elements | Time (s) |
| 1 | 4.53E-06 |
| 2 | 5.72E-06 |
| 3 | 8.82E-06 |
| 5 | 3.17E-05 |
| 6 | 6.72E-05 |
| 7 | 1.56E-04 |
| 8 | 2.63E-04 |
| 9 | 5.58E-04 |
| 10 | 1.29E-03 |
| 12 | 5.64E-03 |
| 13 | 1.13E-02 |
| 14 | 3.14E-02 |
| 15 | 8.13E-02 |
| 16 | 1.06E-01 |
| 17 | 2.10E-01 |
| 18 | 4.36E-01 |
| 20 | 1.88E+00 |
| 21 | 3.81E+00 |
| 22 | 7.85E+00 |
| 23 | 1.61E+01 |

Table 4: Tiempo necesario para hallar una combinación de elementos que se ajuste al volumen máximo en Python.

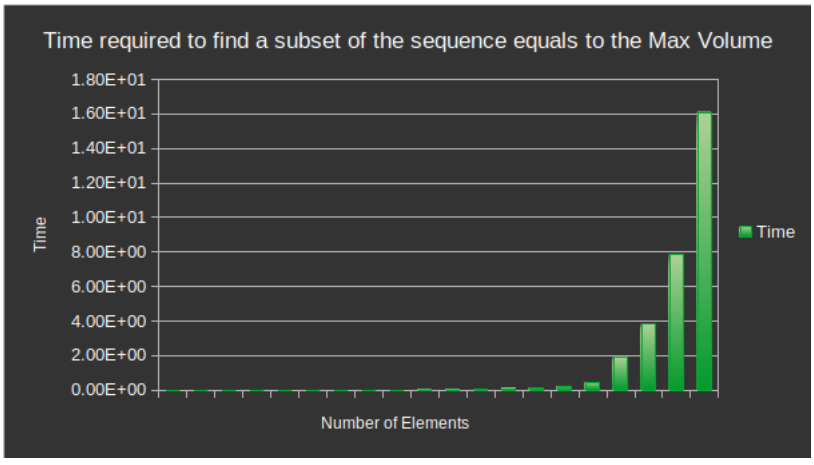


Figure 4: Gráfica de la tabla 4.

Para calcular la complejidad, definimos $O$ como $O(c_1 \cdot (2^n - 1) + c_2 \cdot 2^{n-1})$,

por regla de la suma obtenemos $O(c_1 \cdot 2^n + c_2 \cdot 2^n)$. Por regla del producto obtenemos $O(2^n)$.

# 3 Fibonacci

La sucesión de Fibonacci fue descrita por Leonardo de Pisa en el siglo XIII, la cual hoy en día tiene muchas aplicaciones en diversos campos como las ciencias de la computación, matemáticas y teoría de juegos[1]. En este caso, utilizamos un algoritmo para hallar el enésimo término de la sucesión de Fibonacci.

Sea $n$ el número en la sucesión que se desea encontrar. Sean las $c_n$ las constantes involucradas en el algoritmo (decisiones, declaraciones, retornos...). Sea $T(n)$ la función que describe el número de operaciones que debe hacer el algoritmo. Así, se tiene que:

$T(n) = c_1 + T(n-1) + T(n-2), n \geq 2.$

Definiremos $\phi = \frac{(5+1)}{2}$. Así, $T(n) = c_1 + c_2 \cdot F_n + c_3 \cdot L_n$, Donde $F_n$ es el término enésimo de la sucesión de Fibonacci y $L_n$ es el término enésimo de la sucesión de Lucas. Las fórmulas para la sucesión de Fibonacci y de Lucas, respectivamente, son:

$F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$, $L_n = \phi^n + (1-\phi)^n$

Si utilizamos estas fórmulas para resscribir $T(n)$, obtendremos lo siguiente:

$T(n) = c_1 + k_1 \cdot (\phi^n - (1-\phi)^n) + k_2 \cdot (\phi^n + (1-\phi)^n)$
$T(n) = c_1 + k_3 \cdot \phi^n + k_4 \cdot (1-\phi)^n, k_n$ constante.

A continuación, se presenta una base de datos introducida en Java.

[1] Sacado de: Wikipedia

| Java | |
|---|---|
| Time (s) | Number of Elements |
| 1.800E-05 | 3 |
| 3.200E-06 | 5 |
| 2.800E-06 | 8 |
| 9.300E-06 | 10 |
| 2.720E-05 | 13 |
| 6.080E-05 | 15 |
| 1.762E-04 | 18 |
| 2.730E-04 | 23 |
| 9.209E-04 | 26 |
| 1.385E-03 | 28 |
| 5.812E-03 | 31 |
| 1.665E-02 | 33 |
| 6.276E-02 | 36 |
| 1.566E-01 | 38 |
| 6.558E-01 | 41 |
| 1.71 | 43 |
| 7.24 | 46 |
| 19.09 | 48 |
| 80.26 | 51 |

Table 5: Tiempo necesario para hallar una combinación de elementos que se ajuste al volumen máximo en Java.

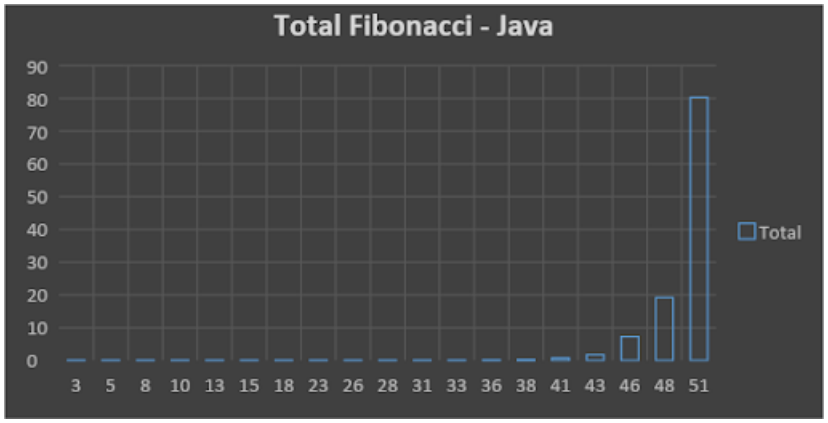


Figure 5: Gráfico de la tabla 5.

La siguiente tabla se introdujo en Python.

| Python | |
|---|---|
| Time (s) | Fibonacci's Index |
| 0.004 | 22 |
| 0.006 | 23 |
| 0.008 | 24 |
| 0.015 | 25 |
| 0.019 | 26 |
| 0.030 | 27 |
| 0.050 | 28 |
| 0.083 | 29 |
| 0.136 | 30 |
| 0.217 | 31 |
| 0.341 | 32 |
| 0.543 | 33 |
| 0.895 | 34 |
| 1.397 | 35 |
| 2.234 | 36 |
| 3.621 | 37 |
| 5.941 | 38 |
| 9.549 | 39 |
| 15.244 | 40 |
| 25.248 | 41 |

Table 6: Tiempo necesario para hallar una combinación de elementos que se ajuste al volumen máximo en Python.

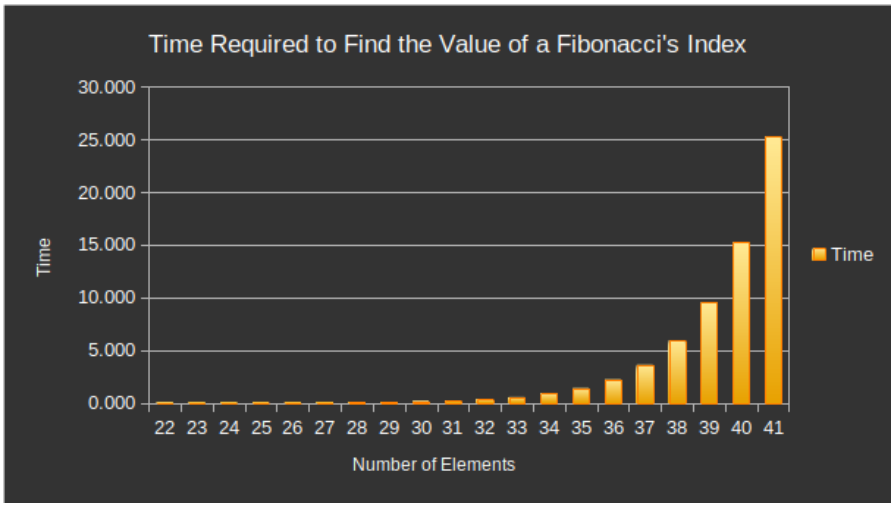


Figure 6: Gráfico de la tabla 6.

La complejidad de este algoritmo se define como $O(c_1 + k_3\phi^n + k_4 \cdot (1-\phi)^n)$. Por regla de la suma, esto es igual a $O(k_3\phi^n + k_4 \cdot (1-\phi)^n)$ que, por regla del producto, finalmente queda como $O(\phi^n + (-\phi)^{-n})$.

# References

[1] https://es.wikipedia.org/wiki/Sucesiprober for active measurement of the internet, 2010.