Complejidad de algoritmos

Salvador López Mendoza

Noviembre 2022

Introducción

En muchas ocasiones es importante estimar el tiempo que se tarda un programa en resolver un problema.

¿Cómo saber de antemano el tiempo que tardará el programa en ejecutarse?

Enfoque empírico:

Ejecutar el programa varias veces.
Para cada ocasión, anotar el tiempo que tarda en ejecutarse.
Extrapolar los resultados.

Introducción (II)

¿De qué depende el tiempo de ejecución?

1. Procesador más veloz.
2. Mayor cantidad de memoria.
3. Mayor cantidad de procesadores.

¿Y el algoritmo?

Complejidad de algoritmos

Al conocer la complejidad de un algoritmo podemos saber qué esperar de él en cuanto al tiempo de ejecución.

Al determinar la complejidad de un algoritmo se puede conocer el tiempo de ejecución del mismo.

Este tiempo generalmente depende de la cantidad de datos que se vaya a procesar y se denota por $T(n)$, donde n es la cantidad de los datos.

Desafortunadamente no es fácil encontrar la función $T(n)$.

Complejidad de algoritmos (II)

En caso de encontrar la función $T(n)$, es una función difícil de manejar.

Se buscan funciones que siendo fáciles de manejar permitan dar una buena idea del comportamiento del algoritmo.

En general se buscan funciones que sean **cotas superiores** de $T(n)$.

Factores de crecimiento

Dado que el tiempo de ejecución de un programa depende de la cantidad de datos, es de esperar que la función $T(n)$ sea una **función creciente**.

Lo importante es conocer el *factor de crecimiento* en función de los datos.

Algunas funciones sencillas, ¿cuál crece más rápido?

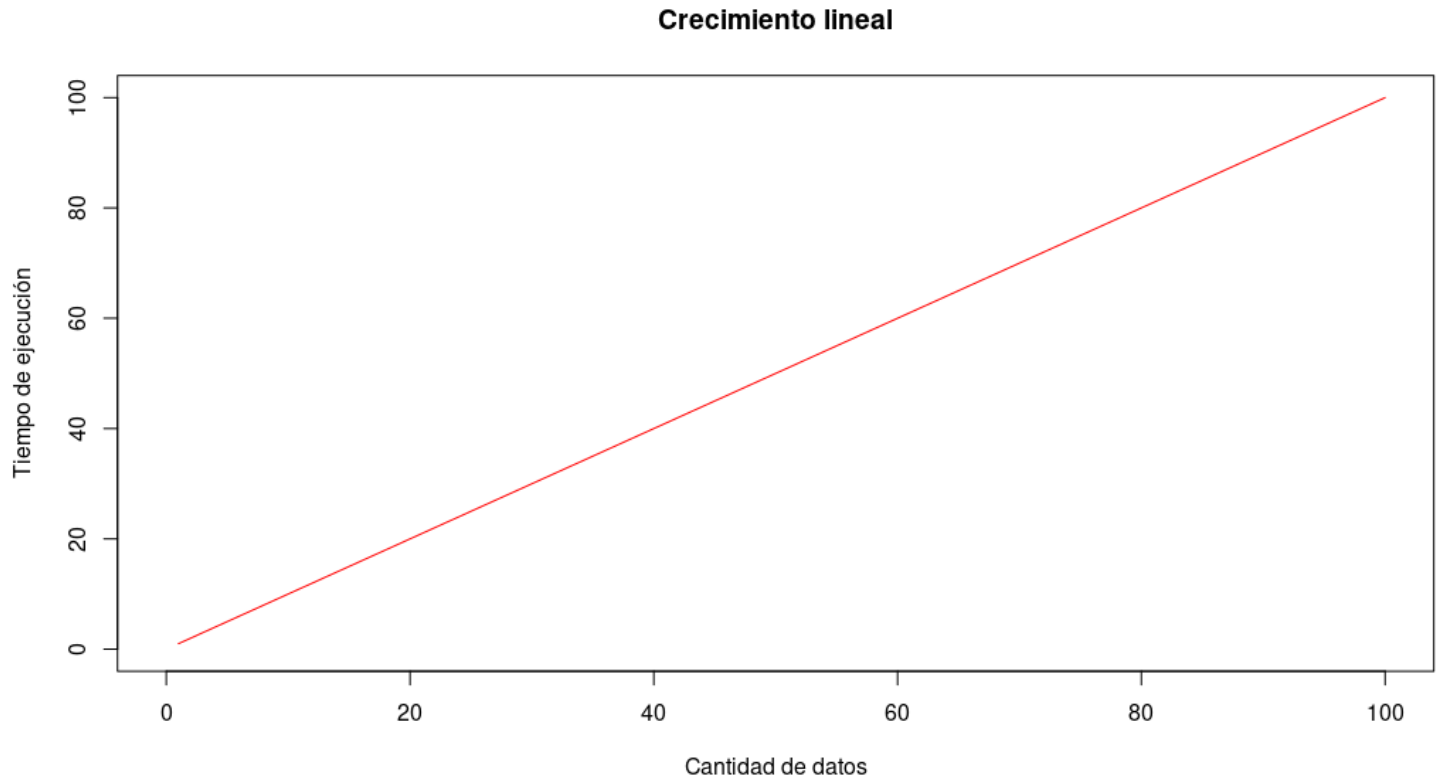
$$f(x) = x$$

$$f(x) = x^2$$

$$f(x) = x^3$$

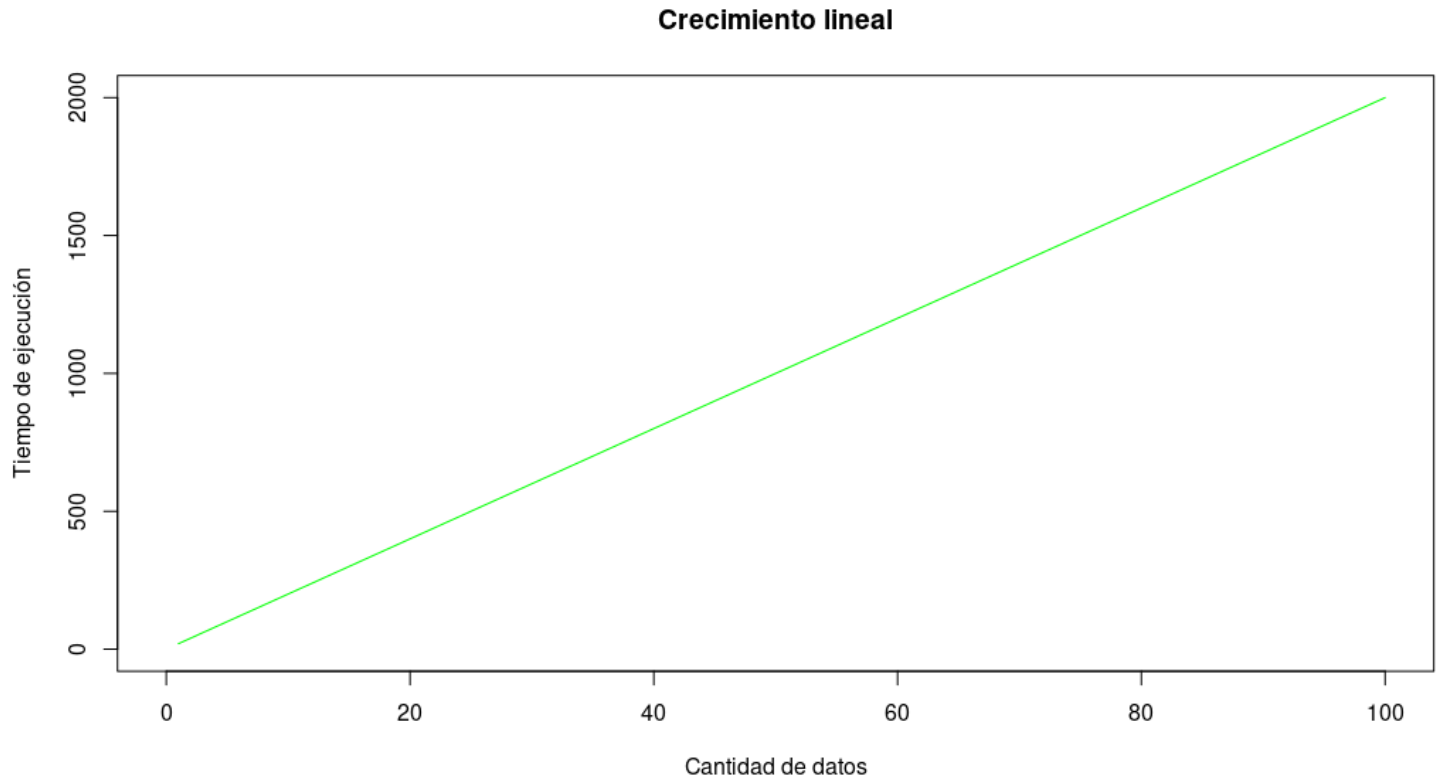
Factores de crecimiento (lineal)

$f(x) = x$



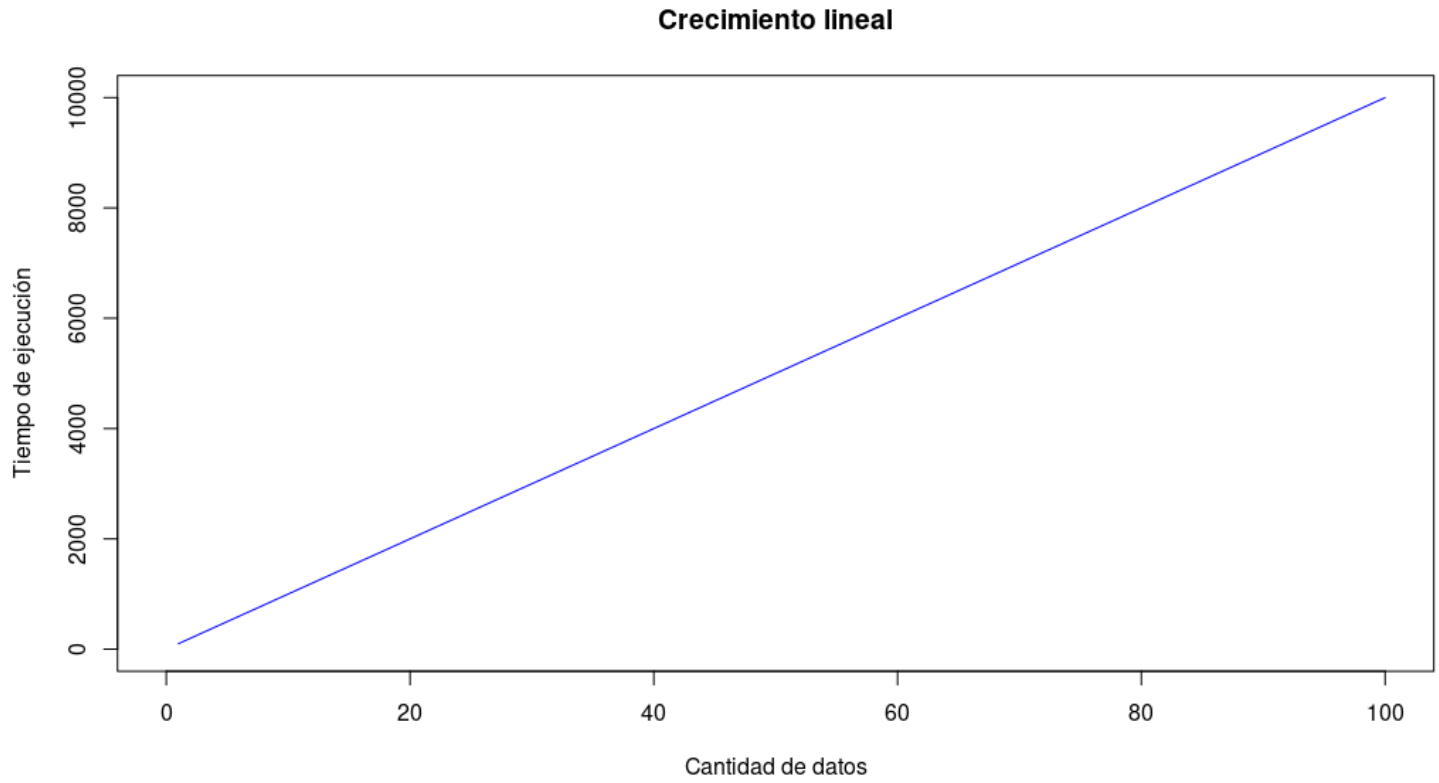
Factores de crecimiento (lineal II)

$f(x) = 20x$



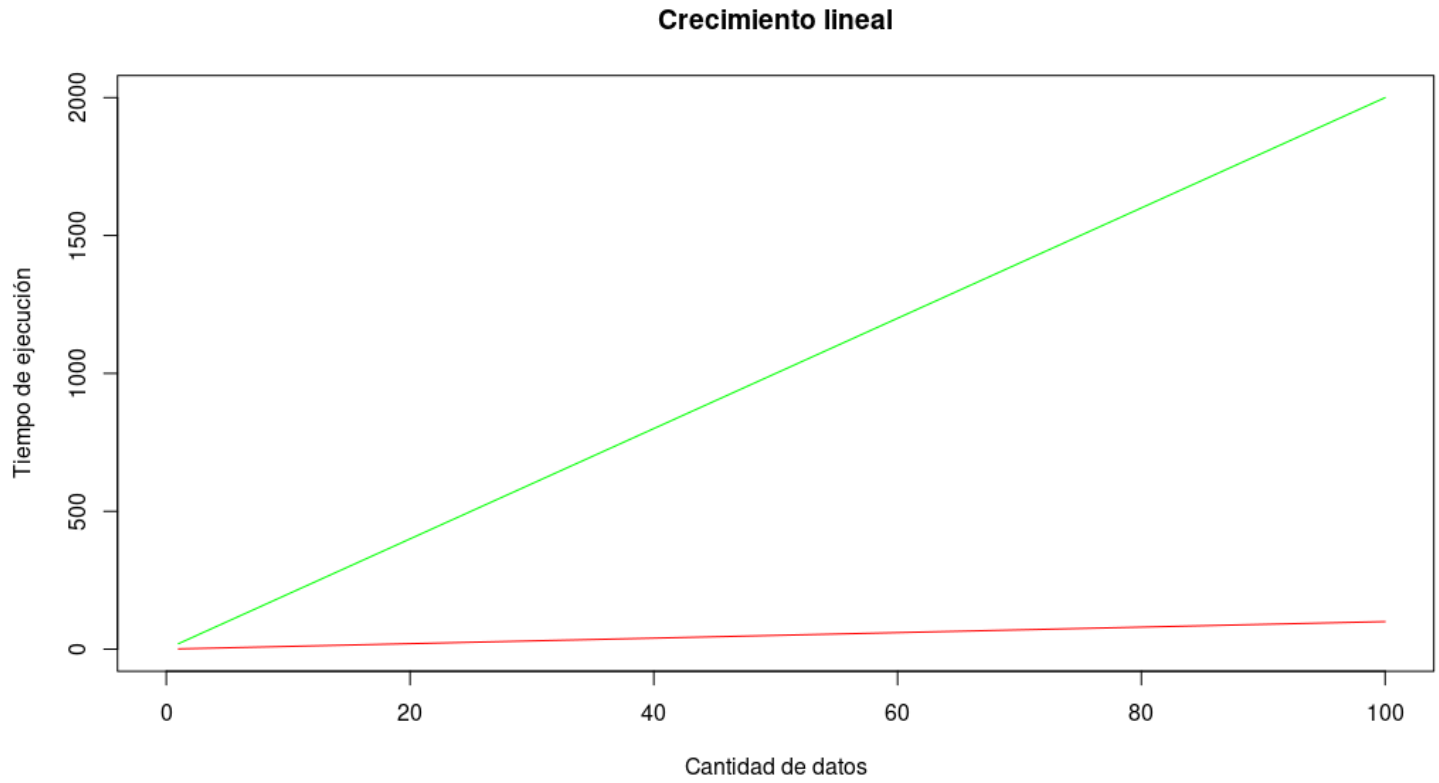
Factores de crecimiento (lineal III)

$f(x) = 100x$



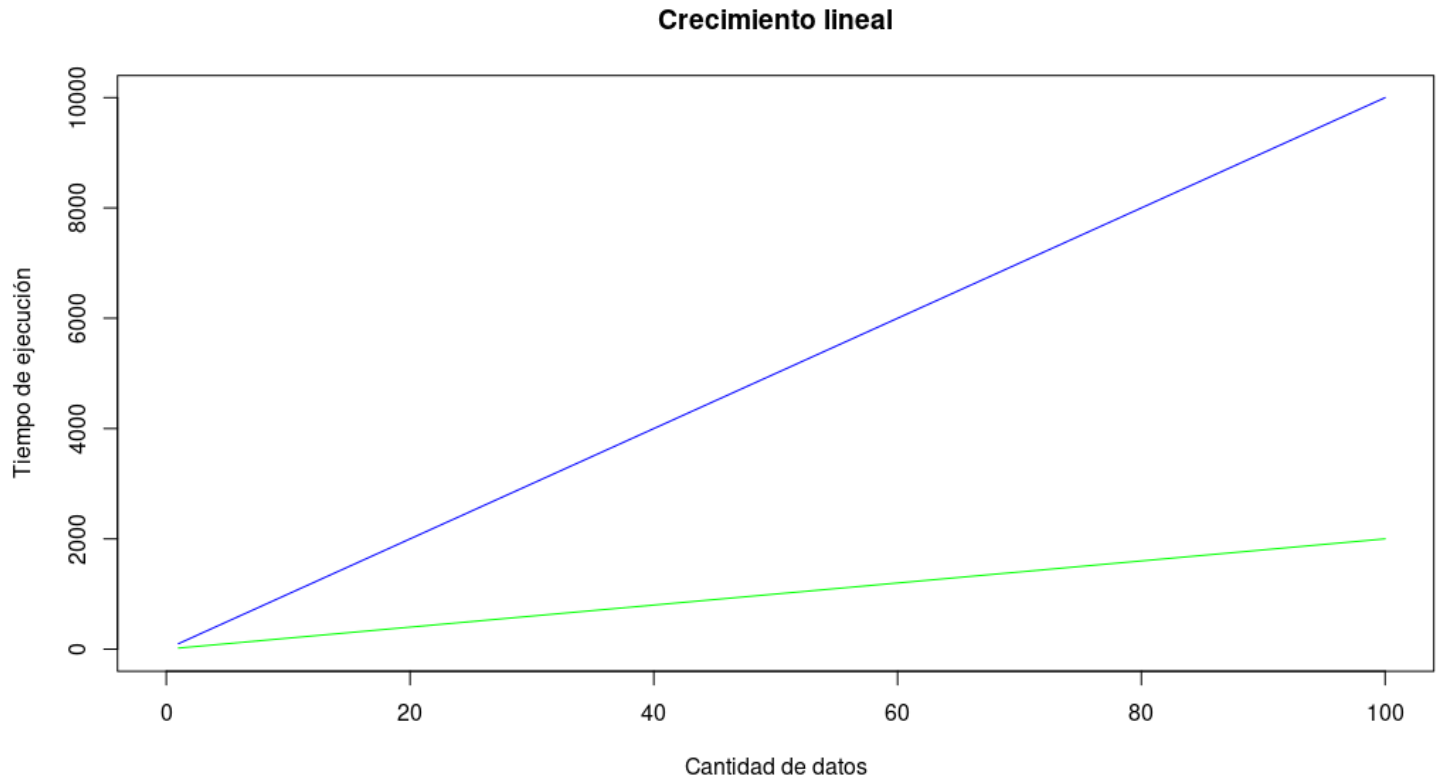
Factores de crecimiento (lineal IV)

$f(x) = x$ vs. $f(x) = 20x$



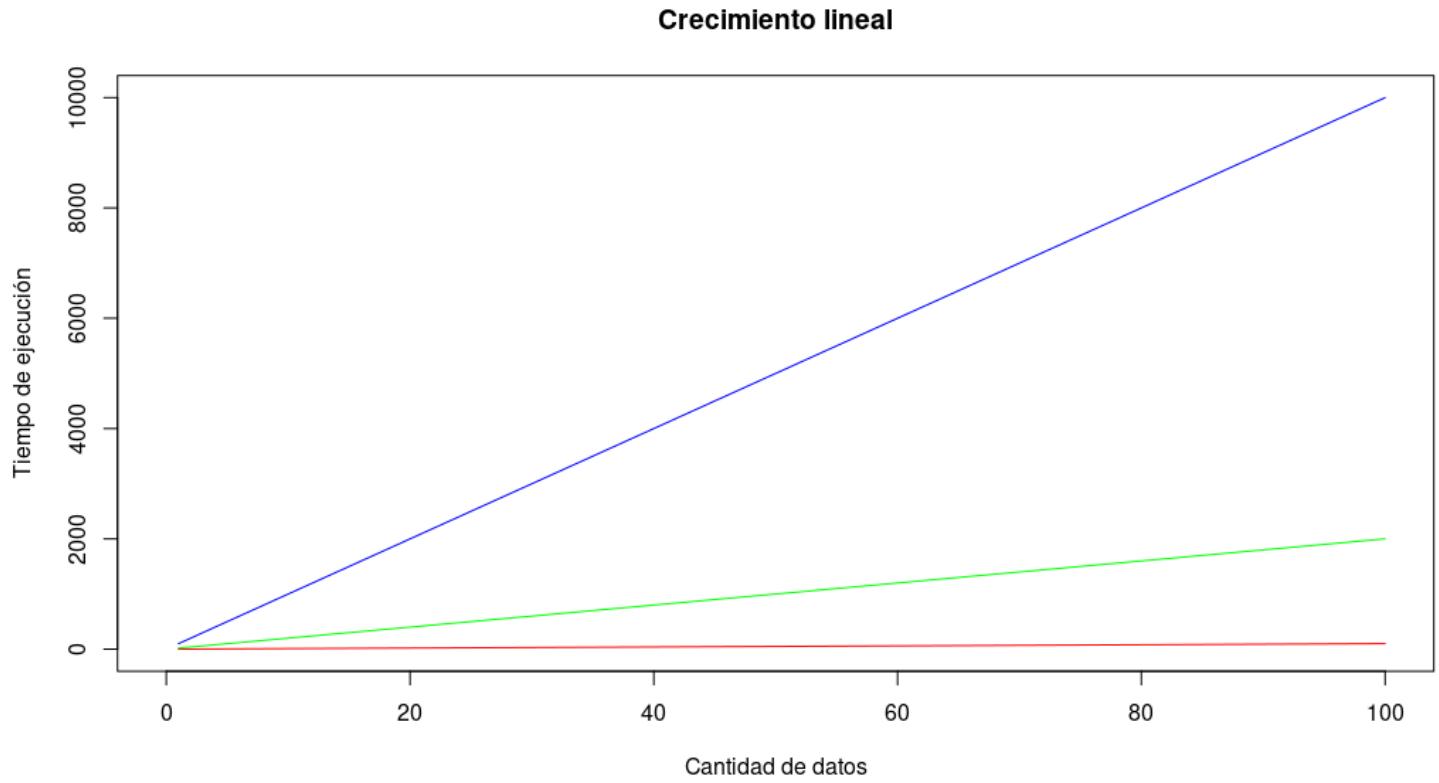
Factores de crecimiento (lineal V)

$f(x) = 20x$ vs. $f(x) = 100x$



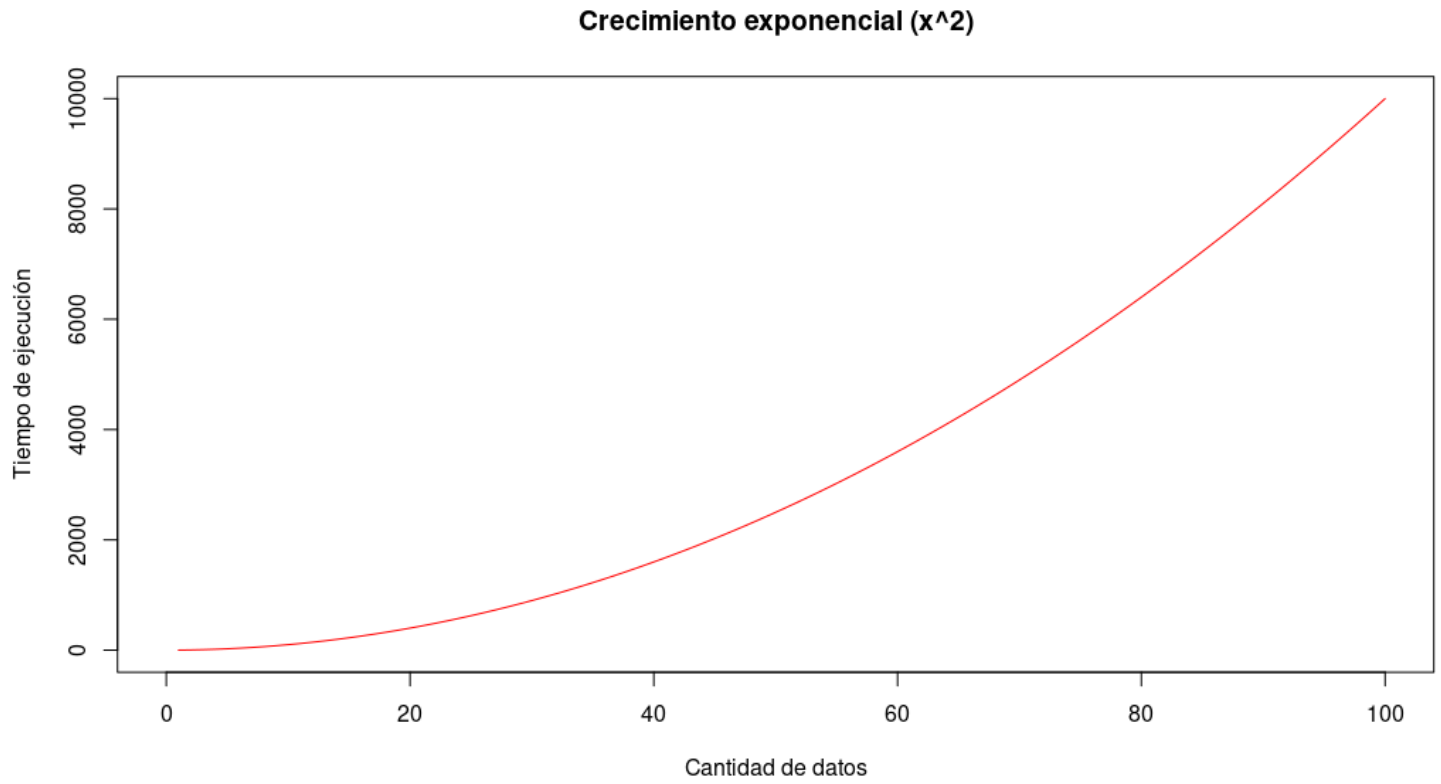
Factores de crecimiento (lineal VI)

$f(x) = x$ vs. $f(x) = 20x$ vs. $f(x) = 100x$



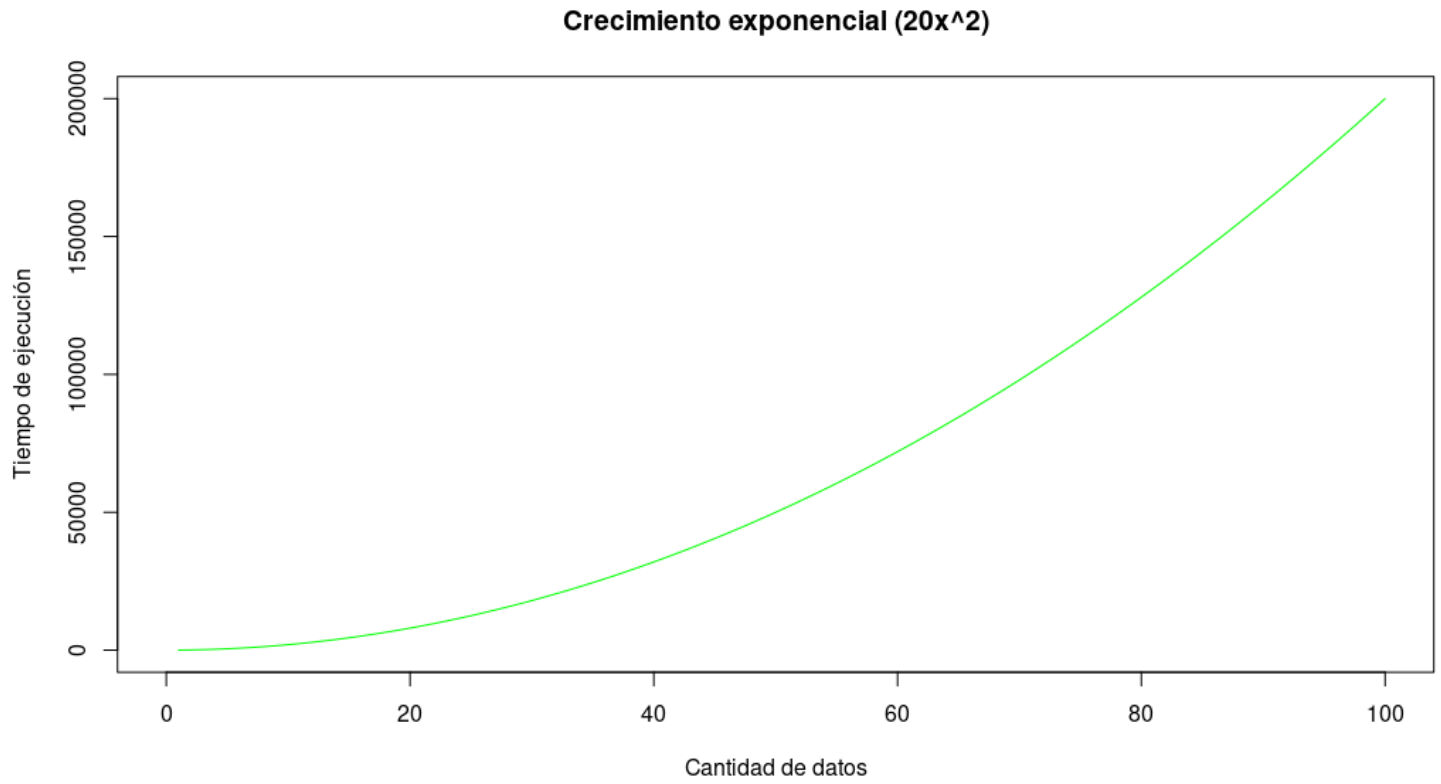
Factores de crecimiento (exponencial)

$f(x) = x^2$



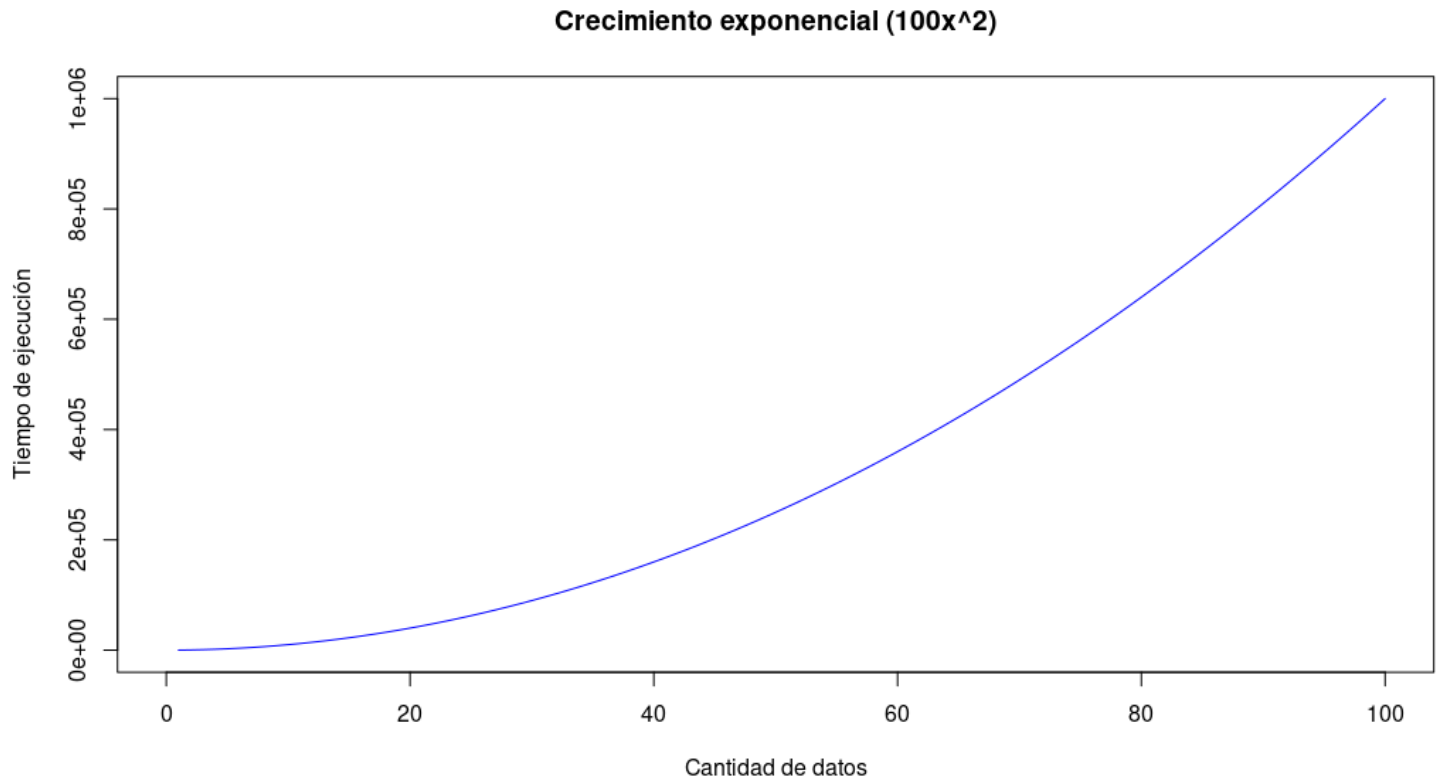
Factores de crecimiento (exponencial)

$f(x) = 20x^2$



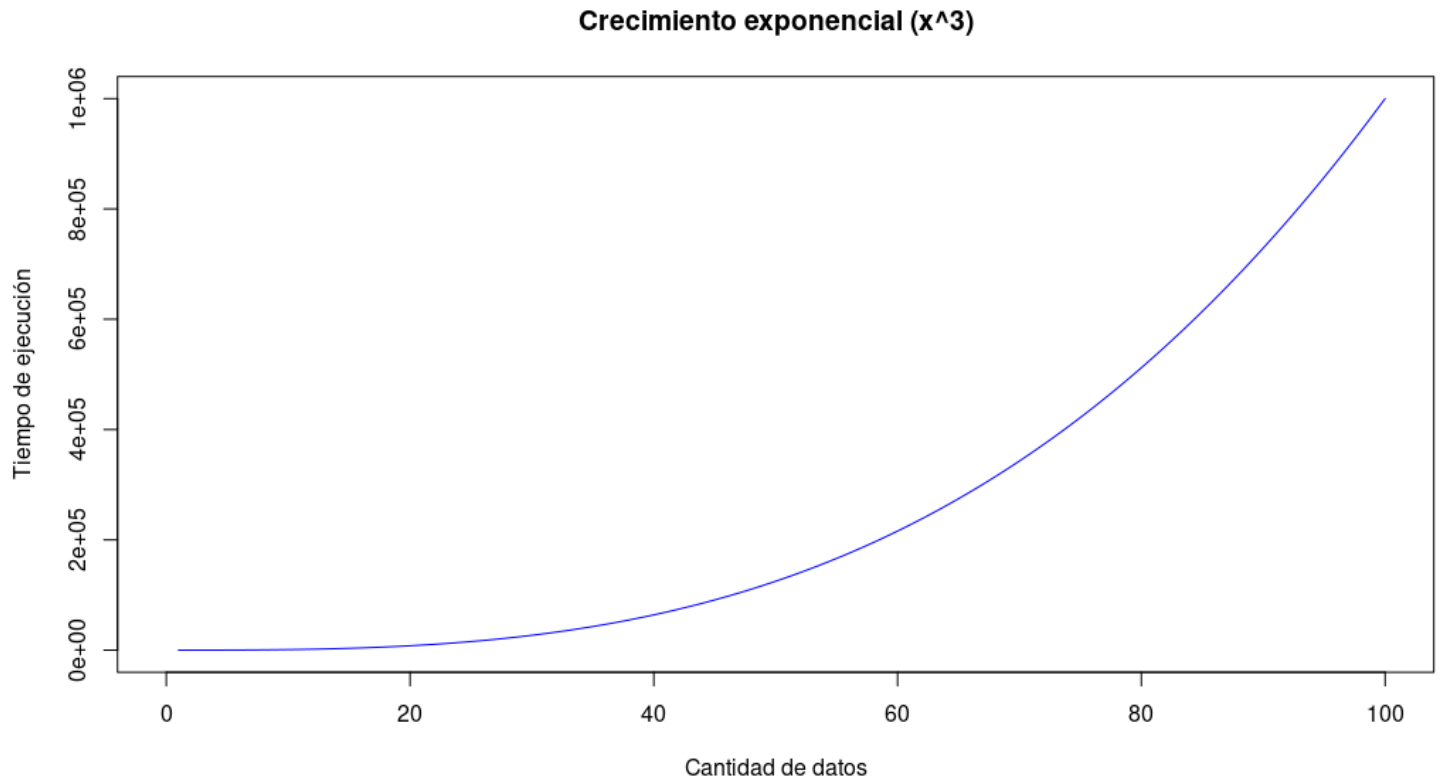
Factores de crecimiento (exponencial)

$f(x) = 100x^2$



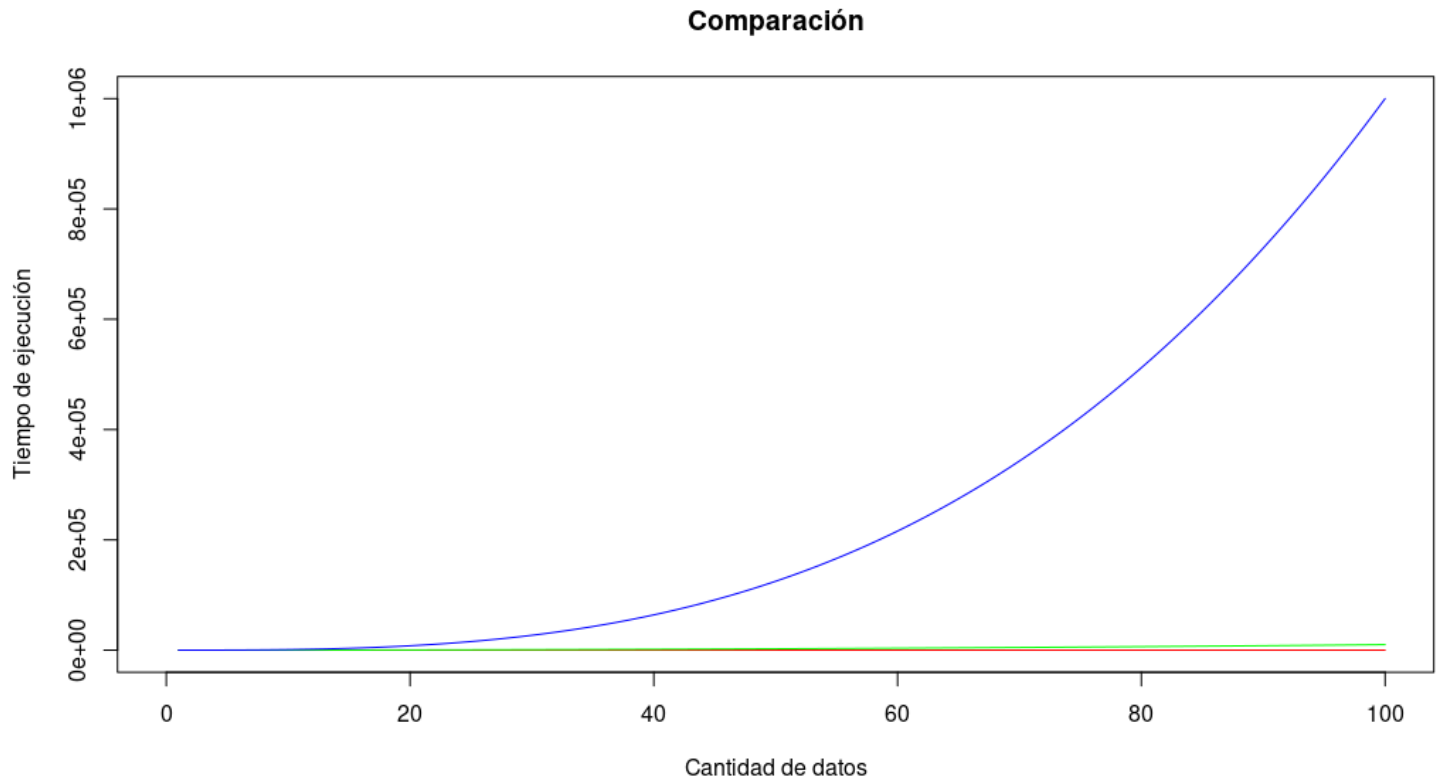
Factores de crecimiento (exponencial)

$f(x) = x^3$



Factores de crecimiento (comparación)

Comparación $f(x) = x$ vs. $f(x) = x^2$ vs. $f(x) = x^3$



Definiciones

Cota superior

Se dice que $T(n)$ es $O(f(n))$ si $\exists\ c, n_0 \in \mathbf{N}$ tales que $T(n) \leq cf(n)$ si $n \geq n_0$.

Si $T(n)$ es $O(f(n))$, entonces se dice que $f(n)$ es una cota superior al factor de crecimiento de $T(n)$.

Definiciones (II)

Cota inferior

Se dice que $T(n)$ es $\Omega(g(n))$ si $\exists\ c \in \mathbf{N}$ tal que $T(n) \geq cg(n)$ para una cantidad infinita de valores n.

En este caso $g(n)$ es una cota inferior a $T(n)$.

En la mayoría de las ocasiones, al evaluar la complejidad de los algoritmos, se encuentran funciones que son polinomios.

En este caso se ignoran las constantes de proporcionalidad tomando en cuenta solamente el término de mayor grado.

Ejemplo

Si $f(n) = 5n^4 + 25n^3 + 7n^2 + 33n + 45$ es cota superior al factor de crecimiento de $T(n)$, el factor $5n^4$ crece mucho más rápido que el resto de los términos.

Se pueden ignorar esos términos y la función $f(n) = 5n^4$ también es cota superior de $T(n)$.

Por definición, $\exists\ c, n_0 \in \mathbf{N}$ tales que $T(n) \leq cf(n)$ si $n \geq n_0$.

Si $f(n) = 5n^4$, se puede definir $g(n) = n^4$ y $c' = 5c$, de tal forma que se cumple: $c', n_0 \in \mathbf{N}$ tales que $T(n) \leq c'g(n)$ si $n \geq n_0$.

Es decir $T(n)$ es $O(g(n))$, en otras palabras $T(n)$ es $O(n^4)$.

Reglas de composición

Se toman en cuenta las propiedades de este tipo de funciones.

Supóngase que se tienen dos programas P_1 y P_2 con tiempos de ejecución $T_1(n)$ y $T_2(n)$ respectivamente, en donde $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$.

1. **Regla de la suma**. Se usa cuando se quiere evaluar la complejidad del algoritmo resultante de ejecutar el algoritmo P_1 seguido del algoritmo P_2.
   $T(P_1 + P_2) = T_1(n) + T_2(n)$ que es $O(\text{máx}(f(n), g(n)))$.
2. **Regla de la multiplicación**. Se usa cuando se quiere evaluar la complejidad del algoritmo resultante de ejecutar el algoritmo P_2 dentro del algoritmo P_1.
   $T(P_1(P_2)) = T_1(n)T_2(n)$ que es $O(f(n)g(n))$.

Evaluación de algoritmos

1. Si el algoritmo es una sucesión de instrucciones se evalúa la complejidad de cada una de ellas y se aplica la *regla de la suma*.
2. Las asignaciones tienen complejidad dependiente de la expresión que determina el valor a asignar. Lo más frecuente es que la complejidad de la expresión sea $O(1)$.
3. La complejidad de las instrucciones condicionales está determinada por la complejidad de la evaluación de la expresión condicional más la complejidad del máximo de las dos opciones a ejecutar. Se aplica la *regla de la suma*.
4. Si hay repeticiones, se evalúa la complejidad del cuerpo de la repetición (lo que está dentro de la repetición) y se multiplica por la cantidad de repeticiones. Se usa la *regla de la multiplicación*.
5. La complejidad de las llamadas a métodos, procedimientos o funciones está determinada por la complejidad del procedimiento o función.

Ejemplo. Multiplicación de matrices

Algoritmo que multiplica dos matrices A y B, cada una de tamaño $n \times n$, dejando el resultado en la matriz C, tambien de tamaño $n \times n$.

```
Repite para i = (1, 2, ..., n)
   Repite para j = (1, 2 , ..., n)
      C[i,j] := 0
      Repite para k = (1, 2, ..., n)
         C[i,j] := C[i,j] + (A[i,k] * B[k,j])
      Termina repetición (k)
   Termina repetición (j)
Termina repetición (i)
```

La complejidad es $O(n^3)$.

Sucesión de Fibonacci

Determina el n-ésimo término de la sucesión de Fibonacci. Toma como parámetro el valor de n.

```
Si n <= 2
Entonces
   Regresa (1)
Si no
   Regresa (Fibonacci(n - 1) + Fibonacci(n - 2))
```

La complejidad es $O(2^n)$.