Complejidad Computacional

Análisis y Diseño de Algoritmos

Dr. Víctor de la Cueva
vcueva@tec.mx

1

Tema: Complejidad Computacional

- Introducción a la Complejidad Computacional.
- Complejidad de algoritmos iterativos.
  - Conceptos matemáticos preliminares.
  - Complejidad Computacional.
  - Notación asintótica.
- Complejidad de algoritmos recursivos.
  - Planteamiento de ecuaciones de recurrencia.
  - Solución de ecuaciones de recurrencia.

2

Criterios para análisis de algoritmos

3

Análisis de Algoritmos

- Los objetivos del análisis son:
  - Mejorarlos, si es posible
  - Seleccionar, entre varios, el que mejor resuelva un problema
- Los criterios más comunes (y los que utilizaremos aquí) para analizar algoritmos son:
  - Corrección
  - Cantidad de trabajo realizado
  - Cantidad de espacio de almacenamiento usado
  - Sencillez y claridad
  - Optimidad

4

Corrección

- Para saber si un algoritmo es correcto se debe demostrar formalmente.
- Se requiere un planteamiento preciso de las características de las entradas con las que se espera que trabaje (llamadas condiciones previas) y del resultado que se espera que produzca con cada entrada (las condiciones posteriores).
- Entonces podemos tratar de demostrar enunciados acerca de las relaciones entre las entradas y las salidas, es decir, si se satisfacen las condiciones previas, las condiciones posteriores se cumplirán cuando el algoritmos termine.

5

Corrección (cont)

- Establecer la corrección de un método puede ser fácil o requerir una larga serie de lemas y teoremas acerca de los objetos con los que se trabaja (grafos, permutaciones, matrices, etc.)
- Los teoremas requeridos pueden pertenecer a otras disciplinas (e.g. álgebra lineal, probabilidad, etc.)
- La corrección de algunos de los algoritmos que veremos no es obvia, se deberá justificar con teoremas.
- Cuando los programas son largos (la mayoría de los sistemas) se recomienda demostrar su corrección por partes, si es que el programa se puede dividir en módulos independientes.

6

Cantidad de trabajo realizado

- ¿Cómo mediremos la cantidad de trabajo realizado por un algoritmo?
  - ¿Tiempo?
  - ¿Número de instrucciones realizadas?
- Se requiere una medida que nos diga algo acerca de la eficiencia del método empleado por un algoritmo, con independencia de:
  - La computadora
  - El lenguaje de programación
  - El programador
  - Detalles de implementación
  - Procesamiento fijo (incremento de índices, cálculo de índices, establecimiento de apuntadores).

7

Operación básica

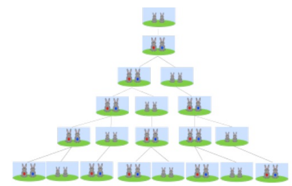
- La opción es usar una “operación básica”.
- Esta operación básica puede ser´:
  - Una iteración (una pasada) de un ciclo.
  - O una específica que sea fundamental para el problema que se estudia.
- Se debe hacer caso omiso de:
  - La inicialización
  - El control de ciclos
  - Y demás tareas de contabilidad
- Sólo se debe contar el número de operaciones básicas que el algoritmo efectúa.

8

Serie de Fibonacci (*Liber Abaci*, 1202)

- «Cierto hombre tenía una pareja de conejos en un lugar cerrado y deseaba saber cuántos se podrían reproducir en un año a partir de la pareja inicial, teniendo en cuenta que de forma natural tienen una pareja en un mes, y que a partir del segundo se empiezan a reproducir» [2].
- Su definición recursiva es:

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$



Fuente: [3]

9

Implementación Directa (*naive implementation*)

Usando Pseudocódigo:

```
fibRec(n):
    if n ≤ 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Fuente: [3]

10

Calculando los número de Fibonacci

- **Entrada**: un entero $n \geq 0$.
- **Salida**: fib(n).

Fuente: [3]

11

Rápido Crecimiento

- **Lema**: $fib(n) \geq 2^{n/2}$, $n \geq 6$.
- **Demostración**: (por inducción):
  - Caso Base: $n = 6$ y $n = 7$.
  - Hipótesis de inducción: $fib(k) \geq 2^{k/2}$, se cumple si $k \leq n$
  - Paso Recursivo: …

2.2.1 pág 6

Fuente: [3]

12

Tiempo de ejecución

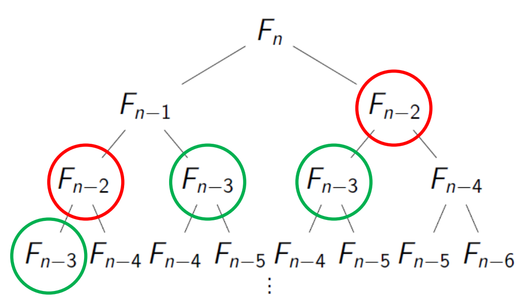
- Sea T(n) el número de líneas de código ejecutadas por la función fibRec(n):
  - Si $n \leq 1$, $T(n) = 2$.
  - Si $n \geq 2$, $T(n) = 3 + T(n-1) + T(n-2)$
- Esto implica que $T(n) \geq fib(n)$
- Y, de acuerdo a la demostración, el fib(n) ¡crece muy rápido!
- $T(100) \approx 1.77*10^{21}$ (1.77 sextillones).
- A 1 GHz tomaría alrededor de 56,000 años.

Fuente: [3]

13

¿Por qué es tan lento?

- Repite muchos de los cálculos una y otra vez:



Fuente: [3]

14

La pregunta clave…

¿Lo podemos hacer mejor?

15

Otro algoritmo

- Imitando el cálculo a mano: sumamos los dos anteriores.
- Eso requiere ir guardando los anteriores → Estructura de Datos
- ¿Se requieren guardar todos?
- ¿Cuál es el número de instrucciones ejecutadas?

Fuente: [3]

16

Nuevo algoritmo

```
fibArreglo(n):
     Crear un arreglo F[0,..,n]
     F[0] ← 0
     F[1] ← 1
     for i = 2 to n:
          F[i] = F[i-1] + F[i-2]
return F[n]
```

- ¿Cuál es el número de líneas de código T(n)?
- T(n) = 2(n-1) + 4 = 2n + 2, T(100) = 202, muy fácil de calcular.

Fuente: [3]

17

Moraleja

El algoritmo correcto, muchas veces combinado con la estructura de datos correcta, hacen la solución más eficiente.

Fuente: [3]

18

Complejidad

- La frase "cantidad de trabajo efectuada por un algoritmo" se acostumbra sustituirla por el término "la complejidad del algoritmo".
- La complejidad implica la cantidad de trabajo realizado, medida según alguna medida de complejidad específica, que en nuestro caso será el número de operaciones básicas realizadas.
- Complejidad no tiene nada que ver con los complicado de un algoritmo (i.e. un algoritmo muy complicado podría tener una complejidad baja).

19

Las entradas

- La cantidad de trabajo realizado no se puede describir con un solo número porque el número de pasos ejecutados no es el mismo para todas las entradas.
- Debemos observar que la cantidad de trabajo efectuado depende, generalmente, del tamaño de las entradas.
- Inclusive, si seleccionamos un solo tamaño de entrada, el número de operaciones podría depender de la naturaleza de las entradas.
- Se requiere entonces una medida del tamaño de las entradas, la cual se selecciona fácilmente, en general.

20

Fibonacci: Análisis del tiempo de ejecución

- 2n + 2 líneas de código. ¿Realmente describe el tiempo de ejecución del algoritmo?
- Línea por línea (hay muchas instrucciones por línea):
  1. Creación del arreglo: depende del sistema de administración de memoria.

  2 y 3. Dos asignaciones

  4. El for: incremento, comparación y ramificación (brinco).
  5. Buscar elemento del arreglo, suma de grandes enteros y asignación.
  6. Buscar elemento del arreglo y regresarlo.

Fuente: [3]

21

Además

- Se debe considerar que el programa correrá en una computadora real por lo que hay que tomar en cuenta:
  - Velocidad de la computadora
  - La arquitectura del sistema
  - El compilador que se está utilizando
  - Detalles de la jerarquía memoria (procesador, RAM, Disco).
- En realidad, sería una tarea muy ardua tomar en cuenta todas estas cosas y, lo más importante, para poder comparar algoritmos todos deberían correr bajo las mismas condiciones.
- En la práctica no es necesario considerar tantos detalles.

Fuente: [3]

22

Idea

- Todos los detalles multiplican el tiempo de ejecución por una "gran" constante.
- De esta forma, se puede pensar en una medida que ignore múltiplos constantes.
- Problema: 1 seg, 1 hora, 1 año, sólo difieren por múltiplos constantes.
- Solución: Considerar tiempos de ejecución asintóticos, que sean escalables al tamaño de la entrada.
- Asintótica: no nos importan las constantes que multiplican o suman a la cantidad, para entradas grandes.

Fuente: [3]

23

Funciones de n

- Es muy común explicar el tiempo de ejecución por medio de una función *f* que depende del tamaño de la entrada *n*.
- La función puede ser de cualquier forma que se requiera pero se utilizan más ciertos tipos:
  - Polinomial
  - Logarítmica
  - Exponencial
  - Factorial
  - Etc.

24

Tiempos de ejecución aproximados

	n	$n \log n$	n^2	2^n
$n = 20$	1 sec	1 sec	1 sec	1 sec
$n = 50$	1 sec	1 sec	1 sec	13 day
$n = 10^2$	1 sec	1 sec	1 sec	$4 \cdot 10^{13}$ year
$n = 10^6$	1 sec	1 sec	17 min	
$n = 10^9$	1 sec	30 sec	30 year	
max n	10^9	$10^{7.5}$	$10^{4.5}$	30

En un segundo

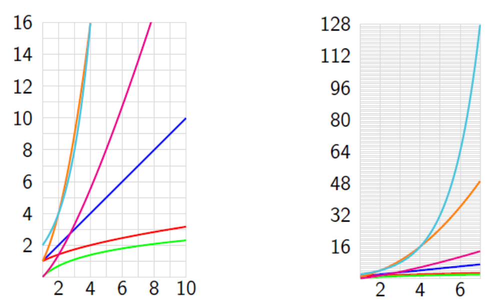
Fuente: [3]

25

Gráficamente

Asintóticamente menor

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$



Fuente: [3]

26

Análisis del peor caso

- Casi siempre se describe el comportamiento de un algoritmo dando su complejidad de “peor caso”, en función del tamaño de la entrada.
- Definición (complejidad de peor caso):
  - Sea D_n el conjunto de entradas de tamaño n para el problema en consideración, y sea I un elemento de D_n. Sea $t(I)$ el número de operaciones básicas que el algoritmo ejecuta con la entrada I. Definimos la función W como:

$$W(n)=max\{t(I) \mid I \in D_n\}$$

  - La función $W(n)$ es la complejidad de peor caso del algoritmo.
  - $W(n)$ es el tamaño máximo de operaciones básicas que el algoritmo ejecuta con cualquier entrada de tamaño n.
- La complejidad de peor caso es valiosa porque proporciona una cota superior del trabajo efectuado por un algoritmo.

27

Análisis del caso promedio

- Podría parecer que una forma más útil y natural de describir el comportamiento de un algoritmo es indicar cuánto trabajo efectúa en promedio.
- En la práctica, algunas entradas se presentan con mayor frecuencia que otras por lo que un promedio ponderado por la probabilidad es más informativo.
- Definición (Complejidad promedio):
  - Sea Pr(I) la probabilidad de que se presente la entrada I. Entonces, el comportamiento promedio del algoritmo se define como:

$$A(n) = \sum_{I \in D_n} \Pr(I)\, t(I)$$

Ej. 35, 36, 37

28

Espacio y Sencillez

- Espacio: Se puede pensar en el espacio como el número de celdas (espacio en el que cabe un dato) usadas en el proceso, el cual depende del tamaño de la entrada n.
- Sencillez: A menudo, la forma más sencilla y directa de resolver un problema no es la más eficiente. No obstante, la sencillez es una característica deseable en un algoritmo porque facilita la verificación de que el algoritmo es correcto, así como su implementación.

29

Optimidad (u Optimalidad)

- Por más brillantes que seamos, no podemos mejorar un algoritmo más allá de cierto punto.
- Def: Decimos que un algoritmo es óptimo (en el peor caso) si ningún otro algoritmo de la clase estudiada efectúa menos operaciones básicas (en el peor caso).
- Cuando se habla de los algoritmos de la clase estudiada no se refiere sólo a los que se han inventado. Estamos hablando de TODOS los algoritmos, incluso lo que todavía no se inventan.
- "Óptimo" no significa "el mejor que se conoce", significa "el mejor posible".

30

Cotas inferiores

- ¿Cómo demostramos que un algoritmo es óptimo?
- No se requiere el análisis de TODOS los algoritmos (incluso lo que no se han inventado).
- Podemos demostrar teoremas que establecen una cota o límite inferior del número de operaciones que se necesitan para resolver un problema.
- Entonces, cualquier algoritmo que ejecute ese número de operaciones será óptimo.
- Cota inferior: Para alguna función F, demostrar un teorema que diga que, para cualquier algoritmo de la clase considerada, existe alguna entrada de tamaño n para la cual el algoritmo debe ejecutar al menos F(n) pasos.

40, 41

31

Funciones de crecimiento

32

Clasificación de funciones por su tasa de crecimiento asintótica

- Puesto que no estamos contando todas las operaciones en un algoritmo, nuestro análisis tiene por fuerza cierta imprecisión.
- Nos conformamos con que el número total de pasos sea aproximadamente proporcional al número de operaciones básicas contadas.
- Esto es suficiente para separar algoritmos que efectúan cantidades de trabajo drásticamente distintas con entradas grandes.
- Ejemplos:
  - $2n$ vs $4.5n$ operaciones básicas $2c_1n$ vs $4.5c_2n$ operaciones en total.
  - $n^3/2$ vs $5n^2$, con n pequeña y con n grande.

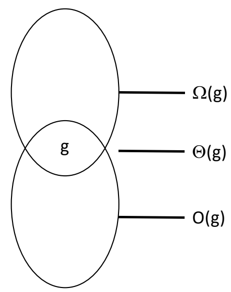
33

Tasa de crecimiento asintótica

- Buscamos una forma de comparar y clasificar funciones que no tomen en cuenta los factores constantes y las entradas pequeñas.
- Llegamos a una clasificación con esas características precisamente si estudiamos la tasa de crecimiento asintótica, el orden asintótico o, simplemente, el orden de las funciones.
- ¿Es razonable hacer caso omiso de las constantes y de las entradas pequeñas?

34

Notación Asintótica



- Hay 3 conjuntos para clasificar a las funciones de acuerdo a su complejidad:
  - $\Omega(g)$ (big Omega): funciones que crecen por lo menos tan rápidamente como g.
  - $\Theta(g)$ (big Theta): funciones que crecen con la misma rapidez que g.
  - O(g) (big O): funciones que no crecen más rápidamente que g.

35

El conjunto O(g) (*Big O Notation*)

- Definición: Sea g una función $\mathbb{Z}^* \rightarrow \mathbb{R}^+$ ($\mathbb{Z}^* \equiv \mathbb{N}$). Entonces, O(g) es el conjunto de las funciones f, también de $\mathbb{Z}^* \rightarrow \mathbb{R}^+$, tal que para alguna constante real $c > 0$ y alguna constante entera $n_0 \geq 0$, $f(n) \leq cg(n)$ para toda $n \geq n_0$.
- Con frecuencia es útil pensar en g como alguna función dada, y en f como la función que estamos analizando.
- Observe que una función f podría estar en O(g) aunque $f(n) > g(n)$ para toda n.
- Lo importante es que f esté acotada por arriba por algún múltiplo constante de g.
- No se considera la relación entre f y g para valores pequeños de n.

36

Ejemplo

$3n^2 + 5n + 2 = O(n^2)$ para $n \geq 1$,

$3n^2 + 5n + 2 \leq 3n^2 + 5n^2 + 2n^2 = 10n^2$

37

Otra técnica

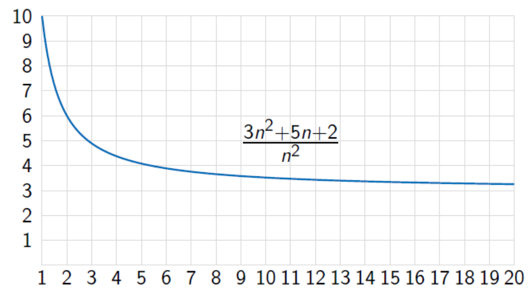
- Hay otra técnica para demostrar que f está en O(g):
- Lema: Una función $f \in O(g)$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, incluido el caso en el que el límite es 0.
- Es decir, si el límite del cociente de f entre g existe y no es ∞, entonces f no crecerá más rápidamente que g. Si el límite es ∞, entonces f sí crece más rápidamente que g.

46,47(15)

38

Ejemplo

$3n^2 + 5n + 2$ tiene el mismo orden de crecimiento que n^2.



Fuente: [3]

39

Notación Big-O

- Se utiliza para reportar tiempos de ejecución de los algoritmos.
- Tiene grandes ventajas:
  - Clarifica el orden de crecimiento
  - Hace más limpia la notación y facilita el álgebra
  - Ignora detalles complicados de la implementación
- Ejemplo:
  - $O(n^2)$ vs. $3n^2 + 5n + 2$
  - $O(n)$ vs. $n + \log_2(n) + \sin(n)$
  - $O(n \log(n))$ vs. $4n \log_2(n) + 7$
- Note que en el último ejemplo, $\log_2(n)$, $\log_3(n)$, $\log_x(n)$, difieren sólo en múltiplos constantes, por lo que la base no necesita especificarse.

Fuente: [3]

40

¡Peligro!, ¡peligro!, ¡peligro!

- Al usar Big-O se pierde información importante sobre los múltiplos constantes.
- Big-O es sólo asintótica.

Fuente: [3]

41

Reglas comunes para usar Big-O

- Los múltiplos constantes pueden ser omitidos:
  - $7n^3 = O(n^3)$ $\frac{n^2}{3} = O(n^2)$
- $n^a \prec n^b$ para $0 < a < b$:
  - $n = O(n^2)$ $\sqrt{n} = O(n)$
- $n^a \prec b^n$ ($a > 0$, $b > 1$):
  - $n^5 = O(\sqrt{2}^n)$ $n^{100} = O(1.1^n)$
- $(\log n)^a \prec n^b$ ($a, b > 0$):
  - $(\log n)^3 = O(\sqrt{n})$ $n \log(n) = O(n^2)$
- Términos pequeños puede omitirse:
  - $n^2 + n = O(n^2)$ $2^n + n^9 = O(2^n)$

Fuente: [3]

42

Big-O en la práctica

Operation	Runtime
create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times
$\quad F[i] \leftarrow F[i-1] + F[i-2]$	$O(1)$
return $F[n]$	$O(1)$
Total:	

$$O(n)+O(1)+O(1)+O(n)\cdot O(1) + O(1) = O(n)$$

Fuente: [3]

43

El conjunto $\Omega(g)$

- La definición de $\Omega(g)$, el conjunto de funciones que crecen al menos tan rápidamente como g, es el dual de la definición $O(g)$.
- <u>Definición</u>: Sea g una función de $\mathbb{Z}^* \rightarrow \mathbb{R}^+$. Entonces, $\Omega(g)$ es el conjunto de funciones f, también de $\mathbb{Z}^* \rightarrow \mathbb{R}^+$, tal que, para alguna constante real $c > 0$ y alguna constante entera $n_0 \geq 0$, $f(n) \geq cg(n)$ para toda $n \geq n_0$.
- <u>Lema</u> (técnica alterna): La función $f \in \Omega(g)$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, incluido el caso en el que el límite es ∞.

44

El conjunto $\Theta(g)$, orden asintótico de g

- Definición: Sea g una función de $\mathbb{Z}^* \to \mathbb{R}^+$. Entonces, $\Theta(g) = O(g) \cap \Omega(g)$, es decir, el conjunto de funciones que están tanto en $O(g)$ como en $\Omega(g)$.
- La forma más común de leer "$f \in \Theta(g)$" es "f es de orden g".
- A menudo se usa la frase "orden asintótico" para expresarnos en forma más definitiva, y también se usa el término "complejidad asintótica".
- Lema: La función $f \in \Theta(g)$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, para alguna constante c tal que $0 < c < \infty$.

45

Importancia del orden asintótico

Algoritmo	1	2	3	4	
Función de tiempo (μs)	$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	2^n
Tamaño de la entrada (n)	*Tiempo para resolver*				
10	0.00033 s	0.0015 s	0.0013 s	0.0034 s	0.001 s
100	0.003 s	0.03 s	0.13 s	3.4 s	$4 \cdot 10^{16}$ años
1,000	0.033 s	0.45 s	13 s	0.94 h	
10,000	0.33 s	6.1 s	22 min	39 días	
100,000	3.3 s	1.3 min	1.5 días	108 años	
Tiempo permitido	*Tamaño de entrada máximo soluble (aprox.)*				
1 segundo	30,000	2,000	280	67	20
1 minuto	1,800,000	82,000	2,200	260	26

Fuente: [1], pp. 49

46

Importancia del orden asintótico (cont)

n	Cray-1 Fortran[a] $3n^3$ nanosegundos	TRS-80 Basic[b] 19,500,000n nanosegundos
10	3 microsegundos	0.2 segundo
100	3 milisegundos	2.0 segundos
1,000	3 segundos	20.0 segundos
2.500	50 segundos	50.0 segundos
10,000	49 minutos	3.2 minutos
1,000,000	95 años	5.4 horas

[a] Cray-1 es una marca comercial de Cray Research, Inc.
[b] TRS-80 es una marca comercial de Tandy Corporation.

Número de pasos realizados para una entrada de tamaño n $f(n)$	Tamaño máximo posible de la entrada s	Tamaño máximo posible de la entrada para tiempos t o cantidades de tiempo muy superiores s_{nuevo}
$\lg n$	s_1	s_1^t
n	s_2	$t\, s_2$
n^2	s_3	$\sqrt{t}\, s_3$
2^n	s_4	$s_4 + \lg t$

Fuente: [1], pp. 50

47

Propiedades de O, Ω y Θ

- Para todas las propiedades, suponemos que f, g, h: $\mathbb{N} \rightarrow \mathbb{R}^*$.
- Lema: Si $f \in O(g)$ y $g \in O(h)$, entonces $f \in O(h)$; es decir, O es transitivo. También lo son Ω, Θ, o y ω.
- Demostración: ... pág 51
- Lema:
  1. $f \in O(g)$ si y sólo si $g \in \Omega(f)$
  2. Si $f \in \Theta(g)$, entonces $g \in \Theta(f)$
  3. Θ define una relación de equivalencia sobre las funciones. Cada conjunto $\Theta(f)$ es una clase de equivalencia, que llamamos clase de complejidad.
  4. $O(f+g) = O(máx(f, g))$. Se cumplen ecuaciones similares para Ω y Θ.

48

Orden asintótico de sumatorias comunes

- Teorema: Sea d una constante no negativa y sea r una constante positiva distinta de 1.
  1. La sumatoria de una serie polinómica incrementa el exponente en 1 ($\sum_{i=1}^{n} i^d$ está en $\Theta(n^{d+1})$).
  2. La sumatoria de una serie geométrica ($\sum_{i=a}^{b} r^i$) está en Θ de su término más grande. La regla es válida para $r > 0$ y $r \neq 1$.
  3. La sumatoria de una serie logarítmica está en Θ del número de términos multiplicado por el logaritmo del término más grande ($\sum_{i=1}^{n} \log(i)$ está en $\Theta(n\log(n))$ y al hablar de orden asintótico la base del logaritmo no importa).
  4. La sumatoria de una seria polinómica-logarítmica, es decir, una sumatoria de la forma $\sum_{i=1}^{n} i^d \log(i)$, está en $\Theta(n^{d+1}\log(n))$.

49

Referencias

- [1] S. Baase y A. Van Gelder. Algoritmos Computacionales: introducción al análisis y diseño. 3ª edición. Addison Wesley (2002).
- [2] Wikipedia. https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci. Consultado el 5-Ago-2016.
- [3] D. Kane. Algorithmic Toolbox. MOOC Course UCSD-Coursera. Specialization: Data Structures and Algorithms. June 2016. https://www.coursera.org/learn/algorithmic-toolbox.

50