

Programación y Algoritmos

Facultad de Matemática y Computación
Universidad de La Habana

Lic. Eric Barcelo, Msc. Rocio Ortiz, Lic. Cristian Vigoa

Índice

1. Introducción a los Algoritmos	2
2. Tema 1: Introducción a la Complejidad Algorítmica	3

Nota Importante

Todos los ejemplos de código de esta conferencia se encuentran en el notebook asociado a ella

1. Introducción a los Algoritmos

Un algoritmo no es algo que nació con las computadoras. Las computadoras nacieron porque ya existían algoritmos. Mucho antes de la informática, los seres humanos llevaban siglos —incluso milenios— diseñando y ejecutando algoritmos para resolver problemas. Cada vez que alguien establecía un procedimiento sistemático para obtener un resultado, estaba construyendo un algoritmo, aunque no usara esa palabra. De hecho, la propia palabra algoritmo proviene de Al-Juarismi (siglo IX), cuyos métodos sistemáticos para resolver ecuaciones eran esencialmente algoritmos algebraicos.

Conexión con otras asignaturas

En la carrera ya hemos visto algunos algoritmos como el de derivación de una función o el de Euclides para la búsqueda del máximo común divisor.

¿Qué es un algoritmo?

Informalmente podemos decir que un algoritmo es una receta para resolver un problema específico. Como receta en fin tiene que constar con una serie de pasos claros y ordenados que si se siguen correctamente te deben llevar de un estado inicial a un estado final, que es no debe variar independientemente de quien lo ejecute.

Definición 1.1. Un **algoritmo** es un conjunto finito de reglas que especifican una secuencia de operaciones para resolver un tipo de problema específico.

Debe satisfacer 5 características importantes :

- (Finitud) Debe terminar después de un número finito de pasos.
- (Precisión) Cada paso debe estar definido de manera exacta y no ambigua.
- (Entrada) Tiene cero o más entradas.
- (Salida) Produce al menos una salida.
- (Efectividad) Cada operación debe ser lo suficientemente básica como para poder realizarse en un tiempo finito.

Fuente: The Art of Computer Programming

Ejemplo 1.1. Un algoritmo con 0 entradas puede ser el que te sume y devuelva el resultado de la suma de los números del 1 al 100.

Observación 1.1. Es interesante e importante destacar que la definición que se propone no depende de computadoras.

Conexión con otras asignaturas

Una visión matemática: Sean X el conjunto de entradas y Y el conjunto de salidas. Un algoritmo implementa una función

$$f : X \rightarrow Y$$

tal que, para todo $x \in X$, el valor $f(x)$ es computable mediante un proceso finito.

Ejemplo 1.2. Un algoritmo para determinar si un número es primo.

```

1 def es_primo(n : int):
2     if n < 2:
3         return False
4     for i in range(2, int(n**0.5) + 1):
5         if n % i == 0:
6             return False
7     return True

```

Ejercicio 1.1. Menciona 10 algoritmos que hayas estudiado previamente (en cualquier asignatura). Para cada uno de ellos, describe de manera clara y ordenada la secuencia de pasos que lo componen (su “receta”).

El diseño y análisis de algoritmos son pilares fundamentales en la matemática moderna. Cuando formulamos un algoritmo desde el punto de vista teórico, una de las primeras preocupaciones es demostrar su corrección: asegurarnos de que, bajo las hipótesis del modelo matemático, siempre produce una solución válida para el problema planteado.

Sin embargo, en la práctica, muchos problemas reales involucran instancias tan grandes —ya sea por la cantidad de datos, la dimensión o la complejidad de su estructura— que resultan imposibles de resolver manualmente en un tiempo razonable. La llegada de la computación digital permitió enfrentar este tipo de situaciones mediante la implementación de algoritmos en máquinas capaces de procesar enormes volúmenes de información a gran velocidad.

Pero pasar del plano teórico al computacional no es un simple trámite: aparecen nuevos desafíos. Por un lado, las computadoras trabajan con sistemas numéricos finitos, lo que introduce limitaciones inevitables como errores de redondeo, desbordamientos, pérdida de precisión o inestabilidad numérica. Estos llamados errores numéricos pueden hacer que el resultado computado difiera, a veces de forma significativa, de la solución exacta del modelo matemático ideal.

Por otro lado, incluso si el algoritmo es matemáticamente correcto, su implementación puede no serlo. Un error en la programación, una mala interpretación de la especificación o un detalle lógico mal resuelto pueden invalidar el resultado. Por eso, la verificación y validación del código —mediante pruebas, depuración o análisis formal— se vuelven etapas esenciales del proceso.

Finalmente, no basta con que un algoritmo sea correcto: también debe ser eficiente. Como el tiempo de ejecución y la memoria disponible son recursos limitados, es fundamental estudiar su complejidad computacional, tanto desde un punto de vista teórico (por ejemplo, mediante la notación O) como a través de pruebas experimentales. Este análisis nos permite prever cómo se comportará el algoritmo cuando el tamaño del problema crezca y evaluar si es realmente viable en la práctica.

2. Tema 1: Introducción a la Complejidad Algorítmica

Cuando pensamos en cuánto tarda un algoritmo en ejecutarse, es natural considerar que una forma de medirlo consiste en calcular el tiempo explícito que toma cada algoritmo al ejecutarse. Esta será nuestra primera aproximación para determinar cuán eficiente es un algoritmo.

Para ello, analizaremos el comportamiento de los algoritmos en los siguientes problemas:

1. ¿Cuál es el valor de la suma de dos números A y B ?
2. ¿Cuál es el valor de la suma de los números del 1 al $n - 1$?

3. Determinar el valor de n^2 .
4. Encontrar el subarreglo de suma máxima en una lista de n números.

Función que nos permitirá medir el tiempo de los algoritmos, esta manera de medir la eficiencia del algoritmo la llamaremos "timimg":

```

1 import time
2
3 def medir_tiempo(func, *args):
4     inicio = time.time()
5     resultado = func(*args)
6     tiempo_transcurrido = time.time() - inicio
7     return tiempo_transcurrido

```

Analicemos ahora el comportamiento del tiempo para el problema 1.

```

1 Resumen de tiempos (segundos):
2 Caso 1: 1 + 2 -> 0.00000000 s
3 Caso 2: 10 + 20 -> 0.00000000 s
4 Caso 3: 100 + 200 -> 0.00000000 s
5 Caso 4: 1000 + 2000 -> 0.00000000 s
6 Caso 5: 10000 + 20000 -> 0.00000000 s
7 Caso 6: 1000000 + 2000000 -> 0.00000000 s
8 Caso 7: 1000000000 + 2000000000 -> 0.00000000 s

```

Es razonable intuir que el tiempo para realizar una suma es prácticamente despreciable. Este hecho será útil cuando introduzcamos otras formas de analizar algoritmos.

Para el problema 2, el algoritmo propuesto consiste en almacenar en una variable el resultado acumulado al iterar desde 1 hasta $n - 1$. Veamos su rendimiento en algunos casos:

```

1 Resumen de tiempos (segundos):
2 Caso 1: Suma del 1 al 10 -> 0.00000000 s
3 Caso 2: Suma del 1 al 20 -> 0.00000000 s
4 Caso 3: Suma del 1 al 100 -> 0.00000000 s
5 Caso 4: Suma del 1 al 1000 -> 0.00000000 s
6 Caso 5: Suma del 1 al 10000 -> 0.00000000 s
7 Caso 6: Suma del 1 al 1000000 -> 0.13128829 s
8 Caso 7: Suma del 1 al 10000000 -> 7.18329406 s
9 Caso 8: Suma del 1 al 20000000 -> 12.08572149 s
10 Caso 9: Suma del 1 al 70000000 -> 53.65519309 s
11 Caso 10: Suma del 1 al 100000000 -> 65.82658482 s

```

Se observa que, al aumentar el valor de n , el tiempo también crece. Esto nos lleva a considerar que el tiempo depende del tamaño de la entrada. Surgen entonces varias preguntas: ¿dependerá de algún otro factor? ¿Podremos predecir cuánto tardará para $n = 1\ 000\ 000\ 000$ sin ejecutarlo?

Para el problema 3, el algoritmo propuesto consiste en sumar n veces el número n que lo obtenemos sumando n números 1. A continuación, se muestran algunos tiempos de ejecución:

```

1 Resumen de tiempos (segundos):
2 Caso 1: Cuadrado de 10 -> 0.00000000 s
3 Caso 2: Cuadrado de 20 -> 0.00000000 s

```

```

4 Caso 3: Cuadrado de 100 -> 0.00000000 s
5 Caso 4: Cuadrado de 1000 -> 0.07154536 s
6 Caso 5: Cuadrado de 2000 -> 0.19181824 s
7 Caso 6: Cuadrado de 10000 -> 4.82975125 s
8 Caso 7: Cuadrado de 20000 -> 17.72740793 s
9 Caso 8: Cuadrado de 100000 -> 503.21836758 s

```

Los problemas 2 y 3 nos permiten comprender que el tiempo de ejecución depende del tamaño de la entrada. Sin embargo, el método de *timing* no nos permite predecir con claridad el tiempo para entradas más grandes sin realizar la ejecución.

No utilizaremos el método de *timing* para el problema 4, ya que presenta varias limitaciones para analizar la eficiencia de un algoritmo:

- No permite predecir el tiempo sin ejecutar el algoritmo.
- Depende de la computadora en la que se ejecute (puedes comprobarlo ejecutando el notebook asociado en tu dispositivo).
- Depende del lenguaje de programación utilizado.

Nuestra siguiente forma de calcular la eficiencia de un algoritmo superará estas limitaciones: utilizaremos el método de *conteo de operaciones*.

Asumiremos que las siguientes operaciones básicas toman tiempo constante (aunque dicha constante no necesariamente sea la misma para todas; trabajaremos bajo el modelo RAM de costo uniforme, que asigna costo 1 a cada una):

- Operaciones matemáticas.
- Comparaciones.
- Asignaciones de variables.
- Acceso a posiciones de memoria.
- Llamadas recursivas.
- Instrucción `return`.

Contemos ahora el número de operaciones básica.

Definición 2.1. La cantidad de operaciones de un algoritmo se denotara por $T(n_1, n_2, \dots, n_m)$ donde los n_i son los parámetros de los que depende el algoritmo.

Problema 1

```

1 def suma(A,B):
2     return A+B

```

Operaciones fuera de ciclos

- Acceso a A y B : 2
- Suma $A + B$: 1
- `return` : 1

Expresión total

$$T(A, B) = 4$$

Problema 2

```

1 def suma_n(n):
2     res=0
3     for i in range(1,n):
4         res+=i
5     return res

```

Operaciones fuera del ciclo

- Asignación $res = 0$: 1
- `return` : 1

Costo: 2

Estructura del ciclo

- Inicialización del índice i : 1
- Comparaciones : n
- Incrementos del indice i : $n - 1$

Costo estructural:

$$n + (n - 1) + 1 = 2n$$

Cuerpo del ciclo (se ejecuta $n - 1$ veces)

En cada iteración:

- Acceso a res : 1
- Acceso a i : 1
- Suma : 1
- Asignación : 1

Total por iteración: 4

Costo total del cuerpo:

$$4(n - 1)$$

Expresión total

$$T(n) = 2 + (2n) + 4(n - 1)$$

Problema 3

```

1 def cuadrado(n):
2     res=0
3     for i in range(n):
4         for j in range(n):
5             res+=1
6     return res

```

Operaciones fuera de los ciclos

- Asignación $res = 0 : 1$
- $return : 1$

Costo: 2

Estructura del ciclo externo

- Inicialización de $i : 1$
- Comparaciones : $n + 1$
- Incrementos de $i : n$

Costo estructural externo:

$$1 + (n + 1) + n = 2n + 2$$

Estructura del ciclo interno (por cada iteración externa)

- Inicialización de $j : 1$
- Comparaciones : $n + 1$
- Incrementos de $j : n$

Costo estructural interno por iteración externa:

$$1 + (n + 1) + n = 2n + 2$$

Cuerpo del ciclo interno

En cada ejecución:

- Acceso a *res* : 1
- Suma : 1
- Asignación : 1

Total por ejecución: 3

Costo total del cuerpo por iteración externa:

$$3n$$

Costo total del ciclo interno

$$(2n + 2) + 3n = 5n + 2$$

Costo total considerando las *n* iteraciones externas

$$n(5n + 2) = 5n^2 + 2n$$

Expresión total

$$T(n) = 2 + (2n + 2) + 5n^2 + 2n$$

$$T(n) = 5n^2 + 4n + 4$$

Problema 4

El algoritmo para solucionar el problema 4, que se propone es calcular la suma de todos los posibles subarreglos y quedarnos con la mayor.

Sea $n = \text{len}(\text{arr})$.

```

1 def max_subarray_suma(arr):
2     max_sum = arr[0]
3     for i in range(len(arr)):
4         sum_actual = 0
5         for j in range(i, len(arr)):
6             sum_actual += arr[j]
7             max_sum = max(max_sum, sum_actual)
8     return max_sum

```

Operaciones fuera de los ciclos

- Acceso a `arr[0]` : 1
- Asignación a `max_sum` : 1
- `return` : 1

Costo: 3

Estructura del ciclo externo

- Inicialización de *i*: 1

- Comparaciones : $n + 1$
- Incrementos : n
- Inicialización de la variable suma-actual : n

Costo estructural externo:

$$2n + 3$$

Estructura del ciclo interno (por cada iteración externa)

- Inicialización de $j : 1$
- Comparaciones : $n - i + 1$
- Incrementos de $j : n - i$

Costo estructural del ciclo interno:

$$\sum_{i=0}^{n-1} ((n - i) + (n - i + 1) + 1) = 2n + n(n + 1) = n^2 + 3n$$

Cuerpo del ciclo interno

En cada iteración interna de tamaño $n - i$:

- Acceso a `arr[j] : 1`
- Suma : 1
- Asignación : 1
- Comparación : 1
- Asignación : 1

Total por iteración: 5

Número total de iteraciones internas:

$$\sum_{i=0}^{n-1} (n - i) = \frac{n(n + 1)}{2}$$

Costo total del cuerpo:

$$\frac{5}{2}n(n + 1)$$

Expresión total

$$T(n) = 3 + (2n + 3) + (n^2 + 3n) + \frac{5}{2}n(n + 1)$$

$$T(n) = \frac{7}{2}n^2 + \frac{17}{2}n + 6$$

Como se puede observar, el cálculo del número de operaciones puede llegar a ser bastante tedioso.

Nota Importante

Una computadora puede realizar aproximadamente 10^9 operaciones por segundo.

Dado el cálculo del número de operaciones y el conocimiento aproximado de cuántas operaciones realiza una computadora, podemos estimar de antemano cuánto tiempo tardará un algoritmo sin necesidad de ejecutarlo. Además, está claro que el número de operaciones no depende ni del lenguaje de programación ni de la computadora utilizada. De este modo, se superan las debilidades del método de *timing*, aunque no es trivial utilizar este enfoque.

Ejercicio 2.1. Propón nuevas implementaciones de los algoritmos de los Problemas 2–4 que varíen el número de operaciones.

Ejercicio 2.2. Para cada función $f(n)$ y tiempo t de la siguiente tabla, determine el mayor tamaño n de un problema que puede resolverse en tiempo t , suponiendo que el algoritmo tarda $f(n)$ microsegundos.

Tiempos disponibles:

$$1 \text{ segundo} = 10^6 \text{ microsegundos}$$

1 minuto

1 hora

1 día

1 mes (30 días)

1 año

1 siglo

Funciones a considerar:

$$\log n, \quad \sqrt{n}, \quad n, \quad n \log n, \quad n^2, \quad n^3, \quad 2^n, \quad n!$$