Física Computacional
Tarea 2

Pedro Porras Flores
Efraín Ossmar Díaz Pérez

11 de septiembre del 2025

Instrucciones

Resuelva los siguientes ejercicios implementando soluciones en Python. Utilice variables adecuadas, estructuras de control de flujo y funciones. Incluya comentarios en su código y muestre los resultados obtenidos.

1. Crecimiento de una grid de punto flotante y límite de memoria

Un programa numérico trabaja con una grid (matriz) de tamaño $n \times m$, donde cada elemento se almacena como un número de punto flotante en doble precisión (64 bits = 8 bytes).

En cada iteración, el programa duplica una de las dimensiones de la grilla, es decir, si la grilla inicial es de $n \times m$, en el siguiente paso puede pasar a $2n \times m$ o a $n \times 2m$. El nuevo arreglo se carga completo en la memoria RAM.

- La grilla inicial tiene tamaño

$$n = 2^2, \quad m = 2^3.$$

- El equipo cuenta con **16 GB de memoria RAM** disponibles exclusivamente para este programa.

(a) Calcule el tamaño en memoria de la grilla inicial y estime cuántos pasos de iteración puede ejecutar el programa antes de agotar la memoria RAM.

(b) Suponga que en cada iteración, además de almacenar la grilla nueva, el programa debe mantener en memoria la grilla anterior. ¿Cómo cambia el número máximo de pasos posibles?

(c) Considere que el sistema operativo y librerías básicas ocupan **2 GB de RAM** y que la computadora reserva además el 10 % de la memoria total como buffer del sistema. Recalcule el número máximo de pasos bajo estas condiciones más realistas.

(d) Escriba un programa en `Python` que simule el crecimiento de la grilla. El programa debe:

- Calcular el tamaño en memoria (en bytes, MB y GB) de la grilla en cada paso.
- Detener la simulación cuando la memoria total supere el límite disponible.
- Imprimir en qué paso se alcanzó el límite y cuál era el tamaño de la grilla en ese momento.

Pruebe su programa bajo las condiciones de los incisos (a), (b) y (c). Compare los resultados obtenidos con sus cálculos manuales.

2. Evaluar un polinomio de grado n de la forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

puede ser costoso si calculamos cada potencia de x por separado.

La **regla de Horner** permite reorganizar el polinomio de manera que sólo se requieren multiplicaciones y sumas sucesivas:

$$P(x) = a_0 + x\big(a_1 + x\big(a_2 + \cdots + x(a_{n-1} + x a_n)\ldots\big)\big).$$

Por ejemplo, para un polinomio cúbico

$$P(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0,$$

la forma de Horner es

$$P(x) = a_0 + x\big(a_1 + x(a_2 + x a_3)\big).$$

Un proyectil es lanzado con una velocidad inicial de 50 m/s en un ángulo de 45° respecto a la horizontal. La fuerza de resistencia del aire

es proporcional al cuadrado de la velocidad: $F_{\text{aire}} = -k \cdot v^2$, donde $k = 0{,}01$ kg/m.

La ecuación para la altura vertical del proyectil en función del tiempo puede aproximarse por el polinomio:

$$y(t) = -4{,}905t^4 + 19{,}62t^3 + 35{,}355t^2 - 0{,}981t$$

donde t está en segundos y $y(t)$ en metros.

a) Implementar la **regla de Horner** para evaluar polinomios eficientemente.

b) Calcular la altura del proyectil en los tiempos: $t = 0{,}5, 1{,}0, 1{,}5, 2{,}0, 2{,}5$ segundos.

c) Determinar el tiempo en que el proyectil alcanza su altura máxima (encontrar las raíces de la derivada usando evaluación polinómica).

d) Calcular la altura máxima alcanzada.

e) Graficar la trayectoria completa del proyectil usando matplotlib.

3. Escriba funciones que calcule las funciones $\sin(x)$, $\cos(x)$, $\tan(x)$, $log(x)$ y utilizando sus series de Taylor. El cálculo debe detenerse cuando el último término de la serie sea menor que la precisión de la máquina. Además los nombres de las funciones deben ser `sin_taylor()`, `cos_taylor()`, etc.

4. La norma euclidiana de un vector n-dimensional $\mathbf{x}$ se define como

$$\|\mathbf{x}\|_2 = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}.$$

Implemente una rutina robusta para calcular esta cantidad para cualquier vector de entrada $\mathbf{x}$. Su rutina debe evitar problemas de *overflow* y de *underflow*. Compare tanto la precisión como el rendimiento de su rutina robusta con una implementación ingenua más sencilla.

a) ¿Puede construir un vector que produzca resultados significativamente distintos entre las dos rutinas?

b) ¿Cuánto rendimiento sacrifica la rutina robusta?

5. Aritmética de intervalos en Python En muchas aplicaciones de física computacional es importante controlar los errores de redondeo introducidos por la aritmética de punto flotante. Una técnica consiste en usar **aritmética de intervalos**, en la cual cada número real se representa como un intervalo cerrado $[a, b]$ que garantiza que el valor real se encuentra dentro de dicho rango.

Dado dos intervalos $X = [a, b]$ y $Y = [c, d]$ (con $a \leq b$, $c \leq d$), las operaciones básicas se definen de la siguiente manera:

- **Suma:**
$$X + Y = [a + c,\ b + d].$$
- **Resta:**
$$X - Y = [a - d,\ b - c].$$
- **Multiplicación:**
$$X \cdot Y = [\text{mín}(ac, ad, bc, bd),\ \text{máx}(ac, ad, bc, bd)].$$
- **División:** si $0 \notin [c, d]$, entonces
$$\frac{X}{Y} = [\text{mín}(a/c, a/d, b/c, b/d),\ \text{máx}(a/c, a/d, b/c, b/d)].$$

a) Implemente en Python una clase `Interval` que modele un intervalo cerrado $[a, b]$, con $a \leq b$.

b) Programe los métodos que implementen las operaciones anteriores: suma, resta, multiplicación y división.

c) Para asegurar que los intervalos siempre contengan el resultado verdadero, utilice **redondeo hacia abajo** para los extremos izquierdos y **redondeo hacia arriba** para los extremos derechos. En Python puede usarse el módulo `decimal` con los modos de redondeo `ROUND_FLOOR` y `ROUND_CEILING`.

d) Pruebe su implementación con operaciones donde el error de redondeo

se haga evidente, por ejemplo:

- Sumar diez veces el número 0,1.
- Calcular el cociente 1/3.

6. En el análisis de algoritmos nos interesa describir cómo crece el tiempo de ejecución $T(n)$ de un programa cuando aumenta el tamaño de la entrada n. Para ello se utilizan notaciones asintóticas:

- **O grande (O):** representa una **cota superior**. Indica que, en el peor de los casos, el algoritmo no crecerá más rápido que una función $f(n)$, salvo una constante. Por ejemplo, $T(n) = 3n^2+2n+5$ está en $O(n^2)$, no confundir con la notación $\mathcal{O}(x^n)$ que se usa para indicar el orden de truncamiento en las series de Taylor.
- **Ω grande (Ω):** representa una **cota inferior**. Indica que, en el mejor de los casos, el algoritmo no puede crecer más lento que $f(n)$, salvo una constante. Por ejemplo, si siempre se requieren al menos $n/2$ pasos, entonces $T(n) = \Omega(n)$.
- **Θ grande (Θ):** representa una **cota ajustada**. Significa que $T(n)$ está acotado superior e inferiormente por la misma función $f(n)$, es decir:

  $$\exists c_1, c_2 > 0 : \quad c_1 f(n) \leq T(n) \leq c_2 f(n), \quad \text{para } n \text{ suficientemente grande.}$$

En resumen:

$$\Omega(f(n)) \ \leq \ T(n) \ \leq \ O(f(n)), \quad \text{y si ambas coinciden, entonces } T(n) = \Theta(f(n)).$$

Implemente dos algoritmos distintos para resolver el mismo problema: *buscar un número en una lista de enteros.*

a) Búsqueda lineal: recorra la lista elemento por elemento hasta encontrar el valor buscado o llegar al final.
   1. Analice[1] cuántas operaciones realiza en el peor caso.
   2. Estime su complejidad en notación O, Θ y Ω.

b) Búsqueda binaria: ordene la lista (puede usar el método `.sort()` o `sorted()`) y divídala a la mitad en cada paso. Hint: revise si el número que busca es mayor o menor al número donde está parado

[1] Al decir "analice" se refiere a escribir en texto la lógica de su razonamiento. No es necesario que sea muy extenso

1. Analice cuántas operaciones realiza en el peor caso.
2. Estime su complejidad en notación O, Θ y Ω.

c) Experimentos computacionales:

1. Genere listas de diferentes tamaños ($n = 10^3, 10^4, 10^5, 10^6$).
2. Mida el tiempo de ejecución de cada algoritmo en Python.
3. Grafique los tiempos de búsqueda en función de n.

d) Responda lo siguiente

1. ¿Coinciden los tiempos medidos con las complejidades teóricas esperadas?
2. ¿Por qué la búsqueda binaria muestra un crecimiento mucho más lento en comparación con la búsqueda lineal?