Algoritmos y Estructuras de Datos

Introducción a la materia
Complejidad, contratos e invariantes

Segundo trimestre de 2025

Maestría en Inteligencia Artificial

Universidad de San Andrés

Presentación de la materia

Modelo de cómputo

Tiempo de ejecución y complejidad asintótica

Contratos e invariantes

Presentación de la materia

Cursada

- Horario: jueves de 19:00 a 22:00.
- 12 semanas — del 19 de junio al 4 de septiembre.

Modalidad de evaluación

- **Ejercicios entregables** (algunas clases).
  La calificación es binaria: aceptable / reentregar.
  Al menos el 75% de las entregas deben estar aceptables.
- **Examen integrador**: jueves 28 de agosto.
  La calificación es numérica; determina la nota final.
- **Recuperatorio del integrador**: jueves 4 de septiembre.

Nociones básicas

¿Qué es un **algoritmo**?

1. Método para llevar a cabo una tarea.
2. Procedimiento para operar con datos o información.
3. Descripción ejecutable de una solución a un problema.
4. Programa (que termina para todo dato de entrada).
5. ...

Ejemplo

Un algoritmo para invertir una palabra p.

- Sea p una palabra de longitud n.
- Escribamos $p = \ell_0 \dots \ell_{n-1}$, donde $\ell_0, \dots, \ell_{n-1}$ son las letras.
- Repetir desde $i = 0$ hasta $i = \lfloor n/2 \rfloor$:
  - Intercambiar la letra ℓ_i con la letra ℓ_{n-1-i}.

Nociones básicas

¿Qué es una **estructura de datos**?

1. Manera de almacenar datos o representar información.
2. Conjunto de operaciones que manipulan datos de acuerdo con una “política”. Se implementan por medio de algoritmos.
3. ...

Ejemplo

Una estructura de datos para registrar invitaciones. Operaciones:

1. **registrar** un invitado por DNI;
2. **determinar** si alguien fue invitado, dado su DNI.

La información se representa usando una lista de DNIs.

1. Para **registrar** un invitado:
   agregar el DNI del invitado al final de la lista.
2. Para **determinar** si alguien fue invitado:
   buscar el DNI en la lista haciendo una búsqueda lineal.

Objetivos de la materia

Estudiar **algoritmos y estructuras de datos fundamentales**:

- búsqueda, ordenamiento, secuencias, árboles, grafos, colas, pilas, diccionarios, ...

Introducir a técnicas de **diseño de algoritmos**:

- *Divide & Conquer*, programación dinámica, *backtracking*, ...

Introducir a técnicas de **análisis de algoritmos**:

- complejidad temporal y espacial en peor caso, notación "O", ...

Usaremos PYTHON como lenguaje de programación.

Cronograma semana a semana

1. Introducción a la materia.
2. *Divide & conquer*.
3. Árboles de búsqueda.
4. *Backtracking*.
5. Ordenamiento.
6. Colas de prioridad.
7. Algoritmos y estructuras sobre strings.
8. Recorridos sobre grafos.
9. Programación dinámica.
10. Estructuras de consulta sobre conjuntos.
11. Examen integrador.
12. Camino mínimo en grafos + Recuperatorio del integrador.

Bibliografía

1. P. Brass. *Advanced Data Structures*. Cambridge books online. Cambridge University Press, 2008.
2. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. Mit Electrical Engineering and Computer Science. MIT Press, 2001.
3. S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw Hill, 2008.
4. Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, tercera edición, 1997.
5. R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Pearson Education, 2013.

Presentación de la materia

Modelo de cómputo

Tiempo de ejecución y complejidad asintótica

Contratos e invariantes

Modelos de cómputo

¿Cómo resolver problemas eficientemente?

Usando menos **recursos**, en la medida de lo posible.

(Hay diversos tipos de recursos: tiempo de ejecución, memoria utilizada, consumo de energía, cantidad de consultas a un servicio externo, ...).

Generalmente nos concentraremos en el **tiempo** y el **espacio**.

Modelos de cómputo y operaciones elementales

Para analizar la eficiencia de un algoritmo, es necesario fijar un **modelo de cómputo** que permita cuantificar el uso de recursos.

Hay muchos modelos de cómputo posibles.

Estableceremos un conjunto de **operaciones elementales** (OE), indicando cuál es el costo de cada una.

Modelo RAM

La memoria está conformada por muchos **arreglos** de n celdas.
Cada **celda** puede contener:

- o bien un valor atómico (entero, booleano, ...),
- o bien una referencia a otra celda.

Crear un arreglo de n celdas cuesta n unidades de tiempo y espacio.
Cada parámetro o variable ocupa 1 unidad de espacio.
Las siguientes OE cuestan 1 unidad de tiempo:

1. Acceder a la i-ésima celda de un arreglo.
2. Modificar la i-ésima celda de un arreglo.
3. Determinar el tamaño de un arreglo.
4. Hacer *una* operación entre valores atómicos.
5. Hacer *un* llamado a una función.
6. Hacer *una* asignación a una variable.
7. Ejecutar *una* instrucción de control (`if`, `while`, ...).

Ejemplo

Consideremos el siguiente algoritmo para invertir una lista:

```
def invertir(l):
    n = len(l)
    for i in range(n // 2):
        l[i], l[n - 1 - i] = l[n - 1 - i], l[i]
```

¿Cuál el el costo en espacio y en tiempo de ejecución, en función del tamaño de la lista `l`, de acuerdo con el modelo RAM?

¿Cómo se corresponde el costo en el modelo con el tiempo real?

Presentación de la materia

Modelo de cómputo

Tiempo de ejecución y complejidad asintótica

Contratos e invariantes

Tiempo de ejecución de un algoritmo

Sea A un algoritmo sobre datos de entrada de un conjunto X.
Suponemos que $A(x)$ termina para todo dato $x \in X$.

Definición (Tiempo de ejecución de un algoritmo)

El *tiempo de ejecución* de A es la función:

$$T_{\mathrm{RAM}} : X \to \mathbb{N}$$

tal que $T_{\mathrm{RAM}}(x)$ es el número de unidades de tiempo que se requieren para ejecutar $A(x)$ en el modelo RAM.

Tiempo de ejecución de un algoritmo

Calculemos $T_{\mathrm{RAM}} : X \to \mathbb{N}$ para el siguiente algoritmo.
Asumimos que $X =$ (Listas de enteros) $\times$ (Enteros).

```
def aparece(a, x):
    for i in range(len(a)):
        if a[i] == x:
            return True
    return False
```

El costo puede depender del valor de los datos de entrada.

En particular, en este algoritmo:

1. Mejor caso: el elemento aparece al inicio de la lista.
   El algoritmo termina en la primera iteración.
2. Peor caso: el elemento no aparece en la lista.
   El costo es proporcional a la longitud de la lista.

Tiempo de ejecución de un algoritmo

Sea A un algoritmo sobre datos de entrada de un conjunto X.
Notamos $|x|$ al tamaño de un dato de entrada $x \in X$.
Notamos X_n a los datos de tamaño n:

$$X_n = \{x \in X \mid |x| = n\}$$

Def. (Tiempo de ejecución en caso peor/mejor/promedio)

$$T_{\text{peor}} : \mathbb{N} \to \mathbb{N}$$
$$T_{\text{peor}}(n) = \max_{x \in X_n} T_{\text{RAM}}(x)$$

$$T_{\text{mejor}} : \mathbb{N} \to \mathbb{N}$$
$$T_{\text{mejor}}(n) = \min_{x \in X_n} T_{\text{RAM}}(x)$$

$$T_{\text{promedio}} : \mathbb{N} \to \mathbb{N}$$
$$T_{\text{promedio}}(n) = \frac{\sum_{x \in X_n} T_{\text{RAM}}(x)}{\# X_n}$$

Complejidad asintótica

Sea $T : \mathbb{N} \to \mathbb{N}$ el tiempo de ejecución de A (en algún caso).

Es difícil e inconveniente calcular T de forma **exacta**.

Vamos a evaluar el comportamiento **asintótico** de T, cuando el tamaño de la entrada tiende a infinito.

Es decir, nos interesa entender el **crecimiento** de T.

Ejemplos motivacionales

$T(n) = 10$	es preferible a	$T(n) = n$
$T(n) = n$	es comparable a	$T(n) = n + 10$
$T(n) = 10n$	es preferible a	$T(n) = n^2$
$T(n) = n^2$	es comparable a	$T(n) = 10n^2$

Complejidad asintótica

Definición (Notación "O")

Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$.
Definimos $O(f)$ como un conjunto de funciones $\mathbb{N} \to \mathbb{R}_{\geq 0}$:

$$g \in O(f) \text{ si y sólo si } \exists n_0 \in \mathbb{N}. \exists c > 0. \forall n \geq n_0.\ g(n) \leq c\, f(n)$$

Propiedades

1. $f \in O(f)$
2. $O(f) = O(g)$ si y sólo si $f \in O(g)$ y $g \in O(f)$.
3. Si $f_1 \in O(g)$ y $f_2 \in O(g)$ entonces $f_1 + f_2 \in O(g)$.
4. $O(f_1 + f_2) = O(\max\{f_1, f_2\})$
5. Si $f_1 \in O(g_1)$ y $f_2 \in O(g_2)$ entonces $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.

Complejidad asintótica

Ejemplo

1. $O(1) = O(5) \subsetneq O(n)$
2. $O(n) = O(3n + 10) \subsetneq O(n^2)$

Más propiedades

1. $O(c\,f(n)) = O(f(n))$
2. $O(a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0) = O(x^n)$ si $a_n > 0$.
3. $O(\log_b n) = O(\log_c n)$

Inclusiones comunes

$$O(1) \subsetneq O(\log n) \subsetneq O(\sqrt{n}) \subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n^2)$$

$$\subsetneq O(n^3) \ldots \subsetneq O(n^p) \subsetneq O(n^{p+1}) \subsetneq O(2^n) \subsetneq O(3^n) \subsetneq O(n!)$$

Ejemplo – complejidad de *selection sort*

Calculemos la complejidad temporal asintótica en peor caso del siguiente algoritmo.

Entrada: una lista de enteros.
Salida: una permutación ordenada de la lista original.

```
def selection_sort(a):
    n = len(a)
    for i in range(n):
        for j in range(i + 1, n):
            if a[i] > a[j]:
                a[i], a[j] = a[j], a[i]
```

- ¿Cuál es la complejidad temporal en mejor caso?
- ¿Cuál es la complejidad espacial en peor caso?

Presentación de la materia

Modelo de cómputo

Tiempo de ejecución y complejidad asintótica

Contratos e invariantes

Contratos

Sea A un algoritmo que recibe datos de un conjunto X, *termina para toda entrada* y devuelve datos de un conjunto Y.

Definición (Contrato, corrección)

Un **contrato** está dado por:

1. Una propiedad $P \subseteq X$, llamada la *precondición*.
2. Una propiedad $Q \subseteq X \times Y$, llamada la *postcondición*.

El algoritmo A es **correcto** con respecto a un contrato (P, Q) si:

- dados datos de entrada que cumplen la precondición,
- produce datos de salida que cumplen la postcondición.

Más precisamente:

$$\forall x \in X. (x \in P \implies (x, A(x)) \in Q)$$

Contratos – ejemplo

Consideremos el algoritmo:

```
def f(a):
    return a[len(a) - 1]
```

¿Es correcto con respecto al siguiente contrato?

- Pre: la lista a no es vacía.
- Post: el resultado es el máximo elemento de a.

¿Es correcto con respecto a este segundo contrato?

- Pre: la lista a no es vacía y está ordenada de menor a mayor.
- Post: el resultado es el máximo elemento de a.

Invariante de un ciclo

Sea A un algoritmo que transforma datos de un conjunto X.
Nos interesa entender el comportamiento de un ciclo:

```
while cond(x):
    x = A(x)
```

Definición (Invariante de un ciclo)

Decimos que una propiedad $I \subseteq X$ es un **invariante** del ciclo si A preserva dicha propiedad. Es decir:

$$\forall x \in X. (I(x) \implies I(A(x)))$$

Teorema

Supongamos que se dan las siguientes condiciones:

1. Vale $I(x)$ para los datos iniciales $x \in X$.
2. I es un invariante del ciclo.
3. El ciclo termina, arrojando un resultado $y \in Y$.

Entonces vale $I(y)$.

Invariante de un ciclo – ejemplo

Recordemos el algoritmo de ordenamiento *selection sort*:

```
def selection_sort(a):
    n = len(a)
    for i in range(n):
        for j in range(i + 1, n):
            if a[i] > a[j]:
                a[i], a[j] = a[j], a[i]
```

Queremos demostrar que es correcto con respecto al contrato:

- Pre: a es una lista de enteros.
- Post: a contiene una permutación ordenada de la lista original.

¿Cuál es el invariante del ciclo externo?

¿Cuál es el invariante del ciclo interno?

Invariante de una estructura de datos

Una estructura de datos está dada por operaciones $O_1, \ldots, O_n$ implementadas como algoritmos que operan sobre datos $x \in X$.

Definición (Invariante de una estructura de datos)

Una propiedad $I \subseteq X$ es un **invariante** de la estructura de datos si todas las operaciones $O_1, \ldots, O_n$ la preservan. Es decir:

$$\forall i \in \{1, ...n\}. \forall x \in X. (I(x) \implies I(A_i(x)))$$

Teorema

Supongamos que se dan las siguientes condiciones:

1. Vale $I(x)$ para el estado inicial $x \in X$.
2. I es un invariante de la estructura de datos.
3. $y \in X$ es un estado alcanzado aplicando las operaciones.

Entonces vale $I(y)$.

Invariante de una estructura de datos – ejemplo

El invariante sirve para garantizar la coherencia de los datos.
Establece "derechos" y "obligaciones" para las operaciones.

Por ejemplo, diseñemos un tipo de datos con la siguiente interfaz:

- **avanzarSegundero**(): avanza la aguja un segundo.
- **segundoActual**(): devuelve el segundo actual.

Dos implementaciones posibles

```
def avanzarSegundero(self):
    self._n += 1


def segundoActual(self):
    return self._n % 60
```

```
def avanzarSegundero(self):
    self._n += 1
    if self._n == 60:
        self._n = 0

def segundoActual(self):
    return self._n
```

¿Cuál es el invariante de cada una?

Invariante de una estructura de datos – ejemplo

El invariante sirve para expresar redundancia entre los datos.

Por ejemplo, diseñemos un tipo de datos con la siguiente interfaz:

- **registrar**(`nota`): registra una nota.
- **ver**(`i`): devuelve la *i*-ésima nota.
- **promedio**(): devuelve el promedio de las notas en $O(1)$.

¿Qué estructura permitiría garantizar la complejidad pedida?
¿Con qué invariante?

```python
def registrar(self, nota):
    self._notas.append(nota)
    self._suma += nota

def ver(self, i):
    return self._notas[i]

def promedio(self):
    return self._suma / len(self._notas)
```