2Algoritmos.pdf

Anónimo

Algoritmia y Estructuras de Datos Avanzadas

2º Grado en Ingeniería Informática

Escuela Politécnica Superior
Universidad Autónoma de Madrid

2️⃣

Algoritmos

1. Algoritmos básicos

¿Definición del algoritmo? → Muchas pero ninguna muy precisa (la definición no es importante)

En los algoritmos hay 3 tipos de bloques de construcción:

- **Secuenciales**: Bloques de secuencias básicas que se ejecutan una detrás de otras (código normal).
- **Selecciones**: La ejecución se separa en diferentes bloques de acuerdo a alguna condición.
    - `if condition:, elif condition:, else:`
- **Bucles**: Sentencias que se repiten mientras se cumpla una condición.
    - `while condition:` , `for`

2. Eficiencia de algoritmos

Para que un algoritmo sea lo más eficiente tiene que:

- Ser **correcto**
- Usar la **menor memoria** posible
- Ser lo más **rápido** posible

2.1. ¿Cómo se miden los tiempos de ejecución?

Se miden por tiempos abstractos que consiste en contar las operaciones clave de un algoritmo. En un algoritmo iterativo, nos fijamos en la operación clave interna.

Ejemplo 1: Multiplicación de matrices (de un código malo y costoso)

```
def matrix_multiplication(m_1, m_2):
  """ ..."""
  n_rows, n_interm, n_columns = \
  m_1.shape[0], m_2.shape[0], m_2.shape[1]

  m_product = np.zeros( (n_rows, n_columns) )

  for p in range(n_rows):
    for q in range(n_columns):
      for r in range(n_interm):
        m_product[p, q] += m_1[p, r] * m_2[r, q]

  return m_product
```

- Operación clave: `m_product[p, q] += m_1[p, r] * m_2[r, q]`
- ¿Cuántas veces se ejecuta? → Suponiendo matrices cuadradas de tamaño N: $N * N * N = N^3$

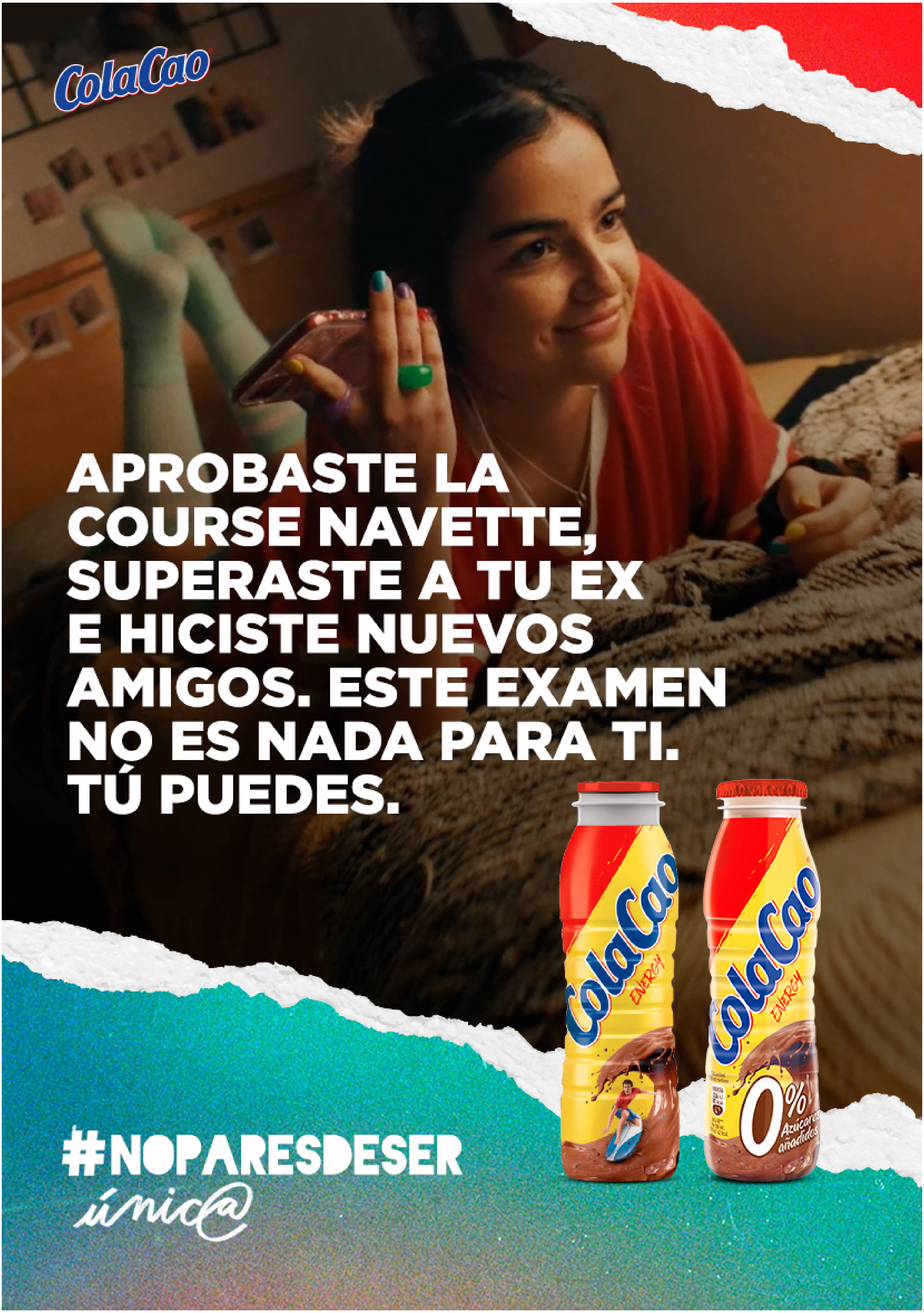
Ejemplo 2: Búsqueda lineal

```
def linear_search(key, l_ints):
  """..."""
  for i, val in enumerate(l_ints):
    if val == key: return i
  return None
```

- Operación clave: `if val == key`
- ¿Cuántas veces se ejecuta? → Encontrar l_ints[0] requiere solo 1 operación. Pero encontrar l_ints[-1] requiere N = `len(l_ints)` operaciones clave.
  - Para búsquedas con éxito: $1 \leq n_{LS}^{e}(N) \leq N$
  - Para búsquedas sin éxito: $n_{LS}^{e}(N) = N$

Pero, no siempre podemos dar estimaciones tan precisas. Por ello, introducimos nueva notación, para poder ser más precisos con aquellos casos que no se puede con el método actual.

2.2. Notaciones o, O, Θ


ColaCao
APROBASTE LA COURSE NAVETTE, SUPERASTE A TU EX E HICISTE NUEVOS AMIGOS. ESTE EXAMEN NO ES NADA PARA TI. TÚ PUEDES.
ColaCao
ENERGY
ColaCao
ENERGY
0%
Azúcares añadidos
#NOPARESDESER
únic@

o

Dado un par f, g, decimos que $f = o(g)$ si $\frac{f(N)}{g(N)} \rightarrow 0$ cuando $N \rightarrow \infty$

O

$f = O(g)$ si podemos encontrar C y N_C t.q. $f(N) \leq C * g(N)$ si $N \geq N_C$

Θ

$f = \Theta(g)$ si $f = O(g)$ y $g = O(f)$

2.3. `timeit` (Magic Command)

`timeit` se utiliza para medir los tiempos de ejecución.

```
timings = %timeit -n 1000 -r 10 -o [k**2 for k in range(10)]

print('\n', dir(timings)[-9 : ], '\n')
print("all_runs", np.array(timings.all_runs).round(3))
print("repeat", timings.repeat)
print("loops", timings.loops)
print("compile_time", timings.compile_time)
print("average_time", timings.average)
print("best_time", timings.best)
print("worst_time", timings.worst)
```

El tiempo estimado de ejecución no es 100% preciso, pero si estimado. Esto se ve en las prácticas.

3. Diseño de algoritmos

3.1. Escritura de un algoritmo

Escribir algoritmos correctamente es complejo y requiere de imaginación, creatividad y experiencia. Aún así, podemos usar algunas técnicas de diseño generales. Veremos 3:

1. **Algoritmos codiciosos**
2. **Divide y venceras** $\rightarrow$ **Recusión**
3. **Programación dinámica**

Veamos como serían los diseños mediante ejemplos:

Ejemplo 1: El problema del cambio

> Suponed que trabajáis en un supermercado y que vuestros clientes quieren el cambio en el menor número de monedas posible.

¿Cómo procedemos? (Algoritmo codicioso)→ En cada iteración devolvemos la moneda más grande posible (sin pasarse del cambio). Si hay devolver 3,44€ devolveríamos: 2€ + 1€ + 2*0,20€ + 2*0,02€.

```
l_coin_values = [1, 2, 5, 10, 20, 50, 100, 200]
def change(c, l_coin_values):
  d_change = {}
  for coin in sorted(l_coin_values)[ : : -1]:
    d_change[coin] = c // coin
    c = c % coin
  return d_change
```

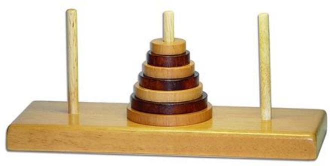
PERO, este algoritmo no funciona en todos los casos. Si hay que devolver 7 maravedís con monedas [1, 3, 4, 5]:

- El algoritmo devolvería 5 + 2*1 (3 monedas)
- La respuesta correcta es: 3 + 4 (2 monedas)

> ℹ️ Los **Algoritmos Codiciosos** son muy intuitivos, pero a veces dan resultados erróneos.

Ejemplo 2: Las Torres de Hanoi

> Nos dan un conjunto de 64 discos dorados de diferentes tamaños apilados en un poste A en tamaños decrecientes, y dos postes vacíos B, C. El objetivo es pasar los discos del



> poste A al B, pero nunca un disco grande puede estar debajo de uno pequeño.

¿Cómo lo resolvemos? → Con 3 discos la solución es sencilla, pero cuantos más discos, más se complica. Sin embargo existe una solución **recursiva** fácil de programar:

1. Mueve los primeros N − 1 discos desde el poste A a C usando B como poste auxiliar.
2. Mueve el disco restante de A a B.
3. Mueve los N − 1 discos restantes de C a B usando A como el poste auxiliar.

```
def hanoi(n_disks, a=1, b=2, c=3):
  assert n_disks > 0, "n_disks at least 1"
  if n_disks == 1:
    print("move disk from %d to %d" % (a, b))
  else:
    hanoi(n_disks - 1, a, c, b)
    print("move disk from %d to %d" % (a, b))
    hanoi(n_disks - 1, c, b, a)
```

Sin embargo, este algoritmo tiene un **alto coste** (aún para n_disks bajos, tarda mucho).

Divide y vencerás

1. Divide un problema P en m subproblemas P_m
2. Resolver los P_m obteniendo las soluciones S_m
3. Combinar las soluciones S_m en una solución S para P

Suelen resolverse con **recursión**. Ejemplos de uso de está técnica: algoritmos de ordenación.