Programación II

Nociones de complejidad

Universidad Nacional de Rosario.
Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Definición de problema

Un **problema** es una función o una asociación de entradas con salidas.
Un **ejemplar**, **instancia** o **entrada** de un problema es un caso particular o concreto del mismo.
Por ejemplo, encontrar el promedio de dos números a y b es un ejemplo de problema. Una instancia posible de este problema es encontrar el promedio entre 4 y 40.

Definición de algoritmo

¿Que es un algoritmo?
Informalmente, podemos decir que un algoritmo es un método sistemático, que puede ser realizado mecánicamente, para resolver un problema dado. Formalmente, es difícil de definir y no hay consenso al respecto. Pero si hay consenso sobre que propiedades debe cumplir un método para ser considerado un algoritmo.

Algoritmos vs. Programas

Nota: Un mismo problema puede ser resuelto por varios algoritmos.
Un **programa** es una instanciación de un algoritmo en un lenguaje de programación.
Nota Un algoritmo puede ser implementado por programas muy diferentes ! (lenguajes de programación, recursión vs. iteración, cuestiones de estilo, etc).

Nociones de complejidad - primer ejemplo

```
def fcn(n: int) -> int:
    r = 0
    for i in range(n):
        for k in range(n):
            r = r + i*k

    return r
```

- Supongamos que mi compu calcula `fcn(12000)` en 1 segundo.
- ¿Cuánto tiempo crees que tardaría en procesar `fcn(24000)` ?
- Supongamos que mi compu calcula `fcn(a)` en m segundos, ¿cuantos segundos tardaría en procesar `fcn(2a)`? ¿y en procesar `fcn(10a)`?

Lecciones importantes

- Medir el tiempo depende de en que computadora ejecutemos, pero la cantidad de operaciones siempre será la misma: por ello, conviene medir el "tiempo" en cantidad de operaciones.
- ¿Que operaciones contar? En general, tendremos que realizar alguna suposición acerca de cuál es la operación mas costosa. En el ejemplo anterior, desestimamos el tiempo incurrido en realizar sumar y asignaciones, dado que son operaciones mucho más sencillas que una multiplicación.
- Nos interesa saber como cambia la cantidad de operaciones dado un cambio en la entrada: nos centraremos en medir como cambia la cantidad de operaciones necesarias como una **función matemática** de su entrada.

Nociones de complejidad - segundo ejemplo

```
def cubo(x: float) -> float:
    return x * x * x
```

Supongamos que mi compu calcula `cubo(a)` en m segundos, ¿cuantos segundos tardaría en procesar `cubo(2a)`? ¿y en procesar `cubo(10a)`?

Lecciones importantes

- Nos vamos a concentrar en encontrar algoritmos cuyo tiempo de ejecución aumente lo menos posible a medida que la entrada se hace más y más grande.
- Es decir, no nos va interesar el número exacto de operaciones, ni siquiera la función exacta, si no **el orden de crecimiento de dicha función**.

Nociones de complejidad - Tamaño de entrada / Cota superior

Para un algoritmo que toma como entrada un elemento de un conjunto E, vamos a considerar un función T_1 con dominio en E y codominio en $\mathbb{N}$ tal que $T_1(e)$ sea la cantidad de operaciones que ejecuta el algoritmo con la entrada e.
Esta conceptualización tiene algunos problemas:

- No tenemos ni idea de la forma que tiene el conjunto E. Como tenemos que escribir una función, nos gustaría tener un dominio más lindo, como $\mathbb{N}$.
- T_1 no necesariamente es creciente, lo cuál es molesto para algunos análisis.

Nociones de complejidad - Tamaño de entrada / Cota superior

Para solucionar estos temas, cambiamos levemente nuestra definición utilizando el concepto de **cota superior** o **análisis pesimista**.

La idea es utilizar una función T_2 con dominio en $\mathbb{N}$ y codominio en $\mathbb{N}$ tal que $T_1(e) \leq T_2(n)$ donde n representa el "tamaño" de la entrada e y además sea creciente.

Así, nos aseguraremos de que la cantidad de operaciones que realice el algoritmo será siempre menor a $T_2(n)$.

Nociones de complejidad - ejemplo 3

```
def find_max(v: list[int]) -> int:
    # precondición: v es no vacía
    r = v[0]
    for value in v[1:]:
        if value > r:
            r = value
    return r
```

Si n es el tamaño de la lista de entrada tenemos

- a lo sumo n asignaciones a r. (recordar ser pesimistas !!).
- $n-1$ comparaciones entre `value` y r.
- $n-1$ asignaciones a `value`.

Por lo tanto... $T(n) = 3n - 2$!

Nociones de complejidad - ejemplo 4

```
def cosa(v: list[int]) -> int:
    r = 0
    for i in range(len(v)):
      for k in range(77):
        r += v[i] % (k+1)
    return r
```

¿cuantas veces se ejecutan asignaciones a r ???

Nociones de complejidad - ejemplo 5

```
def cosas_peores(v: list[int]) -> int:
    r = 0
    for i in range(len(v)):
        for k in range(77):
            r+= v[i] % (k+1)
        for h in range(len(v)):
            for k in range(88):
                r += v[i] % (k+1)

    return r
```

¿cuantas veces se ejecutan asignaciones a *r* ???

Nociones de complejidad - ejemplo 6

```
def funcion1(n: int) -> int:
 r = 0
 for i in range(n):
     for k in range(5):
         r += 1

 return r
```

¿cuantas veces se ejecutan asignaciones a *r* ???

Nociones de complejidad - ejemplo 7

```
def funcion2(n: int) -> int:
 r = 0
 for i in range(n):
     k = 0
     r += 1
     k += 1
     r+=1
     k += 1
     r+=1
     k += 1
     r+=1
     k += 1
     r+=1

 return r
```

¿cuantas veces se ejecutan asignaciones a r ???

La notación O grande

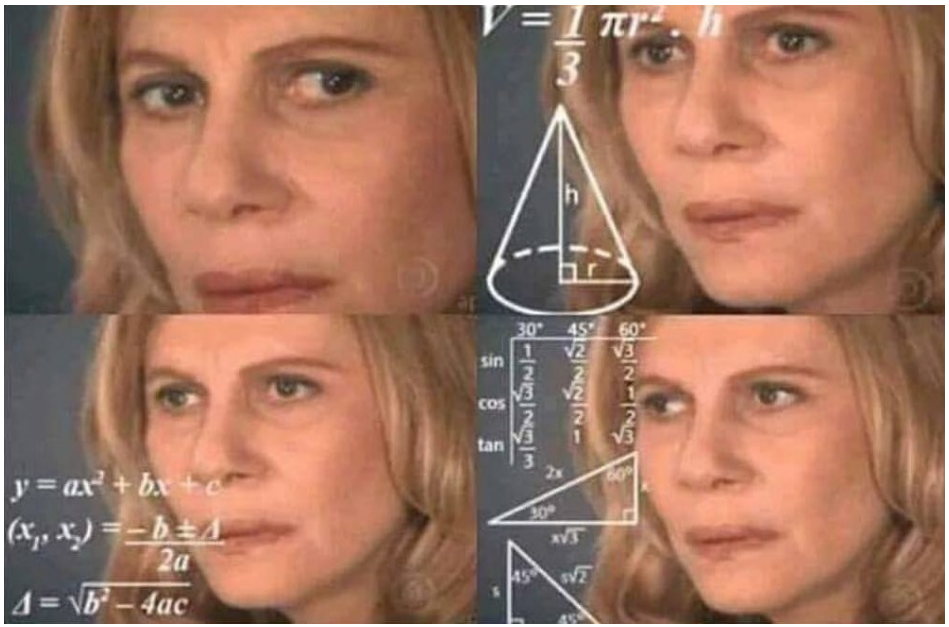
Si f es una función con dominio en $\mathbb{N}$ se dice que f es de orden $O(g(n))$ si y solo si existe una constante K tal que:

$$f(n) < Kg(n) \qquad \forall n \in \mathbb{N}$$

La notación O grande

Si f es una función con dominio en $\mathbb{N}$ se dice que f es de orden $O(g(n))$ si y solo si existe una constante K tal que:

$$f(n) < Kg(n) \qquad \forall n \in \mathbb{N}$$



La notación O grande

Simplifiquemos:

- Las operaciones básicas (asignar, sumar, etc.) conllevan **tiempo constante**: $O(1)$.
- Las constantes multiplicando se pueden descartar.
- Al sumar dos complejidades, nos quedamos con la mayor.
- Lo que hay que tener en cuenta es el parámetro de entrada que nos defina cuántas operaciones ejecutamos (por ejemplo, el límite en un for).

Nociones de complejidad - ejemplo 8

```
def suma_modular(n: int) -> int:
    r = 0
    for i in range(n%100):
        r += i

    return r
```

El ciclo for siempre se ejecuta menos de 100 veces - poniéndonos pesimistas podemos suponer que siempre se ejecuta 100 veces.
Dentro del ciclo for se ejecuta una cantidad de constante de operaciones.
Por lo tanto diremos que el algoritmo es de **complejidad constante** o bien $O(1)$.
Esto quiere decir que el algoritmo siempre tomará más o menos la misma cantidad de tiempo sin importar cuanto crezca n.
Nota: Obviamente siempre nos va interesar la función "mas chica" que sirva como cota.

Nociones de complejidad - ejemplo 9

```
def bubble_sort(lista: list[int]) -> list[int]:
    # Ordena una lista de enteros in-place
    n = len(lista)
    # Recorremos todos los elementos
    for i in range(n):
        for j in range(n-1):
            # Intercambiar si corresponde
            if lista[j] > lista[j + 1]:
                tmp = lista[j]
                lista[j] = lista[j+1]
                lista[j+1] = tmp
    return lista
```

Nociones de complejidad - ejemplo 9

El primer for itera n veces.
Por cada iteración del primer for, el segundo for itera $n-1$ veces.
Por lo tanto, siendo pesimistas, el condicional y su cuerpo se ejecutan a lo sumo $n(n-1)$ veces.
Este algoritmo tiene complejidad $O(n^2)$

Nociones de complejidad - ejemplo 10

```
def f(n: int) -> int:
  r = 0
  for i in range(n):
   r+=n
  return r

def g(n: int) -> int:
  r = 0
  for k in range(n):
    for i in range(n):
      r += f(i) + f(k)

  return r
```

¿cual es la complejidad de g???

Nociones de complejidad - ejemplo 10

Es difícil analizar la cantidad exacta de iteraciones que están ocurriendo.
Por un argumento similar al del ejemplo anterior, podemos ver que en g, asignamos a r en el orden de $O(n^2)$ veces.
A su vez, con cada llamada a f, estamos ejecutando $O(n)$ operaciones.
Se puede concluir que ejecutar f llevara $O(n^3)$ operaciones.

Nociones de complejidad - ejemplo 11

El algoritmo de **búsqueda binaria** es un algoritmo muy estudiado que resuelve de forma eficiente el problema de encontrar si un elemento pertenece o no a una lista **ordenada** de elementos.

```
def buscar(a: list[int], x: int) -> bool:
    n = len(a)
    return _buscar(a, x, desde=0, hasta=n)

def _buscar(a: list[int], x: int,
    desde: int, hasta: int) -> bool:
    while desde < hasta:
        mitad = (desde+hasta)//2
        if a[mitad]==x:
            return True
        elif a[mitad] < x:
            desde = mitad +1
        else:
            hasta = mitad
    if a[desde] == x:
        return True
    return False
```

Medir antes de programar

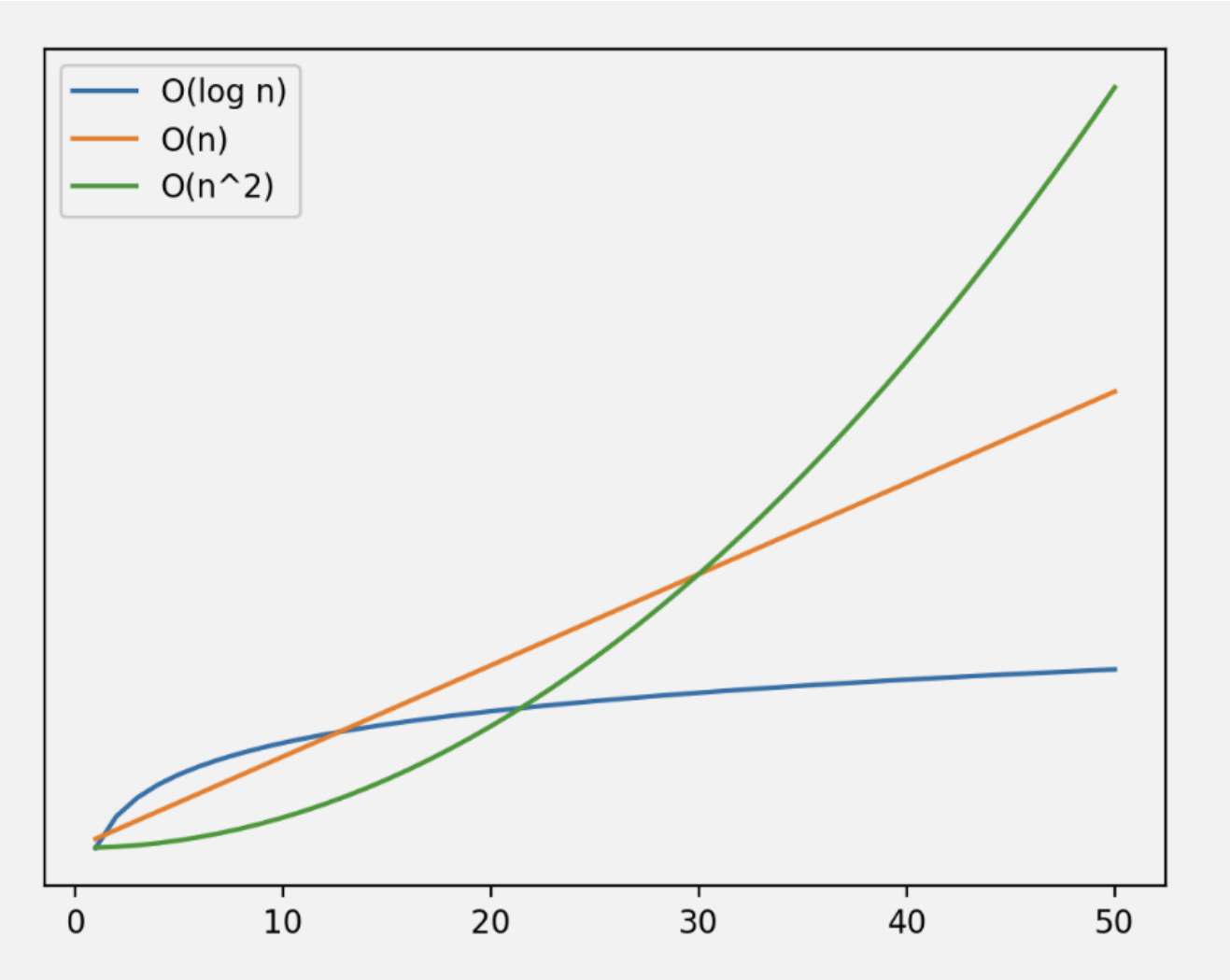
Supongamos que tenemos el problema de encontrar el máximo elemento de una lista de enteros no vacía. Para ello, se proponen dos algoritmos:
Algoritmo 1 Para cada elemento de la lista *i*, chequear si *i* es mayor a cualquier otro elemento de la lista. Si es así, *i* sera el mayor elemento y podemos retornarlo.
Algoritmo 2 Inicializar el valor máximo como `float('-inf')`. Para cada elemento i de la lista, si el valor de máximo es menor que el valor de i entonces se sustituye el valor de máximo por el valor de i. Al finalizar, retornar máximo.
¿que algoritmo nos conviene implementar?

¿Y entonces?

Una computadora con un único procesador de 1,5Ghz puede realizar aproximadamente 200 millones de sumas en un segundo. Esta claro que la cantidad de sumas realizadas por una iteración puede cambiar, pero, para tener una idea, en la siguiente tabla se muestran varias complejidades y valores máximos de n que se podrían procesar con un algoritmo de dicha complejidad si se requiere que el programa funcione en menos de un segundo:

Complejidad	**Valor Máximo de n**
$O(1)$	∞
$O(logn)$	$2^{50000000}$
$O(\sqrt{n})$	10^{15}
$O(n)$	50000000
$O(nlogn)$	5000000
$O(n^2)$	5000
$O(n^3)$	500
$O(n^4)$	80
$O(2^n)$	20
$O(n!)$	11

¿Y entonces?



Conclusiones

- Uno de nuestros objetivos es que puedan **estimar** rápidamente el orden de una función o algoritmo.
- No esperamos que la tengan súper clara ahora, pero si que empiecen a cuestionarse la eficiencia del código que escriben.

¿Donde leer?

- T. Cormen et al, 2009
  Introduction to Algorithms, third edition.
- Luis E. Vargas Azcona
  Problemas y Algoritmos.
- Bel, Walter
  Algoritmos y estructuras de datos en Python. Un enfoque ágil y estructurado.