Prólogo al curso

31 de agosto de 2020 - Bernardo Subercaseaux

A la hora de diseñar y analizar algoritmos nos interesa describir su *calidad* de la forma más precisa posible. La calidad de un algoritmo se puede evaluar a través de diferentes parámetros cuantitativos: el uso de memoria RAM, el número de operaciones realizadas, el número de veces que se llama a una cierta API, etc. A lo largo del curso estudiaremos la calidad de distintos algoritmos y estructuras de datos desde dos puntos de vista: experimental y teórico. Por ejemplo, si $\mathcal{A}$ es un algoritmo para ordenar datos, verán resultados como

- (**Experimental**) el algoritmo $\mathcal{A}$ utilizó $5{,}6 \times 10^{10}$ operaciones, y tardó 172.9 segundos, en ordenar un archivo de 4.2 GBs.
- (**Teórico**) el algoritmo $\mathcal{A}$ requiere n^2 operaciones para ordenar un archivo que contiene n datos.

Mientras que los resultados experimentales involucran magnitudes concretas (172.9 segundos), los resultados teóricos involucran *funciones* (n datos $\to n^2$ operaciones). Las funciones que mencionan los resultados téoricos suelen describir el *costo* (operaciones, bloques de memoria, llamados a una API, etc.) en función del tamaño de la entrada del algoritmo. Más aún, a menos que digamos explícitamente lo contrario, los análisis teóricos se centrarán en el *peor caso* posible para el algoritmo estudiado. ¿Qué quiere decir el *peor caso*? Consideremos un algoritmo $\mathcal{A} : \{0,1\}^* \to \{0,1\}^*$ que recibe strings binarios y entrega strings binarios como resultado. Por cada string binario w podemos definir $C_{\mathcal{A}}(w)$ como el *costo*[1] de correr el algoritmo $\mathcal{A}$ con entrada w. Lo que nos interesará en general será estudiar el costo de $\mathcal{A}$ sobre la entrada w que más costo tiene, es decir, estudiaremos la función de rendimiento $R_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ definida como

$$R_{\mathcal{A}}(n) = \max_{w \in \{0,1\}^n} C_{\mathcal{A}}(w)$$

Esta se conoce también como el *análisis de peor caso* de $\mathcal{A}$, o la *complejidad (en el peor caso)* de $\mathcal{A}$. **¡Ejercicio, saca un lápiz!**

Problema 1. ($\star$) ¿Cómo se definiría la función $\bar{R}_{\mathcal{A}}$ que describe el rendimiento *promedio* del algoritmo $\mathcal{A}$?

Aún quedan por precisar varios detalles. Intentemos desarrollarlos a través de un ejemplo. Consideremos que $\mathcal{A}$ es un algoritmo que recibe un arreglo L de n enteros, y retorna en qué posición aparece por primera vez el número 7, o -1 si 7 no aparece en L. El algoritmo $\mathcal{A}$ está implementado en Python de la siguiente manera:

```
def find_seven(L):
  for i in range(len(L)):
    if L[i] == 7:
      return i
  return -1
```

Además, consideremos que nos interesa estudiar el costo como *número de operaciones*, este será el caso principal durante el curso. La pregunta natural es, ¿qué es exactamente una operación? Por ejemplo, la línea `if L[i] == 7:`, ¿cuenta como 1 operación? ¿2 operaciones? ¿3 operaciones? Hay varias formas de resolver esta ambigüedad.

[1] Número de operaciones, número de accesos a memoria, número de llamados a una API, etc.

1. (**Modelo de Computación**) una forma de resolver esta ambigüedad es fijar un *lenguaje de programación* o tipo de *máquina*, y definir explícitamente cómo se cuentan las operaciones en el modelo dado. Por ejemplo, en el modelo de máquinas de Turing, en general se cuenta una operación como un paso en la función de transición de la máquina. Esta definición con máquinas de Turing es sencilla, el problema es que programar en máquinas de Turing es muy engorroso. Una opción intermedia es usar un lenguaje de bajo nivel, suficientemente similar a una máquina de Turing para que sea sencillo contar operaciones, y suficientemente legible para que podamos programar en él.

2. (**Operaciones especiales**) otra solución, que usaremos varias veces en el curso, es solo considerar un cierto tipo de operaciones especiales, y despreciar todas las otras. Por ejemplo, podríamos contar solo la cantidad de veces que se ejecuta una comparación con ==.

3. (**Don't worry be happy**) una tercera solución, que se toma muy frecuentemente y tomaremos durante el curso, es simplemente olvidarse el problema y no especificar. ¿Son 1, 2 o 23 operaciones? No importa. Lo importante es que es un número entero *constante*. Es fundamental notar que n (el tamaño de la entrada) no es una constante. Por lo tanto nos dará lo mismo si un algoritmo requiere 24 o 9271 operaciones, o si requiere $n + 7$ o $n + 9$ operaciones. Por el contrario, **NO** nos dará lo mismo si requiere n o n^2 operaciones, ya que en ese caso la diferencia no es *constante*. Formalizaremos esta idea más adelante.

Sigamos con nuestro ejemplo tomando la estrategia número 2, en que solo contamos la cantidad de veces que se evalúa la comparación con == en la tercera línea del código. Así tenemos que $C_{\mathcal{A}}([7,7,7]) = 1$, ya que el algoritmo retornará tras la primera comparación. Por otra parte, $C_{\mathcal{A}}([5,6,7]) = 3$, ya que se comparan los 3 elementos del arreglo de entrada. Además, es claro que para cualquier arreglo L de tamaño 3 ser cumple que $C_{\mathcal{A}}(L) \leq 3$, ya que no hay más de 3 elementos que comparar. De esto deducimos que $R_{\mathcal{A}}(3) = 3$, ya que el peor caso es que una lista de tamaño 3 requiera las 3 comparaciones. Notemos además que el número 3 es simplemente un ejemplo, y el mismo análisis funciona para cualquier natural n, de donde $R_{\mathcal{A}}(n) = n$.

Hasta ahora todo parece andar bien, pero nuevamente hay un detalle que saltamos. Habíamos definido la función $C_{\mathcal{A}}$ sobre strings binarios, y sin embargo la acabamos de usar con arreglos (e.g. $C_{\mathcal{A}}([7,7,7]) = 1$). Una forma de resolver este abuso de notación es usar el hecho de que todos los objetos que uno trata en un computador se pueden representar de alguna manera en binario, y ver entonces cómo representar arreglos (el tipo de entrada en nuestro caso) en binario[2]. Esto suele ser engorroso también, así que en general para simplificarnos la vida nos olvidaremos de la representación binaria y mediremos el tamaño de la entrada de distintas formas dependiendo del problema que estemos estudiando. Por ejemplo, en problemas sobre arreglos es natural definir el tamaño de la entrada como el número de elementos (diciendo por ejemplo $n := |L|$), mientras que en problemas de grafos uno puede definir el tamaño como la suma de la cantidad de nodos y aristas ($n := |V| + |E|$).

Otro ejemplo; usaremos la estrategia número 3 (*Don't worry be happy*), estudiando un algoritmo para encontrar la mediana de un arreglo ordenado. En un arreglo ordenado L de tamaño $n = 2k+1$ (i.e. impar), la mediana es $L[k]$, indexando desde 0, mientras que si el tamaño es de la forma $n = 2k$ (i.e. par) entonces la mediana es el promedio de los elementos centrales, es decir, $(L[k-1] + L[k])/2$. El siguiente código en Python implementa esto:

[2]Por ejemplo, una forma ineficiente de representar listas con números $a_1, \ldots, a_n$ sería escribirlas como $01^{a_1}0 \ldots 01^{a_n}0$. De esta forma la lista $[7,7,7]$ se representaría como $01111111101111111011111110$.

```python
def median_sorted(L):
  n = len(L)
  k = n // 2 # integer division
  if n % 2 == 1:
    return L[k]
  else:
      return (L[k-1] + L[k])/2
```

¿Cuál es la complejidad de este código/algoritmo? (en general así nos preguntaremos por la función $R_{\mathcal{A}}$; sin mencionarla explícitamente) La respuesta es ¡**depende**! Por ejemplo, de cuánto cueste hacer la división por 2, o tomar módulo 2 para evaluar la paridad. Asumamos que estas tienen un costo constante[3], es decir, que no depende de n. La estrategia *Don't worry be happy* nos dice entonces que cada parte del algoritmo requiere un número constante de operaciones, y dado que el algoritmo consta de un número constante de partes, entonces siempre ejecutará un número constante de operaciones. No podemos decir cuántas, pero si podemos decir que es constante. *Good Enough*. Podríamos intentar definir precisamente qué vamos a contar como operación, incluyendo la resta (-), la suma (+), el módulo (%), la asignación (=), los accesos a un arreglo ([]), la comparación (==), etc. Obteniendo algo cómo (no se preocupen de los números concretos!):

$$R_{\mathcal{A}}(n) = \begin{cases} 6 & \text{si } n \text{ es impar} \\ 11 & \text{si no.} \end{cases}$$

En general no vamos a querer hacer esto, porque no es relevante si es 6 o 11, solo el hecho de que es constante, y por tanto funcionará rápidamente independiente del tamaño de la lista. Además, ponerse a contar operaciones de manera tan *fina* es engorroso y aburrido. Formalmente diremos que $R_{\mathcal{A}} \in O(1)$, lo que quiere decir que existe una constante k tal que $R_{\mathcal{A}}(n) \leq k$ para todo n. Generalicemos esta idea! Queremos las funciones desde un punto de vista que desprecia la diferencia entre constantes aditivas o multiplicativas. Esto es justamente lo que la *notación asintótica* consigue. En particular comencemos por la más importante, la notación O:

Definición 1 (Big O) *Dadas dos funciones no negativas*[4] $f(n)$ *y* $g(n)$*, diremos que* $f(n) \in O(g(n))$ [5] *si existe una constante* $C \in \mathbb{R}^+$ *y un natural* n_0 *a partir del cuál* $f(n) \leq Cg(n)$*.*

Por ejemplo, $n/2 + 3 \in O(n)$, ya que al tomar $C = 1$ y $n_0 = 6$ se cumple que $n/2 + 3 \leq n$ para todo $n \geq n_0 = 6$. Por otra parte, podemos ver por contradicción que $n^2 \notin O(n)$, en efecto, si $n^2 \in O(n)$ tendríamos que a partir de un cierto n_0 y para una constante C se cumple que $n^2 \leq Cn$, y por lo tanto $n \leq C$. Pero esto último no es cierto ya que siempre habrá un natural n mayor que la constante C.

La parte clave de la definición es el símbolo $\leq$, en efecto, podemos entender $f(n) \in O(g(n))$ como *f crece más lento que g, o al mismo ritmo*.

Problema 2. ($\star$) Pruebe usando la definición que

1. $n \log n \in O(n^2)$
2. $2n + 7 \in O(n)$

[3]Esto es usualmente cierto, pero no es cierto que $n \,\%k$ se pueda calcular en tiempo constate para cualquier valor de k. Si tienen dudas concretas de si una operación funciona en tiempo constante pueden preguntarlo.

[4]En general asumiremos que se trata de funciones de $\mathbb{N}$ a $\mathbb{R}^+$.

[5]que leemos *f es oh de g*, o también *f es orden de g*

3. $1/n \in O(1)$
4. $4^n \notin O(2^n)$
5. $\sqrt{n} \notin O(\log^2 n)$

Note que el punto 2 del problema anterior dice que $2n + 7 \in O(n)$, lo que quizás entra en conflicto con la intuición de que $2n + 7$ crece más lento que n. En efecto, lo que ocurre es que ambas crecen al mismo *ritmo*, ya que la definición de la notación O olvida las constantes multiplicativas y aditivas en un sentido preciso, lo que probaremos en el siguiente ejercicio.

Problema 3. (⋆) Demuestre que para cualquier función f, tal que a partir de un cierto n_0' cumple $f(n) \geq k$ para una constante positiva k, entonces se cumple que para cualquier constante positiva c, $cf(n) + k \in O(f(n))$

Solución 3. Sabemos que a partir de n_0' se cumple que $cf(n) + k \leq (c + 1)f(n)$, por lo que basta tomar $C = c + 1$ y $n_0 = n_0'$.

Note que esto prueba directamente el Problema 2.2, ya que a partir de $n_0' = 7$ se cumple que $f(n) = n \geq 7$. Note también que no es correcto decir que la notación O desprecia todas las constantes, ya que el Problema 2.4 muestra que no siempre es así.

Problema 3. (⋆) Demuestre la regla del producto: si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n)f_2(n) \in O(g_1(n)g_2(n))$.

Problema 4. (⋆) Demuestre la regla de la suma: si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n) + f_2(n) \in O\left(\max(g_1(n), g_2(n)\right)$

Problema 5. (⋆) Demuestre que la notación O es transitiva, es decir, si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$ entonces $f(n) \in O(h(n))$.

Problema 6. (⋆) Muestre un algoritmo que use a lo más[6] $O(n^3)$ operaciones para multiplicar dos matrices de $n \times n$.

Solución 6. Utilizaremos esta solución para ejemplificar el nivel de detalle que uno usa normalmente en problemas del curso (en realidad durante el curso usaremos aún menos detalle, pero dado que esto es una introducción...). En primer lugar, no nos molestaremos pensando en la representación binaria de las matrices de entrada, y consideraremos que tenemos las matrices de entrada como arreglos 2D (arreglos de arreglos). Luego inicializaremos una matriz para almacenar el resultado, de tamaño $n \times n$. Esto toma $O(n^2)$ operaciones, ya que se deben inicializar n^2 casillas, e inicializar una casilla toma tiempo constante. Esto último es muy importante; no estamos diciendo que inicializar una casilla sea lento o rápido, solo que es constante, es decir, que no depende de n. Luego, recordando la definición de la multiplicación de matrices, computaremos cada una de las n^2 casillas del resultado como un producto punto entre una fila de la primera matriz y una columna de la segunda matriz. Es decir, el producto punto de dos vectores de n elementos. Note que esto se puede hacer con $O(n)$ operaciones; n productos y $n - 1$ sumas = $2n + 1 \in O(n)$. Dado que se computamos n^2 casillas, haciendo $O(n)$ operaciones por cada una, el costo total es $O(n^3)$.

[6] esta palabra indica análisis de peor caso!

Problema 6. (⋆) Muestre que $\sum_{i=1}^{n} i \in O(n^2)$.

Problema 7. (⋆⋆) Sea $p(n)$ un polinomio de grado $k \geq 0$ con coeficientes en $\mathbb{Z}$ y donde el coeficiente del término de grado k es positivo. Demuestre que $p(n) \in O(n^k)$.

Problema 8. (⋆⋆) ¿Cuáles de las siguientes afirmaciones son ciertas?

1. Si $f(n) \in O(n)$ entonces $f^2(n) \in O(n^2)$
2. Para todo par de funciones f y g se cumple que $f(n) \in O(g(n))$ o bien $g(n) \in O(f(n))$, o potencialmente ambas.
3. Si $f(n) \in O(n)$ entonces $2^{f(n)} \in O(2^n)$
4. Si $f(n) \in O(g(n))$ entonces $f(2n) \in O(g(n))$
5. Para toda función f se cumple que $f(n) \in O(f(n-1))$.

Como mencionamos anteriormente, la notación O es una especie de $\leq$ entre funciones. Ahora definiremos los equivalentes a un $\geq$ a y un $=$.

Definición 2 (Big Omega) *Dadas dos funciones no negativas $f(n)$ y $g(n)$, diremos que $f(n) \in \Omega(g(n))$ si existe una constante $C \in \mathbb{R}^+$ y un natural n_0 a partir del cuál $f(n) \geq Cg(n)$.*

Problema 9. (⋆) Pruebe que $2^n \in \Omega(n^2)$ y $n \notin \Omega(\log n)$.

Problema 10. (⋆) Pruebe que $f(n) \in O(g(n)$ si y solo si $g(n) \in \Omega(f(n))$.

Definición 3 (Big Theta) *Dadas dos funciones no negativas $f(n)$ y $g(n)$, diremos que $f(n) \in \Theta(g(n))$ si $f(n) \in O(g(n))$ y también $f(n) \in \Omega(g(n))$.*

Problema 11. (⋆) Sean $n \geq k$ enteros, muestre que $\binom{n}{k} \in \Theta(n^k)$.

Problema 12. (⋆) El problema k-CLIQUE tiene como entrada un grafo de n vértices y un entero k, y se debe responder si existe un grupo de k vértices en el grafo donde cada par está conectado por una arista (a tal grupo se le llama un clique). Muestre un algoritmo que realiza $\Theta(k^2 n^k)$ operaciones para determinar si un grafo de n vertices tiene un k-clique.

Problema 13. (⋆⋆) Suponga que $\lim_{x\to\infty} \frac{f(x)}{g(x)} = l$. Demuestre lo siguiente:

- Si $l = 0$ entonces $f \in O(g)$ y $g \notin O(f)$.
- Si $l = \infty$ entonces $g \in O(f)$ y $f \in O(g)$.
- Si $l \in \mathbb{R}^+$ entonces $f \in \Theta(g)$.

Problema 14. (⋆) Demuestre que $n! \in \Omega(2^n)$ y que $n^n \notin O(n!)$. Utilice la aproximación de Stirling, que encontrará en internet[7].

Estos ejercicios deberían ser suficiente para soltar la mano con la notación. Antes de cerrar hay dos puntos que quisiera mencionar:

- **¡Esta notación tiene mucho sentido!** Dado que el rendimiento de un algoritmo es una función, nos interesa entonces tener formas de comparar funciones; de decir que una función crece más lento que otra, lo que significaría que su algoritmo asociado es más eficiente. Pero una función está compuesta por una infinidad de puntos! Una opción sería decir que una función f es menor que g si es menor en todos sus puntos. Sin embargo, esto es demasiado restrictivo. Por ejemplo, si $f(1) = 2$ y $g(1) = 1$, y después de eso $f(n) = n$ y $g(n) = n^7$ no podremos decir que f es menor que g, ya que falla el punto 1. Para evitar eso tiene sentido hablar de lo que ocurre a partir de un cierto n_0; preocuparse de lo que pasa desde ahí en adelante permite obviar irregularidades iniciales. De ahí también el nombre de notación *asintótica* [8]. Por otra parte, la constante multiplicativa C aparece como consecuencia de la forma de contar operaciones que discutimos: queremos obviar si algo requiere 2, 3, o 42 operaciones, mientras sea un número constante. Así, si una función hace el doble o el triple de operaciones que otra, sigue siendo *menor o igual* bajo el orden que establece la notación O.

- Comenzamos hablando de que durante el curso analizarán algoritmos en la práctica y desde la teoría. Es importante y beneficioso hacer el puente entre ambos mundos. Para eso se suelen hacer aproximaciones a la realidad. Por ejemplo, un computador de escritorio puede hacer aproximadamente 10^9 operaciones básicas[9] en 1 segundo. Esto quiere decir que si tenemos un algoritmo que hace $\Theta(n^5)$ operaciones, y en un caso $n = 1000$, entonces esperaríamos *al rededor*[10] de 10^{15} operaciones, lo que tomaría $10^{15}/10^9 = 10^6$ segundos, o aproximadamente ¡**11 días**! Esto es buena motivación a diseñar mejores algoritmos, ya que uno que requiera $\Theta(n^4)$ operaciones tardaría tan solo $10^{12}/10^9 = 1000$ segundos, o aproximadamente ¡**17 minutos**! A la hora de experimentar podremos contrastar nuestros resultados con estas aproximaciones.

[7] Esta aproximación se presenta de diferentes formas, y buscar en internet alguna que le sirva para demostrar lo que quiere es un buen ejercicio.

[8] El Problema 13 relaciona directamente esta notación con los límites cuando la variable tiende a infinito

[9] una definición precisa de lo que son operaciones básicas dependerá del procesador en cuestión

[10] notación asintótica! pueden haber constantes de diferencia