

Demostraciones matemáticas

Federico Lebrón



A la Universidad de Buenos Aires.

Índice

Parte I:	Demostraciones	5
1	¿Qué es una demostración matemática?	6
1.1	Demostraciones en la historia	6
1.2	Contexto en el que están demostrando	9
1.3	Formalidad y rigor	11
2	¿Cuándo, qué, y para qué demostramos?	13
3	¿Cómo aprendemos a demostrar?	16
4	¿Cómo demostramos?	17
4.1	Formalizar la consigna	18
4.2	Comprender qué se nos pide	19
4.3	Considerar ejemplos	20
4.4	Estrategias de demostración	24
4.4.1	Inducción	24
4.4.2	Correctitud de ciclos en algoritmos	31
4.4.3	Correctitud de algoritmos recursivos	33
4.4.4	Definiciones equivalentes	34
4.4.5	Contrarecíproco y contradicción	36
4.4.6	Si y sólo si	40
4.4.7	Partir en casos	42
4.4.8	Unicidad	43
4.5	Pasar en limpio	43
5	Errores comunes	49
5.1	Ser informal	49
5.2	No decir nada	50
5.3	Empezar con la conclusión	51
5.4	No entender qué estamos asumiendo	52
Parte II:	Fundamentos matemáticos	53
1	Lógica	54
1.1	Ejercicios	57
2	Conjuntos	58
2.1	Ejercicios	63
3	Funciones	64
3.1	Ejercicios	67
4	Sucesiones y series	68
4.1	Ejercicios	72
5	Combinatoria	73
6	Análisis asintótico	78
6.1	Ejercicios	86
7	Álgebra asintótica	87
7.1	Errores frecuentes	89
7.2	Ejercicios	92
8	Recurrencias	93
8.1	Expansión	93
8.2	Árboles de recursión	96
8.3	Teorema maestro	99
8.4	Ejercicios	105
Parte III:	Diseño y correctitud de algoritmos	106

1	Correctitud de algoritmos	107
2	Análisis de complejidad de algoritmos	107
2.1	Modelo de cómputo	107
2.2	Tamaño de entrada	108
2.3	Funciones de costo	109
2.4	Peor caso, mejor caso, y caso promedio	114
2.5	Del algoritmo a la función asintótica	115
2.6	Ejercicios	127
3	Divide and conquer y programación dinámica	127
3.1	Ejercicios	145
4	Backtracking	149
5	Greedy	153
Parte IV:	Teoría de grafos	176
1	Definiciones básicas	177
2	Árboles	177
3	Recorridos	181
4	Caminos mínimos	181
5	Árboles generadores mínimos	197
6	Flujo	198
7	Matching	200
8	Circuitos y caminos	200
9	Coloreo	204
10	Planaridad	209
11	Homomorfismo e isomorfismo de grafos	211
Parte V:	Complejidad computacional	217
Parte VI:	Programación lineal y entera	219
Parte VII:	Apéndice	221
1	Demostración del Teorema Maestro	221
2	Demostración del número de comparaciones de Mergesort	229
	Bibliografía	231

Parte I: Demostraciones

1 ¿Qué es una demostración matemática?

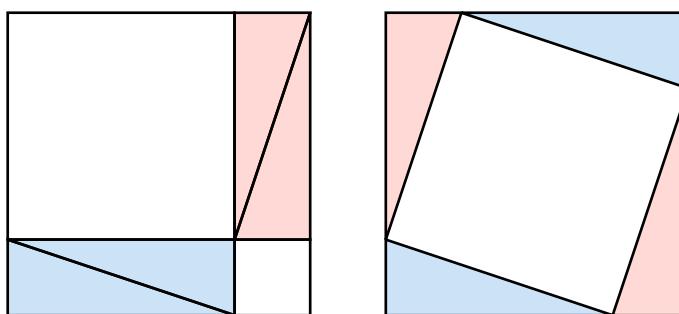
Una demostración matemática es un argumento convincente de la veracidad de una proposición matemática. Inmediatamente tenemos la pregunta, ¿convence a *quién*? Distintos argumentos van a convencer a distinta gente. Por ejemplo, el siguiente es un argumento de la veracidad del teorema de Pitágoras.

Teorema 1 (Teorema de Pitágoras)

Dado un triángulo rectángulo, la suma de los cuadrados de los catetos es igual al cuadrado de la hipotenusa.



Demostración.



¿Esto es siquiera una demostración? ¿Los convence? ¿Convencería a sus amigos o a alguien que parase en la calle? ¿Cómo saben, o definen, si «está bien»?

1.1 Demostraciones en la historia

Históricamente, empezamos haciendo matemática sin generalidad, y sin demostraciones. Un matemático en el 2000 a.c.¹ hubiera afirmado una proposición, y listo. Con suerte hubiera dado ejemplos que mostraran la veracidad de esa proposición en casos particulares. No había un lenguaje formal para hablar en generalidades sobre, por ejemplo, «los enteros». Había tablas de recíprocos de enteros y soluciones a ecuaciones cuadráticas, pero no un algoritmo de división o una expresión como $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Los griegos son los primeros que intentan dar argumentos de algún tipo para proposiciones generales. Thales de Mileto demostró el siguiente teorema, aunque la primer demostración que sobrevive es la de Euclides.

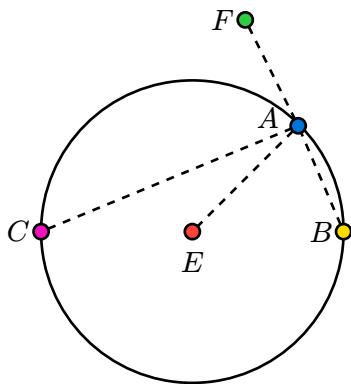
Teorema 2 (Teorema de Thales)

Si tenemos tres puntos A , B , y C en un círculo, donde la línea \overline{AB} pasa por el centro del círculo, entonces $\angle ABC$ es un triángulo rectángulo.



Demostración.

¹No había un concepto de «matemático» en ese momento. Los babilonios entendían conceptos sobre aritmética y álgebra desde el punto de vista de la geometría. Describían relaciones entre ángulos, lados, y perímetros de figuras dibujadas, y usaban esto para construir, arar, estimar distancias, etc.



Dada por Euclides en [1].

Sea ABC un círculo, \overline{BC} su diámetro, E su centro. Unir \overline{BA} , y \overline{AC} .

Digo que el ángulo $\angle BAC$ es rectángulo.

Unir \overline{AE} , y extender \overline{BA} hasta F . Como \overline{BE} es igual a \overline{EA} , el ángulo $\angle ABE$ también es igual al ángulo $\angle BAE$. Como \overline{CE} es igual a \overline{EA} , el ángulo $\angle ACE$ es igual al ángulo $\angle CAE$.

Luego, el ángulo $\angle BAC$ es igual a la suma de los dos ángulos $\angle ABC$ y $\angle ACB$. Pero el ángulo $\angle FAC$ exterior al triángulo

ABC también es igual a la suma de los dos ángulos $\angle ABC$ y $\angle ACB$. Luego el ángulo $\angle BAC$ también es igual al ángulo $\angle FAC$. Luego ambos son rectángulos. Luego el ángulo $\angle BAC$ es rectángulo.

□

¿Los convence esa demostración? ¿Dónde precisamente está F ? ¿Por qué $\angle FAC$ es igual a la suma de $\angle ABC$ y $\angle ACB$? En su momento, esta demostración no sólo era convincente, sino que fue parte del libro de demostraciones de geometría más famoso y celebrado de la historia. ¿Por qué puede ser que hoy en día nos resulta confusa?

Viajamos luego al 1758, donde en «De numeris qui sunt aggregata duorum quadratorum» («Sobre números que son la suma de dos cuadrados») Leonhard Euler prueba el siguiente teorema.

Teorema 3

Si p y q son dos números, cada uno de los cuales la suma de dos cuadrados, entonces el producto pq es también una suma de cuadrados.

♥

Demostración. Sea $p = aa + bb$ y $q = cc + dd$. Tendremos que $pq = (aa + bb)(cc + dd) = aacc + aadd + bbcc + bbdd$, que se puede representar de manera tal que $pq = aacc + 2abcd + bbdd + aadd - 2abcd + bbcc$ y por tal razón $pq = (ac + bd)^2 + (ad - bc)^2$, de donde el producto pq será la suma de dos cuadrados.

□

¿Qué les parece esa demostración? ¿Es más fácil de entender que las dos anteriores? ¿Los convence? ¿Les parece que convencería a *cualquiera*? ¿Qué asume esta demostración? ¿Qué significa «número» acá²? Si mis números son las horas del reloj, y estoy haciendo aritmética modular módulo 12, ¿sigue valiendo el teorema? ¿Y si mis «números» son funciones de \mathbb{R} a \mathbb{C} ? ¿Y si son elementos de punto flotante³ en una computadora, sigue valiendo, o se rompe algo que está asumiendo?

Ahora veamos una demostración formal, escrita en el lenguaje de programación Lean 4.

²El concepto formal de número real que usamos hoy en dia se definió en 1872, en paralelo por Richard Dedekind y Georg Cantor. Euler y sus matemáticos contemporaneos usaban nociones no totalmente formales, llegando a veces a paradojas.

³Siguiendo el standard de IEEE 754, por ejemplo punto flotante de 32 bits.

Teorema 4

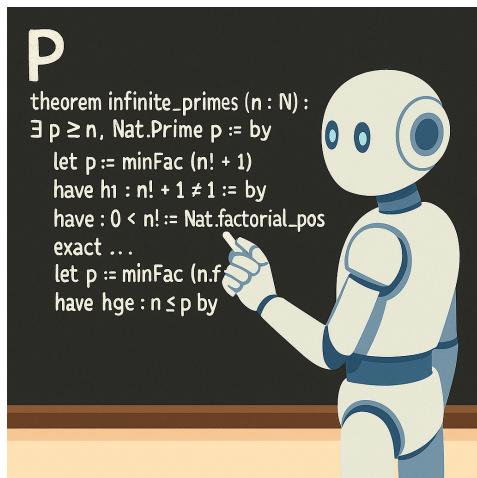
Para todo $n \in \mathbb{N}$, existe un número primo más grande que n .

♥

Demostración.

```
-- Primo (p : N) : Prop
theorem hay_infinitos_primos (n : N) : ∃ p, n ≤ p ∧ Primo p :=
let fac := n ! + 1
-- minFac (n : N) : N, el mínimo factor de n
let p := minFac fac
-- factorial_pos (n : N) : 0 < n !
-- succ_lt_succ {n m : N} : n < m → n + 1 < m + 1
-- ne_of_gt {a b : N} (h : b < a) : a ≠ b
have f1 : fac ≠ 1 := ne_of_gt <| succ_lt_succ <| factorial_pos n
-- minFac_es_primo {n : N} (n1 : n ≠ 1) : Primo (minFac n)
have pp : Primo p := minFac_es_primo f1
-- le_of_not_ge: {α: Tipo} {a b : α} : ¬(a ≤ b) → b ≤ a
have np : n ≤ p :=
le_of_not_ge fun h =>
-- dvd_factorial {m n : N} : 0 < m → m ≤ n → m | n !
-- minFac_pos (n : N) : 0 < minFac n
have h1 : p | n ! := dvd_factorial (minFac_pos fac) h
-- dvd_suma {k m n : N} (h : k | m) : k | n ↔ k | m + n
-- minFac_dvd (n : N) : (minFac n) | n
have h2 : p | 1 := (dvd_suma h1).2 (minFac_dvd fac)
-- no_divide_uno {p : N} (pp : Primo p) : ¬(p | 1)
no_divide_uno pp h2
(p, np, pp)
```

□



La correctitud de esta demostración es verificable automáticamente por una computadora. Esta demostración, ¿los convence? ¿Siquiera la pueden leer? Indudablemente es correcta, formal, y rigurosa (porque las computadoras no nos «creen» nada), ¿eso significa que «está bien»? ¿Ustedes escribirían algo así para comunicarle algo a un par? Si estas demostraciones se pueden verificar formalmente, ¿por qué no escribimos todas las demostraciones así? Esto nos dice que la formalidad no implica legibilidad para nuestra audiencia.

Por último, veamos un teorema moderno, traducido del libro Algebra de Serge Lang. Este es un libro de estudio clásico para álgebra abstracta, y el teorema se ve en la carrera de Matemática en la facultad.

Teorema 5 (Teorema de Jordan-Hölder)

Sea G un grupo, y sea

$$G = G_1 \supset G_2 \supset \dots \supset G_r = \{e\}$$

sea una torre normal tal que cada grupo G_i/G_{i+1} es simple, y $G_i \neq G_{i+1}$ para $i = 1, \dots, r-1$. Entonces cualquier otra torre normal de G teniendo las mismas propiedades es equivalente a esta.

♥

Demostración. Dado cualquier refinamiento $\{G_{i,j}\}$ para nuestra torre, observamos que para cada i existe precisamente un único índice j tal que $G_i/G_{i+1} = G_{i,j}/G_{i,j+1}$. Luego esta secuencia de factores no-triviales para la torre original, o la torre refinada, es la misma. Esto prueba nuestro teorema. □

¿Qué tal esta última? ¿Los convence? ¿Les parece clara la demostración? Imagino que a la mayoría les va a resultar incomprendible. Si le está faltando algo para convencerlos, ¿significa que esta demostración «está mal»?

Con estos ejemplos concluimos que lo que se considera una demostración matemática ha evolucionado, como ha evolucionado la noción de matemática que usamos. También, lo que se considera una demostración correcta va a depender del contexto en que uno se encuentre, qué puede uno asumir del lector, y para qué propósito uno está demostrando.

1.2 Contexto en el que están demostrando

Ustedes se encuentran cursando una carrera universitaria en las ciencias formales. El objetivo de la carrera es formarlos como computadores científicos. Como tales, deberían salir de la carrera ambos con conocimientos sobre objetos y herramientas específicas como grafos, números racionales, programación dinámica, sistemas operativos, inducción, cálculo diferencial, teoría de lenguajes, y algoritmos numéricos; pero más importantemente **deberían adquirir la habilidad de razonar formalmente**, de demostrar a cualquiera que se los pida que sus conclusiones son correctas, de entender y criticar razonamientos de otros, de saber cuándo y cómo usar las herramientas que conocen, y cuándo y cómo desarrollar ideas y algoritmos nuevos.

Para la primer parte, el temario de la carrera incluye esos temas en distintas materias. Para la segunda es que se los entrena en demostrar formalmente la veracidad de proposiciones, y la correctitud⁴ de argumentos. Se usan los objetos de estudio como grafos o números enteros como sujetos de las proposiciones y algoritmos que desarrollean.

Es por esto que la pregunta clásica de «¿pero cuándo voy a usar esto?» está mal planteada. Asume que se les enseña sobre, por ejemplo, grafos, porque alguien les va a «dar un grafo» en su vida profesional. En cambio, se les enseña sobre grafos para que ustedes los introduzcan al modelar problemas que tengan, y para que los usen como sujetos en proposiciones al aprender a demostrar. Podrían estos sujetos ser otros, como fluidos y campos eléctricos en física, o anillos y ecuaciones diferenciales en matemática. En computación, se usan los objetos de ese campo de estudio.

⁴Nota pedante: La RAE no reconoce el término «correctitud», sino «corrección». Sin embargo, el término «corrección» tiene otros significados que pueden confundirlos en el contexto de una carrera universitaria (e.g. corrección de un parcial, corrección de una aproximación numérica). Es común, entonces, usar el término «correctitud» para referirse a la propiedad de ser correcto. Es la diferencia en Inglés entre «correctness» y «correction», que la RAE no hace.

Sus demostraciones, entonces, cumplen un doble propósito. Tienen que:

1. Convencer al lector de la veracidad de la proposición que afirman. Esto es común a todas las demostraciones matemáticas.
2. Convencer al docente que entienden cómo convencer a *cualquier persona*. El docente ya sabe que la proposición es cierta, no se les va a pedir demostrar proposiciones falsas. Luego, ya está convencido/a de la veracidad de la proposición antes de empezar a leer lo que escribieron. El docente va a evaluar si sus argumentos pueden convencer a *cualquiera*.

Como vimos, *cualquiera* está definido dentro de un contexto. Sus argumentos no van a convencer a un bebé de seis meses, porque no los puede leer. Muchas veces tampoco van a convencer a alguien que no hable español.



¿A quién están convenciendo, entonces? Al interlocutor para el cual la carrera los prepara: Un par suyo, de la comunidad científica. Estamos formando científicos, después de todo. Llegamos entonces a la siguiente definición.

Definición 1.1.1 (Demostración)

Una demostración es un argumento formal de la veracidad de una proposición, que puede convencer a *cualquier* par suyo en la comunidad científica.

Esto nos dice que tenemos que ser un poco paranoicos, si nuestro objetivo es convencer a *cualquier* par. Si somos imprecisos, nuestro lector puede no comprender lo que quisimos decir, y luego no estar convencido. Si no demostramos una proposición que usamos, nuestro lector puede no creernos que esa proposición es cierta, y luego no estar convencido. Tenemos no sólo que estar seguros de la veracidad de lo que estamos probando, sino también saber comunicar las razones por las que *cualquier* par tiene que estar de acuerdo, inclusive si nuestro par nos odia, si nunca vio la proposición antes, si no sabe si es cierta o no antes de empezar a leernos, o si es su primera vez viendo el objeto de estudio sobre el cual estamos hablando (números naturales, programas imperativos, árboles, lo que sea).

Por otro lado, uno puede valerse de que el lector es un par. Sus conocimientos son similares a los que tienen ustedes, y al momento de estar cursando una materia, pueden asumir que el lector cursó y aprobó las materias correlativas. Por ende, si sabemos que $m \leq \frac{n(n-1)}{2}$, podemos concluir que $m < n^2$, porque asumimos que el lector aprobó álgebra del secundario. Debemos *decir* que estamos usando que $m \leq \frac{n(n-1)}{2}$ para concluir que $m \leq n^2$, pero no hace falta demostrar esa conclusión. Si debemos decir quién es n y quién es m , y si estamos usando, por ejemplo, que $n \geq 0$, debemos afirmarlo explícitamente. De nuevo: Estamos siendo un poco paranoicos, porque no queremos que quede ninguna duda en la mente de ningún par que nos lea (en particular, en la mente del docente que nos va a evaluar).

1.3 Formalidad y rigor

Cuando uno habla de «formalidad» en matemática, se está refiriendo a la *forma* en la que algo está escrito. Las demostraciones, entonces, existen en un continuo de formalismo. Ese formalismo va desde argumentos heurísticos, como « $n! + 1$ no es divisible por nadie debajo de n , luego es primo», hasta demostraciones verificables por un asistente de demostración, como la que vimos en el Teorema 4. El formalismo, en el contexto en el que están, se usa para intentar ser riguroso en nuestros argumentos. Un argumento informal como el primero puede ser cierto, como puede ser falso (en particular, $n! + 1$ no siempre es primo, por ejemplo $4! + 1 = 25 = 5^2$).

Razonar informalmente es parte de hacer matemática y computación, pero es sólo la primer parte. Al intentar argumentos, primero los vamos a pensar de forma vaga, no formal. Luego, si queremos asegurarnos de su veracidad, los formalizamos, para estar seguros de ser rigurosos.

La siguiente tabla muestra ejemplos de niveles de formalidad. No es importante que sepan o recuerden cómo tomar derivadas o que sepan lo que es un grafo, el objetivo es que vean el tipo de oración que se usa.

Heurístico	El número de subconjuntos de un conjunto de n elementos es 2^n , porque cada cosa puede o estar o no estar.
Poco formal	Como vemos en el siguiente dibujo, siempre vamos a tener suficientes aristas para que u y v tengan un vecino en común.
Razonablemente formal	<p>Sea $S = 1 + 2 + \dots + n$ la suma de los primeros n enteros positivos. Podemos escribir a S de dos maneras:</p> $S = 1 + 2 + \dots + n$ $S = n + (n - 1) + \dots + 1$ <p>Sumando término a término, cada término suma $n + 1$, con lo cual</p> $2S = n(n + 1)$ <p>y $S = \frac{n(n+1)}{2}$.</p>
Obviamente formal	<p>Queremos ver que $f : \mathbb{R} \rightarrow \mathbb{R}$, $f(x) = 2x$ es diferenciable en todo su dominio. Sea $x_0 \in \mathbb{R}$. Vemos que $\lim_{h \rightarrow 0} \frac{2(x_0+h)-2x_0}{h} = \lim_{h \rightarrow 0} \frac{2x_0+2h-2x_0}{h} = 2$. Luego, la derivada existe en x_0, y es igual a 2. Luego f es diferenciable en todo su dominio.</p> <hr/> <p>Sea X un conjunto con $X = n$, y sea $\mathcal{P}(X) = \{A \subseteq X\}$ su conjunto de partes. Para cada $k \in \{0, \dots, n\}$, sea $\mathcal{P}_k(X) = \{A \subseteq X \mid A = k\}$. Luego, $\mathcal{P}(X) = \bigsqcup_{k=0}^n \mathcal{P}_k(X)$. Entonces,</p> $ \mathcal{P}(X) = \sum_{k=0}^n \mathcal{P}_k(X) $ $= \sum_{k=0}^n \binom{n}{k}$

	Asimismo, $ \mathcal{P}(X) = 2^n$ por la proposición 7. Luego, $\sum_{k=0}^n \binom{n}{k} = 2^n$.
Extremadamente formal	<p>Queremos probar que $1 + 1 = 2$. Los naturales \mathbb{N} están definidos inductivamente como:</p> <ul style="list-style-type: none"> • $0 \in \mathbb{N}$. • $\forall x. x \in \mathbb{N} \Rightarrow S(x) \in \mathbb{N}$. <p>La suma entre naturales está definida inductivamente como:</p> $a + 0 = a$ $a + S(b) = S(a + b)$ <p>Notamos por conveniencia $1 = S(0)$, y $2 = S(S(0))$. Luego,</p> $\begin{aligned} 1 + 1 &= S(0) + S(0) \\ &= S(S(0) + 0) \\ &= S(S(0)) \\ &= 2 \end{aligned}$ <p>Luego $1 + 1 = 2$, que es lo que queríamos demostrar.</p>
Totalmente formal	<pre> inductive N where Z : N S (n : N) : N open N def suma (m n : N) : N := match n with Z => m S n => S (suma m n) def uno : N := S Z def dos : N := S (S Z) theorem teorema_de_lebron : suma uno uno = dos := rfl </pre>

Lo que se espera de ustedes es que puedan escribir y leer demostraciones entre los niveles «Razonablemente formal» y «Obviamente formal».

Mencioné dos veces la palabra «rigor», pero ¿qué significa? En el contexto de demostraciones matemáticas, el rigor significa que **el lector debería poder seguir la demostración paso a paso**, y no preguntarse ¿y esto por qué vale? a cada momento.

Importante

Mientras que la formalidad se refiere a **la forma en la que escribimos**, el rigor se refiere a **la implicación lógica de nuestras oraciones**.

Acá vemos otra vez que el contexto es importante. Podemos asumir, al momento de escribir demostraciones en una materia universitaria, que el lector ha completado las materias correlativas a la que están cursando. Por ejemplo, si ya vieron que diferenciabilidad implica continuidad, pueden asumir que el lector sabe eso, y no tienen que demostrarlo.

Veamos un ejemplo de una demostración no rigurosa:

Ejercicio 1.1.2

Sea $A \subseteq \mathbb{N}$, $A \neq \emptyset$. Demostrar que existe un $a \in A$ tal que para todo $b \in A$, $a \leq b$.



Demostración. Supongamos por contradicción que A no tiene un tal mínimo elemento. Sea x_0 cualquier elemento de A . Como A no tiene un mínimo elemento, sea $x_1 \in \mathbb{N}$ tal que $x_0 > x_1$ (si x_1 no existiera, x_0 sería menor o igual a todo elemento en A , que asumimos no sucede).

Como A no tiene un mínimo elemento, sea $x_2 \in \mathbb{N}$ tal que $x_1 > x_2$. Siguiendo de esta manera tenemos una sucesión infinitamente decreciente de naturales, que no puede suceder. Por lo tanto, lo que asumimos debe haber sido falso, y \mathbb{N} tiene un mínimo elemento. \square

El lenguaje en el que está escrito esto es razonablemente formal. Sin embargo, esta demostración no es rigurosa. Un lector se va a preguntar qué exactamente está pasando cuando decimos «Siguiendo de esta manera». Estamos haciendo alusión a un proceso implícito, y usando alguna noción no definida de límite. Después de todo, todos los procesos que podríamos enumerar en una demostración son finitos, entonces hacer alusión a que primero elegimos x_0 , luego x_1 , luego x_2 , «...», esconde la dificultad en explicitar exactamente qué es ese «...».

Una vez más, esto depende del contexto. Los antiguos griegos usaban este tipo de argumentos todo el tiempo, y esa demostración hubiera sido considerada rigurosa. Fue sólo a principios del 1900, con el trabajo de Ernst Zermelo[2], que nos dimos cuenta que ese tipo de razonamientos, si no tenemos cuidado, llevan a paradojas.

2 ¿Cuándo, qué, y para qué demostramos?

Un computador científico escribe demostraciones cuando quiere establecer sin dudas la veracidad de una proposición lógica. Por ejemplo, si queremos probar que nuestro sistema no va a quedarse sin memoria independientemente de las consultas que arriben, si queremos probar que nuestro programa no se va a ralentizar desmedidamente a medida que se aumente el tamaño de su entrada, o si queremos probar que en nuestro programa con ejecución en paralelo no va a haber «deadlock»⁵ o «livelock»⁶ en ninguna circunstancia. El siguiente es un ejemplo clásico de un algoritmo concurrente que resuelve el problema de los filósofos comensales[3]. Probar que este programa nunca va a sufrir deadlock debe hacerse formalmente, no basta con probarlo varias veces y ver cómo se comporta.

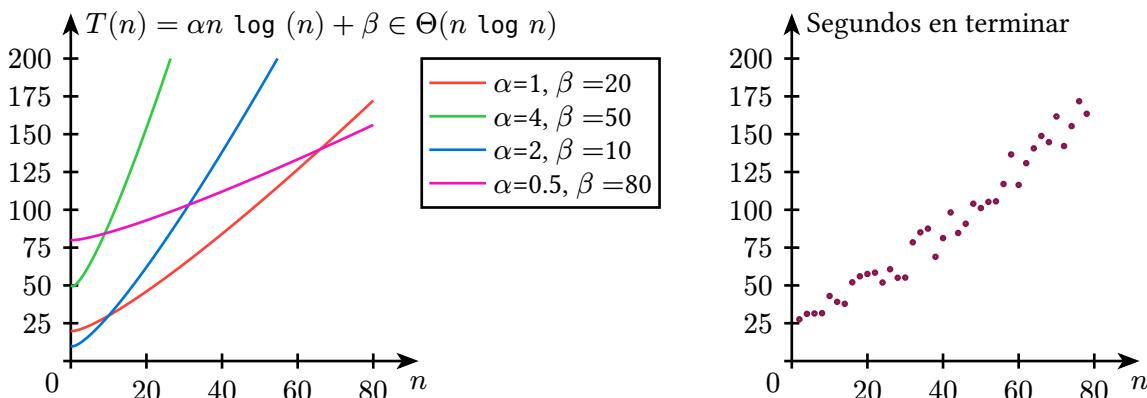
```
1: procedure FILOSOFO( $i$ ,  $n$ , Palitos)
2:   while TRUE do
3:     PENSAR()
4:     AGARRAR-PALITO(Palitos[ $i$ ])
5:     AGARRAR-PALITO(Palitos[( $i + 1$ ) mod  $n$ ])
6:     COMER()
7:     SOLTAR-PALITO(Palitos[( $i + 1$ ) mod  $n$ ])
8:     SOLTAR-PALITO(Palitos[ $i$ ])
9:   end
10: end
```

⁵Un deadlock ocurre cuando hay al menos dos componentes en un programa, cada una esperando a que la otra haga algo, y ninguna pudiendo avanzar.

⁶Un livelock ocurre cuando hay al menos dos componentes en un programa, y cada una cambia su estado en respuesta a la otra, pero ninguna progresa.

No está muy lejos de la verdad el decir que **la ciencia de la computación sin demostraciones es ingeniería**⁷. Vamos a ver luego cómo podemos usar herramientas formales para referirnos a los algoritmos que escribimos, y demostrar propiedades sobre los mismos.

No todas las cosas que queremos saber las vamos a demostrar formalmente. El motivo por el cual la computación es una ciencia, y no sólo matemática, es que hay propiedades de nuestro sistema que vamos a evaluar con argumentos prácticos (por ejemplo, tomando mediciones de nuestro sistema). A veces vamos a usar ambas cosas: Vamos a ver que nuestro programa se comporta bien en tamaños pequeños, y luego mostramos que su comportamiento asintótico es bueno. Al usar nociones asintóticas de complejidad, nos van a quedar constantes sin resolver analíticamente. Para encontrar esas constantes, vamos a medir el sistema real.



Cuando decidimos demostrar algo formalmente, entonces, es porque realmente queremos concluir algo con total seguridad. No nos alcanza con argumentos heurísticos, como mirar qué pasa con «*n* chico», o verificar que es cierto para todos los casos que se nos ocurre.

Recordando el contexto en el que están como alumnos de una carrera universitaria, el otro motivo es, como dijimos antes, convencer a su docente que pueden convencer a cualquier par. Esto a veces va a requerir explicitar argumentos en más detalle que lo que esperan. Veamos un ejemplo de la diferencia. El siguiente es un ejercicio de la práctica 1, y la demostración dada por un alumno, verbatim.

Ejercicio 1.2.1

Calcule la complejidad de un algoritmo que utiliza $T(n)$ pasos para una entrada de tamaño n , donde T cumple:

$$T(n) = 2T(n - 4)$$

Demostración.

$$\begin{aligned} T(n) &= 2T(n - 4) = 2(2T(n - 4 - 4)) \\ &\dots = 2^i T(n - 4i) \end{aligned}$$

Como $n - 4i = 1 \Leftrightarrow i = \frac{n-1}{4}$. Luego,

$$= 2^{\frac{n-1}{4}} T(1) = \left(2^{\frac{1}{4}}\right)^n 2^{-\frac{1}{4}} = O\left(\left(2^{\frac{1}{4}}\right)^n\right)$$

⁷Un ingeniero respondería, «la ingeniería irrelevante y pedante es ciencia de la computación».

□

¿Qué les parece esa demostración? ¿Los convence? ¿Les parece que sería marcado correcto en un parcial?

Esta demostración a mi me convenció al leerla. Si me la presentara un compañero en el trabajo, estaría de acuerdo que $T \in O\left(\left(2^{\frac{1}{4}}\right)^n\right)$. Sin embargo, alguien podría preguntar, «estás restando de a 4 a n hasta llegar a 1 pero, ¿y si n no es congruente con 1 módulo 4?», o «¿cómo sabés qué $T(1) = 1$?» El alumno asumió que $T(1) = 1$, y no pensó realmente en una inducción formal, ni en una definición precisa de T (que hubiera requerido definir su dominio, y asignarle a cada elemento de tal dominio un valor del codominio).

Veamos otra demostración, un poco más formal.

Demostración. Sea $T : \mathbb{N} \rightarrow \mathbb{N}$. Asumo que $T(n) = 2T(n - 4)$ para todo $n \geq 4$. Sea $a = \max(T(0), T(1), T(2), T(3))$.

Definimos $P(n) : T(n) \leq a2^{\frac{n}{4}}$. Vamos a probar que $P(n)$ es cierta para todo $n \in \mathbb{N}$.

Como nuestra recurrencia usa un valor de n menor por 4 que el que recibe, tenemos que probar cuatro casos base, donde no tiene sentido restar 4 a n , pues saldríamos de \mathbb{N} .

Para todo $0 \leq n \leq 3$, tenemos que $T(n) \leq \max(T(0), T(1), T(2), T(3)) = a$. Como $2^{\frac{n}{4}} \geq 1$ para $0 \leq n \leq 3$, y $T(n) \leq a$, tenemos que $T(n) \leq a2^{\frac{n}{4}}$, que prueba nuestros cuatro casos base.

Ahora el paso inductivo. Asumo $P(k)$ para todo $k < n$, quiero probar $P(n)$. Si $n \leq 3$, ya probamos los casos base arriba, y vale P para ellos. Si $n \geq 4$, entonces como sabemos, $T(n) = 2T(n - 4)$. Como $0 \leq n - 4 < n$, podemos usar la hipótesis inductiva, $P(n - 4)$, obteniendo que $T(n - 4) \leq a2^{\frac{n-4}{4}}$. Entonces, como $2^{\frac{n-4}{4}} = 2^{\frac{n}{4}}2^{-1}$, vemos que $T(n) = 2T(n - 4) \leq 2a2^{\frac{n}{4}}2^{-1} = a2^{\frac{n}{4}}$. Esto es precisamente $P(n)$, que es lo que queríamos demostrar.

Como sabemos que $T(n) \leq a2^{\frac{n}{4}}$ para todo $n \in \mathbb{N}$, podemos usar la definición de O , que es que $f \in O(g) \Leftrightarrow \exists \alpha \in \mathbb{R}_{\geq 0}, n_0 \in \mathbb{N}$, tal que para todo $n \in \mathbb{N}$, si $n \geq n_0$, entonces $f(n) \leq \alpha g(n)$.

Podemos acá elegir $\alpha = a$, $n_0 = 0$, $g(n) = 2^{\frac{n}{4}}$, y vemos que $T \in O(g)$. □

¿Qué les pareció esa demostración? Es más larga. Es más detallada. La anterior no «estaba mal», sólo no muestra que el escritor entiende los conceptos que se están evaluando (en este caso, inducción, comportamiento asintótico, y funciones recursivas). Al no tener en cuenta los detalles difíciles sobre inducción en esta demostración, el docente no puede tener confianza que el alumno sabe hacer esto bien, y no va a cometer un error por olvidárselos. De nuevo: Nunca se les va a pedir probar algo falso, por lo cual no importa que sea *cierto* que $T \in O(2^{\frac{n}{4}})$. Lo que importa es si lo saben demostrar.

Para un computador científico, la segunda demostración tiene otro «sabor». Al leerla, nos lleva de la mano, de paso a paso, explicitando cada uno. Uno llega a la conclusión con una seguridad de que cada paso está bien fundado, sin tener que adivinar qué quiere decir cada oración, y sin tener que preguntarse «¿y qué pasa si tal cosa no se cumple?». Las demostraciones que ustedes escriban tienen que dejarlos, y dejar al lector, con la misma sensación. Parece poco serio lo que estoy diciendo, pero

realmente es una buena guía para saber cuándo están haciendo las cosas bien. Esa sensación la van a afilar practicando, haciendo demostraciones, recibiendo correcciones de sus docentes, cometiendo errores, viendo dónde les faltó definir algo, asumieron algo, no se acordaron de un caso, o no entendieron la consigna.

Finalmente, la longitud de la demostración no es algo a intentar emular en sí. De hecho, lo contrario es cierto: si pueden ser precisos y concisos, ¡mejor! Es muy común, sin embargo, que erren por el otro lado, haciendo demostraciones extremadamente escuetas, de una o dos oraciones, pretendiendo que eso le demuestre al lector lo que dijimos que una demostración tiene que mostrar. A veces es porque piensan que, como no saben probar algo, mejor ni lo mencionan y «si pasa pasa»⁸. Otras veces es porque no se dan cuenta que están asumiendo algo. Otras es porque no entendieron qué se les está pidiendo demostrar. En las próximas secciones vamos a ver errores clásicos y cómo no cometerlos.

3 ¿Cómo aprendemos a demostrar?

Nadie aprendió a andar en bicicleta viendo a otros andar en bicicleta. Tampoco van a aprender a demostrar leyendo cómo alguien más demostró. La **única** forma que van a aprender es escribiendo demostraciones ustedes mismos. Por cada segundo que pasan leyendo este documento, sugiero que pasen cinco pensando y escribiendo sus propias demostraciones.

Empiecen demostrando proposiciones simples. Si intentan ambos aprender un tema nuevo (como teoría de grafos) y *al mismo tiempo* aprender a demostrar, les va a resultar demasiado difícil, se van a confundir, y frustrar. Demuestren, primero, propiedades de conjuntos, de enteros, de racionales, de matrices, de objetos con los que ya saben trabajar.

Teorema 6 (Teorema de Euclides)

Hay un número infinito de primos.



Demostración. Esta demostración se encuentra en «Elementos» [1].

Sea $L = [p_1, \dots, p_n]$ una lista finita de números primos distintos. Vamos a mostrar que existe algún número primo que no está en esta lista.

Sea $P = \prod_{p \in L} p$ el producto de todos los números en L . Sea $q = P + 1$. Entonces o bien q es primo, o no es primo.

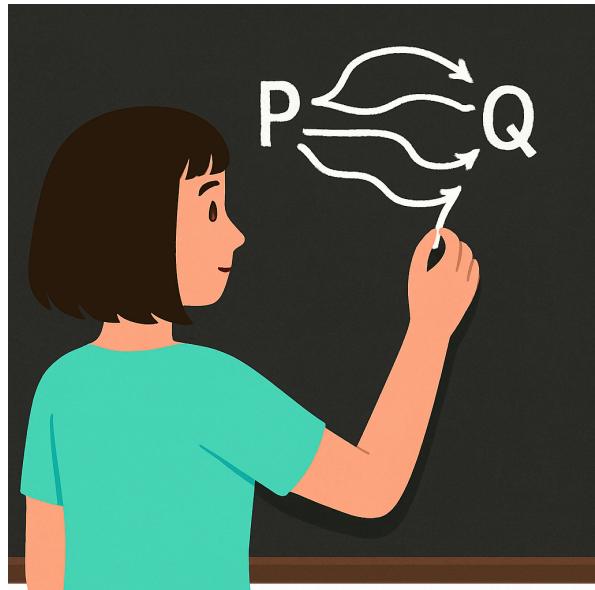
1. Si q es primo, entonces como $q > p$ para todo $p \in L$, al menos hay un número primo (q mismo) fuera de L .
2. Si q no es primo, entonces hay algún factor primo p que divide a q . Entonces $p \in L$, o $p \notin L$.
 1. Si $p \notin L$, encontramos un número primo (p) que no está en L .
 2. Si $p \in L$, entonces $p \mid P$, pues P es el producto de todos los números en L . Como $p \mid P$ y $p \mid q$, entonces $p \mid (q - P)$, es decir $p \mid 1$. Como ningún número primo puede dividir a 1, este caso no puede suceder.

⁸En mi experiencia como docente, esto no funciona nunca. El docente ya sabe cómo demostrar lo que hay que demostrar, y cuál es la parte difícil de la demostración. Si su demostración no menciona y resuelve esa parte explícitamente, es inmediato darse cuenta que el/la alumno/a está mandando fruta.

Concluimos que para toda lista finita L de números primos, existe algún número primo que no está en L , y luego hay infinitos números primos. \square

En este momento, muchos de ustedes no tienen todavía una intuición sobre cuándo una demostración es correcta, versus cuándo les están mintiendo. Más aún, tienen sesgos entendibles por su educación: si leen una demostración en un libro, «debe estar bien», o si se los dice un docente, «debe estar bien». Lo que el practicar les va a dar es la confianza de decir «No, no creo que lo que esté diciendo el docente / libro esté bien.», así como también «Estoy seguro/a de que el argumento que acabo de escribir es correcto.» La matemática es el único lugar donde podemos tener esta certeza, ninguna ciencia puede tenerla. ¡Aprovechémolas! **No acepten como cierto algo sólo porque lo dice un docente.** Si no lo pueden demostrar, no saben si es cierto, y no deberían usarlo.

La matemática que van a ver en la universidad no es como la que vieron en secundario. No hace falta seguir una receta particular, hacer cálculos larguísimos de memoria como derivar 50 polinomios de grado 9, o memorizar patrones de demostración y usar los que el docente quiera. Si se les pide demostrar que P implica Q , el objetivo es que ustedes produzcan una demostración clara y convincente de ese hecho. Si el docente pensó que la demostración tenía que ser por inducción, y ustedes la hicieron partiendo en casos, por contrarecíproco, o por absurdo; no importa. Tienen una enorme flexibilidad a la hora de argumentar, lo que da lugar a la **creatividad** en matemática.



Finalmente, al momento de empezar una demostración, sepan que les debería tomar tiempo. Si esperan que las demostraciones les salgan en 10 minutos, van a cometer errores. Es frecuente que lleven horas. No está *mal* que tomen ese tiempo, y no se deberían sentir mal cuando lo hacen. Tengan paciencia, practiquen con tiempo, entiendan que el objetivo *no* es «que salga el ejercicio». El objetivo es que aprendan a demostrar. Si llegan de P a Q , pero no están seguros si lo que hicieron está bien, *no terminaron el ejercicio*. Si la demostración ni siquiera los convence a ustedes, ¿cómo va a convencer a cualquier par? Vayan lento, vayan seguro, y van a aprender a hacerlo. No hay trucos, sólo sudor y tiza.

💡 Consejo

El aprendizaje no sucede cuando terminan un ejercicio. El aprendizaje sucede cuando piensan, intentan, juegan, fallan, y reflexionan, durante horas. Su objetivo debe ser este, y no el terminar el mayor número de ejercicios.

4 ¿Cómo demostramos?

Ahora miremos cómo se construye una demostración, proceduralmente.

4.1 Formalizar la consigna

En su vida profesional muy rara vez les van a dar un problema pre-formalizado, en términos de secuencias de enteros, permutaciones, o teoría de grafos. En cambio, van a tener que transformar el problema que tienen, a uno donde puedan usar las herramientas que conocen. Esto se llama formalizar. Esta parte es **crucial**: Si formalizan incorrectamente, todo lo que hagan después es totalmente irrelevante. Parte de esto es lectura y comprensión de lo que les piden, y la otra parte es poder usar lenguaje formal.

Ejercicio 1.4.1

Tenemos una máquina vendedora, y queremos devolver cambio. Necesitamos saber si usando sólo monedas de 3 y 5 centavos, podemos devolver cualquier cambio mayor o igual a 8 centavos.

La oración habla sobre monedas, no de algo que veamos directamente en la carrera. Para usar las herramientas que tenemos, lo traducimos a alguna estructura que nos sirva. En la carrera vemos varias, como ser números reales, listas, registros, números enteros, árboles, grafos, lenguajes formales, matrices, redes, autómatas, interrupciones del procesador, etc. De todas esas, tenemos que fijarnos cuál es la que probablemente nos sirva para este problema. Como el enunciado habla sobre armar combinaciones de monedas, queremos algo que modele combinaciones lineales. Podríamos formalizarlo de esta manera:

Ejercicio 1.4.2

Sea $n \in \mathbb{N}$, $n \geq 8$. Probar que existen $a, b \in \mathbb{Z}$ tales que $3a + 5b = n$.

Acá podríamos usar lo que sabemos sobre combinaciones lineales de enteros, es decir, que esto sucede si y sólo si $\gcd(3, 5) | n$. Como $\gcd(3, 5) = 1$, y $1 | n$ para todo $n \in \mathbb{N}$, la proposición es cierta. Sin embargo, **esto está mal formalizado**, y la demostración usando gcd es irrelevante. El problema es que no podemos devolver números negativos de monedas. Por ejemplo, si queremos mostrar que podemos devolver 13 centavos de cambio, es irrelevante decir que $13 = 3 \times 6 + 5 \times (-1)$, porque no podemos devolver «-1» monedas de 5 centavos.

Luego, tenemos que restringir a, b a ser números naturales, no enteros.

Ejercicio 1.4.3

Sea $n \in \mathbb{N}$, $n \geq 8$. Probar que existen $a, b \in \mathbb{N}$ tales que $3a + 5b = n$.

Demostrar esto se hace de forma totalmente diferente. Una forma es usar inducción.

Demostración. Sea $P(n)$ el predicado «existen $a, b \in \mathbb{N}$ tales que $3a + 5b = n$ ». Queremos probar que para todo $n \in \mathbb{N}$ tal que $n > 7$, vale $P(n)$.

Probamos por separado $P(8)$, $P(9)$, y $P(10)$.

- $P(8)$: Tomamos $a = 1$, $b = 1$, pues $3 \times 1 + 5 \times 1 = 8$.
- $P(9)$: Tomamos $a = 3$, $b = 0$, pues $3 \times 3 + 5 \times 0 = 9$.
- $P(10)$: Tomamos $a = 0$, $b = 2$, pues $3 \times 0 + 5 \times 2 = 10$.

Ahora para $n \in \mathbb{N}$, con $n > 10$, podemos asumir que vale $P(k)$ para todo $8 < k < n$. Como $n > 10$, entonces $7 < n - 3 < n$. Luego, tomamos $k = n - 3$, y por hipótesis inductiva ($P(k)$), existen $a', b' \in \mathbb{N}$ tales que $3a' + 5b' = k$. Entonces, tomando $a = a' + 1 \in \mathbb{N}$, $b = b' \in \mathbb{N}$, tenemos que $3a + 5b = 3(a' + 1) + 5b' = 3a' + 5b' + 3 = k + 3 = n$. Luego, vale $P(n)$. \square

Vemos cuán importante es leer detenidamente, y tener cuidado a la hora de formalizar el enunciado. Cuando modelamos algo del mundo real (como monedas) con un concepto matemático (como números naturales), tenemos que pensar si el modelo tiene sentido, y si estamos modelando exactamente los objetos que queremos modelar. Si nuestro modelo es demasiado amplio, como ocurrió con nuestro primer intento de formalización del problema de las monedas, las soluciones que encontramos no van a tener contraparte en el mundo real. Si, por otra parte, nuestro modelo es demasiado restrictivo, puede que no podamos encontrar soluciones que sí existen en el mundo real.

4.2 Comprender qué se nos pide

Una herramienta útil para entender qué hay que probar es pensarlo como una conversación entre dos personas. A nosotros nos van a pedir que demostremos algo, y nosotros tenemos que convencer al interlocutor. Luego, una proposición como la siguiente:

Ejercicio 1.4.4 (Continuidad de $f(x) = e^x$ en x_0)

Para todo $\varepsilon > 0 \in \mathbb{R}$, existe un $\delta > 0 \in \mathbb{R}$, tal que para todo $x \in \mathbb{R}$, si $|x - x_0| < \delta$, entonces $|e^x - e^{x_0}| < \varepsilon$.

Probar un «para todo» es equivalente a la siguiente conversación, entre el que está probando algo (Alicia) y el que está pidiendo que le demuestren algo (Beto):

Beto: No te creo que es cierto, a ver, para $\varepsilon = 0.2$, quién es δ ?

Alicia: Para $\varepsilon = 0.2$, te doy $\delta = 0.4$.

Beto: OK, ponele que $\varepsilon = 0.2$, y $\delta = 0.4$. Te doy un x tal que $|x - x_0| < \delta$, por ejemplo $x = x_0 - 0.3$. Por qué vale que $|e^{x_0-0.3} - e^{x_0}| < 0.2$?

Alicia: [Demostración de que con ese ε y x , tenemos que $|e^x - e^{x_0}| < \varepsilon$.]

5

²Hay maneras de probar la existencia de algo sin darlo explícitamente, y es común en matemática. Inicialmente, sugiero que consideren que demostrar existencia se hace dando un objeto explícito.



Nosotros vamos a tomar el rol de Alicia. Nos van a dar un ε , y tenemos que decir quién es δ . Como nos están dando un ε , nuestro δ puede (y en general va a) depender de ε . Por ejemplo, a veces vamos a concluir que $\delta = \frac{\varepsilon}{8}$. Este es el baile del «para todo / existe». El término «para todo» resulta en «se nos va a dar un». El término «existe» resulta en «tenemos que devolver».

Por el contrario, si tuvieramos probar «existe un x tal que para todo $y \geq x$, $x = y$ », entonces tenemos que dar un x explícito⁹, y mostrar que sin importar cuál y elija Beto, podemos probar que $x = y$.

Luego de que devolvemos ese δ , le sigue otro «para todo», «para todo $x \in \mathbb{R}$ ». Entonces, nos van a dar otro x . A ese «para todo», le sigue un «si», «si $|x - x_0| < \delta$ ». Un «si» nos deja asumir algo - para probar que «si X , entonces Y » (que se escribe $X \Rightarrow Y$), podemos *asumir* X , puesto que de otra manera no hay nada que probar («falso implica todo»). Entonces, esto se traduce en «nos van a dar un x , y podemos asumir que $|x - x_0| < \delta$ ». El «entonces» de un «si» es lo que tenemos que probar. Luego, tenemos que probar que para ese x que nos dieron, vale $|e^x - e^{x_0}| < \varepsilon$.

Noten cómo al igual que ocurre en la conversación, tuvimos que decir quién es δ sin saber quién es x . Entonces, no puede ser que δ dependa de x ! En la conversación, las únicas variables que vinieron antes de δ fueron ε y x_0 . Entonces, sólo de esas dos puede depender δ .

Noten también cómo usé cuantificadores en Español, no usando símbolos. No sugiero enfocarse en escribir usando el mayor número de símbolos posibles. Comparen « $\forall x \in X. \exists y \in Y. x > y \Rightarrow (\exists z \in Z. z = x + y \vee z = x - y)$ », con «Sea $x \in X$. Entonces existe un $y \in Y$, tal que si $x > y$, entonces $x + y \in Z$, o $x - y \in Z$ ». Al saber leer lenguaje natural, nos es más fácil entender qué significa la segunda oración. ¡Esto es a pesar de ser más larga! Cuando escribimos cuidadosamente, usando lenguaje estándar, podemos tener ambas precisión, y comprensión del lector.



4.3 Considerar ejemplos

En general, las cosas que probamos van a ser de la forma $A \Rightarrow B$, con A algo que podemos asumir, y B algo que queremos demostrar. Para demostrar esto, es frecuentemente útil considerar ejemplos de cosas que cumplen A , y ver «por qué» se tiene que cumplir B para ellas. Podemos empezar con ejemplos pequeños, si nuestra estructura tiene alguna noción de «tamaño» (la longitud de una lista, el valor absoluto de un número real, el número de vértices mas aristas de un grafo, el numero de líneas de un programa, el número de reglas de una gramática, etc).

Por ejemplo, veamos el siguiente enunciado formal:

Ejercicio 1.4.5

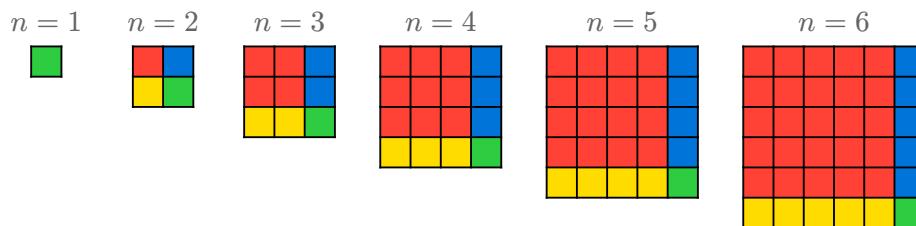
Dar una fórmula cerrada para la suma de los primeros n números naturales impares,
 $\sum_{i=1}^n (2i - 1)$.

Primero les voy a mostrar la serie de pensamientos que sigo al intentar resolver este ejercicio. No es una demostración formal, sólo una serie de ideas.

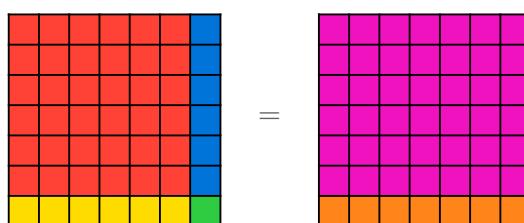
Podemos considerar ejemplos pequeños:

n	1	n	3	4	5
$\sum_{i=1}^n (2i - 1)$	1	$1 + 3 = 4$	$1 + 3 + 5 = 9$	$1 + 3 + 5 + 7 = 16$	$1 + 3 + 5 + 7 + 9 = 25$

Vemos cómo los resultados son los cuadrados consecutivos, 1, 4, 9, 16, 25. Podemos pensar, entonces, en cómo los cuadrados se relacionan con la suma. Si dibujamos estos cuadrados a medida que aumenta n , podemos ver un patrón. Acá el bloque rojo es la suma de los anteriores números impares, y los cuadrados azul, verde, y amarillo, es el nuevo número impar que estamos sumando.



Vemos que cada vez empezamos con el cuadrado anterior, y luego le sumamos una fila amarilla de longitud n , una columna azul de longitud n , y tenemos que restar 1, el cuadrado verde, pues si no lo estaríamos contando dos veces, pues pertenece a ambas la columna y la fila. Esto es lo mismo que tener un rectángulo (rosa) de $n \times (n + 1)$, y agregarle una fila (naranja) de longitud $n + 1$:



Es decir, $n^2 + 2n - 1 = n \times (n + 1) + (n + 1) = (n + 1)^2$. Esto no es una demostración, es sólo una intuición que nos va a guiar hacia cómo demostrarlo. Para ver cómo podemos demostrar esto, podemos usar inducción, pues estamos asumiendo la forma que tiene la suma de los primeros n números impares, y calculando lo mismo para $n + 1$.

Ahora dadas esas ideas, podemos hacer una demostración formal.

Demostración. Sea $S_n = \sum_{i=1}^n (2i - 1)$ la suma de los primeros n números impares. Sea el predicado $P(n) : S_n = n^2$. Queremos probar que para todo $n \in \mathbb{N}$ tal que $n \geq 1$, vale $P(n)$.

- Probamos el caso base, $P(1)$. Tenemos que S_1 es la suma del primer 1 número impar, que es simplemente 1, y luego $S_1 = 1$. Por otro lado, $1 = 1^2$, luego vale $P(1)$.
- Para el paso inductivo, podemos asumir que vale $P(k)$ para todo $1 \leq k < n$, y queremos probar $P(n)$. Por definición de S_n , tenemos que $S_n = S_{n-1} + (2n - 1)$. Como vale $P(n-1)$ por hipótesis inductiva, dado que $1 \leq n-1 < n$, tenemos que $S_{n-1} = (n-1)^2$. Luego, $S_n = (n-1)^2 + (2n - 1) = (n-1)^2 + (n-1) + n = (n-1) \times n + n = n^2$, que es lo que queríamos demostrar.

□

Veamos otro ejemplo de cómo jugar con ejemplos pequeños nos revela cómo lidiar con casos generales.

Ejercicio 1.4.6

Demostrar que para todo $n \in \mathbb{N}$, $\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$.



Veamos qué pasa con $n = 1, 2, 3$.

1. Hay un sólo término, y es 1^2 . Esto es $\frac{1 \times 2 \times 3}{6}$, pero no me es claro «por qué» todavía.
2. $1^2 + 2^2 = 5 = \frac{2 \times 3 \times 5}{6}$. Mmm todavía nada. No creo que el patrón de $2 \times 3 \times \frac{X}{6}$ continúe.
3. $1^2 + 2^2 + 3^2$. ¿Puedo hacer algo con esto más que sumar cada cuadrado? Esto es $1 + 2 \times 2 + 3 \times 3$, o quizás, $1 + 2 + 2 + 3 + 3 + 3$. Me queda algo sospechoso, $(1 + 2 + 3) + (2 + 3) + 3$. Como sumas de no-cuadrados, hasta n , que se van haciendo más cortas. Esto es $\frac{n(n-1)}{2} + \left(\frac{n(n-1)}{2} - 1\right) + \left(\frac{n(n-1)}{2} - 3\right)$. El patrón queda algo como $\sum_{i=0}^{n-1} N - \frac{i(i+1)}{2}$, con $N = \frac{n(n+1)}{2}$.

A ver, pensémoslo con cuidado. Con $n = 4$, tenemos

$$\begin{aligned} \sum_{i=0}^n i^2 &= 0^2 + 1^2 + 2^2 + 3^2 + 4^2 \\ &= 0 + 1 \times 1 + 2 \times 2 + 3 \times 3 + 4 \times 4 \\ &= 1 + 2 + 2 + 3 + 3 + 3 + 4 + 4 + 4 + 4 \\ &= (1 + 2 + 3 + 4) + (2 + 3 + 4) + (3 + 4) + (4) \\ &= N + (N - 1) + (N - (1 + 2)) + (N - (1 + 2 + 3)) \\ &= N + \left(N - 1 \times \frac{2}{2}\right) + \left(N - 2 \times \frac{3}{2}\right) + \left(N - 3 \times \frac{4}{2}\right) \end{aligned}$$

OK, esto es entonces $\sum_{i=1}^n N - \frac{(i-1)i}{2} = Nn - \sum_{i=1}^n \frac{(i-1)i}{2}$. Ese $\frac{(i-1)i}{2}$ se va a convertir en $\frac{i^2 - i}{2}$, y si separo eso, ¡me va a quedar otra vez la suma de cuadrados!

Juguemos, entonces. Llamemos $X = \sum_{i=0}^n i^2 = \sum_{i=1}^n i^2$.

$$\begin{aligned}
X &= Nn - \sum_{i=1}^n \frac{(i-1)i}{2} \\
&= \frac{n^2(n+1)}{2} - \left(\frac{1}{2}\right) \left(\sum_{i=1}^n i^2 - i \right) \\
&= \frac{n^2(n+1)}{2} - \left(\frac{1}{2}\right) \left(\sum_{i=1}^n i^2 \right) + \left(\frac{1}{2}\right) \sum_{i=1}^n i \\
&= \frac{n^2(n+1)}{2} - \frac{X}{2} + \frac{n(n+1)}{4}
\end{aligned}$$

Multiplicamos todo por 4, y pasamos todos los X a la izquierda.

$$\begin{aligned}
6X &= 2n^2(n+1) + n(n+1) \\
&= n(n+1)(2n+1)
\end{aligned}$$

Con lo cual $X = \frac{n(n+1)(2n+1)}{6}$, que ¡es lo que quería!

Habiendo jugado con ejemplos pequeños, ahora sabemos «por qué» la proposición es cierta. Resta hacer un argumento convincente de que el patrón que encontramos se va a repetir para todo $n \in \mathbb{N}$. Usamos la herramienta formal de inducción, para hacer riguroso el argumento.

Demostración. Vamos a probar la proposición $P(n) : \sum_{i=0}^n i^2 = \sum_{i=1}^n \frac{n(n+1)-(i-1)i}{2}$, para todo $n \in \mathbb{N}$.

- $P(0)$. $\sum_{i=0}^1 i^2 = 0^2 = 0$, mientras que $\sum_{i=1}^0$ (lo que sea) = 0, porque hay cero términos en la suma. Luego vale $P(0)$.
- $P(n) \Rightarrow P(n+1)$. Entonces:

$$\begin{aligned}
\sum_{i=0}^{n+1} i^2 &= \left(\sum_{i=0}^n i^2 \right) + (n+1)^2 \\
&= \left(\sum_{i=1}^n \frac{n(n+1)-(i-1)i}{2} \right) + (n+1)(n+1) \\
&= \left(\sum_{i=1}^{n+1} \frac{n(n+1)-(i-1)i}{2} \right) + (n+1)(n+1), \text{ pues el último término es } 0 \\
&= \sum_{i=1}^{n+1} \left(\frac{n(n+1)-(i-1)i}{2} + n+1 \right), \text{ sumo } n+1 \text{ a cada uno de los } n+1 \text{ términos} \\
&= \sum_{i=1}^{n+1} \frac{2n+2+n(n+1)-(i-1)i}{2} \\
&= \sum_{i=1}^{n+1} \frac{(n+1)(n+2)-(i-1)i}{2}
\end{aligned}$$

que es lo que queríamos demostrar.

Habiendo probado $P(n)$ para todo $n \in \mathbb{N}$, veamos cómo probar que $\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$ para todo $n \in \mathbb{N}$. Sea $n \in \mathbb{N}$, y llamemos $X = \sum_{i=0}^n i^2$.

$$\begin{aligned}
X &= \sum_{i=0}^n i^2 = \sum_{i=1}^n \frac{n(n+1) - (i-1)i}{2} \\
&= n^2 \frac{n+1}{2} - \sum_{i=1}^n \frac{i(i-1)}{2} \\
&= n^2 \frac{n+1}{2} - \left(\frac{1}{2}\right) \sum_{i=1}^n i^2 + \left(\frac{1}{2}\right) \sum_{i=1}^n i \\
&= n^2 \frac{n+1}{2} - \left(\frac{1}{2}\right) X + \frac{n(n+1)}{4}
\end{aligned}$$

Multiplicando por 4 y pasando las X a la izquierda...

$$\begin{aligned}
6X &= 2n^2(n+1) + n(n+1) \\
&= n(n+1)(2n+1)
\end{aligned}$$

Y por lo tanto $\sum_{i=0}^n i^2 = X = \frac{n(n+1)(2n+1)}{6}$, que es lo que queríamos demostrar. \square

Nuestro cerebro es muy eficiente para ser perezoso, y frecuentemente al hacer cuentas repetitivas, vamos a encontrar patrones que nos ahorren esfuerzo, y al mismo tiempo revelen propiedades sobre los objetos que estamos mencionando. Jamás se me hubiera ocurrido distribuir un $n+1$ a cada uno de los $n+1$ términos de la sumatoria, si no fuera porque fue exactamente lo que hice para $n=3$ y $n=4$, notando ese patrón en un caso chico.

4.4 Estrategias de demostración

Una vez que entendemos qué es lo que se no pide probar y tenemos una idea intuitiva de por qué funciona, podemos planear estructuras que va a tener nuestra demostración. En general vamos a usar varias de ellas en la misma demostración.

4.4.1 Inducción

Si los objetos con los que estamos trabajando tienen una estructura recursiva, como los números naturales, los árboles, los grafos, o las cadenas de texto, entonces podemos considerar inducción como técnica de demostración.

Vamos a plantear un predicado sobre los naturales, P . Vamos a probar que P vale para los primeros k números naturales (es posible que $k=0$, o $k=1$, o $k>1$). Luego vamos a probar que dado un $n \geq k$, si vale $P(j)$ para todo $j < n$, entonces vale $P(n)$. Esto nos permite concluir que vale $P(n)$ para todo $n \in \mathbb{N}$.

Veamos un caso simple, con $k=1$.

Proposición 1.4.7

Para todo $n \in \mathbb{N}$, $\sum_{i=1}^n i = \frac{n(n+1)}{2}$.



Demostración. Sea el predicado $P(n) : \sum_{i=1}^n i = \frac{n(n+1)}{2}$. Vamos a probar que vale $P(n)$ para todo $n \in \mathbb{N}$.

- Caso base, $P(0)$. Tenemos que $\sum_{i=1}^0 i = 0$, y $\frac{0(0+1)}{2} = 0$, luego vale $P(0)$.
- Paso inductivo, $P(n)$ con $n \geq 1$. Asumimos $P(j)$ para todo $j < n$, queremos probar $P(n)$.
Tenemos:

$$\begin{aligned}
\sum_{i=1}^n i &= \left(\sum_{i=1}^{n-1} i \right) + n \\
&= \frac{(n-1)n}{2} + n, \text{ por hipótesis inductiva } P(n-1) \\
&= \frac{n^2 - n + 2n}{2} \\
&= \frac{n(n+1)}{2},
\end{aligned}$$

que es lo que queríamos demostrar.

□

Veamos un caso con $k = 0$. Quizás les resulte extraño que no sean necesarios casos base. Esto sucede cuando la demostración de $P(n)$ no siempre usa $P(n-r)$ para algún r , sino que la estructura inductiva es otra. En el siguiente caso, la estructura es multiplicativa, no aditiva. Los «casos base» son los números primos, si uno quisiera llamarlos así.

Proposición 1.4.8

Sea $n \in \mathbb{N}$, $n > 0$. Entonces n se puede escribir como un producto de números primos.

♡

Demostración. Sea el predicado $P(n)$: «Si $n > 0$, entonces n se puede escribir como un producto de números primos». Vamos a probar que vale $P(n)$ para todo $n \in \mathbb{N}$ con $n > 0$.

Sea $n \in \mathbb{N}$, $n > 0$. Asumimos que vale $P(j)$ para todo $j < n$. Como $n > 0$, o bien n es primo, o no lo es.

- Si n es primo, entonces n se puede escribir como un producto de números primos, simplemente n .
- Si n no es primo, entonces existen $a, b \in \mathbb{N}$, con $1 \leq a, b < n$, tales que $n = a \times b$. Como $a < n$ y $b < n$, sabemos que valen $P(a)$ y $P(b)$. Como $a > 0$ y $b > 0$, $P(a)$ y $P(b)$ nos dan formas de escribir a a y b como productos de números primos, $a = q_1 \times \dots \times q_s$ y $b = r_1 \times \dots \times r_t$. Luego vemos que $n = q_1 \times \dots \times q_s \times r_1 \times \dots \times r_t$ es una forma de escribir n como producto de números primos, que es lo que queríamos demostrar.

□

ⓘ Nota

Noten en la anterior demostración que $P(a)$ y $P(b)$ son implicaciones. No nos dicen que existe una factorización de a y b como producto de primos, nos dicen que **si** $a > 0$, **entonces** existe una factorización de a como producto de primos. Lo mismo para b . Por eso es importante que tomamos a y b mayores que cero, pues de otra forma no sirve la hipótesis inductiva.

Una generalización de la inducción es la inducción en conjuntos bien-fundados, también conocida

como inducción Noetheriana¹⁰ o inducción estructural. No es necesario para los temas de este libro, pero es útil para entender una forma de inducción sobre tipos de datos algebraicos.

Definición 1 (Inducción estructural)

Sea A un conjunto, y \preceq un orden parcial sobre A . Escribimos $a \prec b$ para significar que $a \preceq b$ y $a \neq b$. Dado un subconjunto M de A , llamamos a un elemento $m \in M$ **\preceq -minimal** cuando no existe ningún $m' \in M$ tal que $m' \prec m$.

Decimos que (A, \preceq) es un **orden bien fundado** si para todo subconjunto no vacío M de A , M tiene un elemento \preceq -minimal.

Sea P un predicado sobre A , con (A, \preceq) un orden bien fundado. Para probar que vale $P(a)$ para todo $a \in A$, alcanza con probar dos cosas:

- $P(a)$ vale para todo elemento \preceq -minimal de A .
- Para todo $a \in A$ no \preceq -minimal, si vale $P(a')$ para todo $a' \prec a$, entonces vale $P(a)$.

Por ejemplo, veamos cómo usar inducción estructural para probar una propiedad sobre árboles.

Ejercicio 1.4.9

Definimos el tipo de datos algebraico Tree como:

```
data Tree = Nil | Node Int Tree Tree
```

Es decir, un Tree es o bien Nil, o bien es un nodo que contiene un entero y dos Trees, cada uno posiblemente Nil. Luego definimos:

```
espejar :: Tree -> Tree
espejar Nil = Nil
espejar (Node x l r) = Node x (espejar r) (espejar l)
```

Demostrar que para todo $t :: Tree$, vale que $espejar (espejar t) = t$.

Demostración. Sea T el conjunto de todos los Tree. Definimos un orden parcial \preceq sobre T como: para todo $t_1, t_2 \in T$, tenemos $t_1 \preceq t_2$ cuando t_1 es un sub-árbol de t_2 . Es decir, si $t_1 \preceq t_2$, entonces o bien $t_1 = t_2$, o bien $t_2 = Node x l r$, y $t_1 \preceq l$ o $t_1 \preceq r$.

El único elemento minimal es Nil, pues para todo $t = Node x l r$, tenemos que $l \preceq t$, y $r \preceq t$, con $l \neq t$ y $r \neq t$, y luego $l \prec t$ y $r \prec t$, con lo cual t no es minimal.

Probemos la propiedad $P(t)$: « $espejar (espejar t) = t$ », para todo $t \in T$, usando inducción estructural sobre (T, \preceq) .

- Caso base, $t = Nil$. Probemos $P(Nil)$. Tenemos que $espejar (espejar Nil) = espejar Nil = Nil$, luego vale $P(Nil)$.
- Paso inductivo, $t = Node x l r$. Como $l \prec t$ y $r \prec t$, podemos asumir que vale $P(l)$ y $P(r)$, queremos probar $P(t)$. Tenemos:

```
espejar (espejar (Node x l r))
= espejar (Node x (espejar r) (espejar l))
```

¹⁰El nombre es en honor a la matemática Emmy Noether (1882 - 1935), que fue la primera en usar una técnica similar para probar propiedades sobre ciertos anillos, hoy llamados anillos Noetherianos.

```
= Node x (espejar (espejar l)) (espejar (espejar r))
= Node x l r -- por hipótesis inductiva P(l) y P(r)
```

que es lo que queríamos demostrar.

□

Para estructuras finitas, la inducción estructural es equivalente a la inducción matemática «clásica» sobre \mathbb{N} , pues podemos asignar a cada elemento de la estructura un número natural que mide su «tamaño», que induce un orden parcial, y hacer inducción sobre ese número. Veamos un ejemplo.

Ejercicio 1.4.10

Definimos el tipo de datos algebraico List como:

```
data List = Nil | Cons Int List
```

Es decir, una List es o bien Nil, o bien es un par formado por un entero y otra List. Luego definimos:

```
length :: List -> Int
length Nil = 0
length (Cons _ xs) = 1 + length xs

concat :: List -> List -> List
concat Nil ys = ys
concat (Cons x xs) ys = Cons x (concat xs ys)
```

Demostrar que para todas dos $xs :: List$ y $ys :: List$, vale que $length (concat xs ys) = length xs + length ys$.

♦

Demostración. Vamos a probar el predicado sobre los números naturales $P(n)$: «Para toda $xs :: List$ tal que $length xs = n$, y para toda $ys :: List$, vale que $length (concat xs ys) = n + length ys$ ».

- Caso base, $P(0)$. Sea $xs :: List$ tal que $length xs = 0$. Entonces, usando la definición de length, $xs = Nil$. Sea $ys :: List$ cualquiera. Entonces:

```
length (concat xs ys)
= length (concat Nil ys)
= length ys
= 0 + length ys
```

que es lo que queríamos demostrar.

- Paso inductivo. Tenemos $n > 0 \in \mathbb{N}$, sabemos que vale $P(j)$ para todo $0 < j < n$, y queremos probar $P(n)$. Sea $xs :: List$ tal que $length xs = n$. Como $n > 0$, tenemos que $xs = Cons x xs'$, para algún $x :: Int$, y $xs' :: List$. Además, como $length xs = n$, tenemos que $length xs' = n - 1$. Sea $ys :: List$ cualquiera. Entonces:

```
length (concat xs ys)
= length (concat (Cons x xs') ys)
= length (Cons x (concat xs' ys))
= 1 + length (concat xs' ys)
```

```
= 1 + (n - 1) + length ys -- por hipótesis inductiva P(n-1)
= n + length ys
```

que es lo que queríamos demostrar.

Notar que usamos $P(n - 1)$ para saber que $\text{length}(\text{concat } xs' \text{ } ys) = (n - 1) + \text{length } ys$, pues $\text{length } xs' = n - 1$.

□

⚠️ Advertencia

Sugiero ser formal cuando hacen inducción. Es muy común que se confundan cuando son informales. El siguiente es un ejemplo de una demostración incorrecta de una proposición falsa, por ser informal y no definir explícitamente una proposición sobre los números naturales, con cuantificadores.

Proposición falsa 1.4.11

Sea $P(n)$: Para todo conjunto S de enteros de tamaño n , si $1 \in S$, entonces $\max(S) = |S|$.

♡

Demostración incorrecta.

1. Caso base. Sea S un conjunto de tamaño 1, con $1 \in S$. Entonces $S = \{1\}$, y luego $\max(S) = 1 = |S|$.
2. Paso inductivo. Asumimos $P(n)$, vamos a probar $P(n + 1)$. Sea S un conjunto de tamaño n , tal que $1 \in S$. Por $P(n)$ sabemos que $\max(S) = |S|$. Construimos $S' = S \cup \{k\}$, con $k = |S| + 1 = n + 1$. Como $\max(S) = |S| = n$, en particular $n + 1 \notin S$, y luego $|S'| = n + 1$. Además, $1 \in S'$, porque $1 \in S$, y $S \subseteq S'$. Entonces, $\max(S') = k = n + 1 = |S'|$, que es lo que queríamos demostrar.

□

Esto es claramente incorrecto, puesto que existe $S = \{1, 100\}$, con $1 \in S$, pero $\max(S) = 100 \neq |S| = 2$. El problema es que la demostración es demasiado informal, y termina siendo insuficientemente rigurosa. No usa $P(n)$, sólo lo menciona. Para probar $P(n + 1)$, tendríamos que probar que *cualquier* conjunto S' de tamaño $n + 1$ que contiene al 1, cumple que $\max(S') = |S'|$. Pero en la demostración, sólo se prueba para un conjunto particular, el que se construye agregando un elemento específico al conjunto S . El problema es que no todo conjunto de tamaño $n + 1$ que contiene al 1, se puede construir de esa forma a partir de un conjunto de tamaño n que contiene al 1.

⚠️ Advertencia

Si nuestra demostración de $P(n)$ requiere que $P(n - 1), P(n - 2), \dots, P(n - k)$ sea cierto para algún $k \geq 1$ fijo, entonces vamos a necesitar k casos base. Esto es porque nuestra demostración no va a tener sentido cuando $n < k$, porque estaríamos diciendo que $P(n - k)$ vale, con $n - k < 0$, que no tiene sentido pues P es una proposición sobre los naturales.

Veamos primero una demostración correcta que toma esto en cuenta, y luego tres que son incorrectas por no hacerlo.

Ejercicio 1.4.12 (Fórmula cerrada para la sucesión de Fibonacci)

Sea $a_0 = 0, a_1 = 1$, y para todo $n > 1$, definamos $a_n = a_{n-1} + a_{n-2}$. Entonces para todo $n \in \mathbb{N}$, tenemos $a_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$, con $\varphi = \frac{1+\sqrt{5}}{2}$, y $\psi = \frac{1-\sqrt{5}}{2}$.



Demostración. La propiedad que vamos a probar por inducción es $P(n) : a_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$. Como a_n está definida en términos de a_{n-1} y a_{n-2} , vamos a querer decir algo sobre a_{n-1} y a_{n-2} , lo cual significa que vamos a usar $P(n-1)$ y $P(n-2)$. Luego, tenemos dos casos base.

- Caso base $n = 0$. Si $n = 0$, entonces $a_n = a_0 = 0$, por definición. $\varphi^0 = \psi^0 = 1$, y también $\varphi^n - \psi^n = 0$, y por lo tanto $a_n = a_0 = 0 = \frac{\varphi^0 - \psi^0}{\sqrt{5}} = \frac{\varphi^n - \psi^n}{\sqrt{5}}$, que es lo que queríamos demostrar.
- Caso base $n = 1$. Si $n = 1$, entonces $a_n = a_1 = 1$, por definición. $\varphi^1 = \frac{1+\sqrt{5}}{2}$, y $\psi^1 = \frac{1-\sqrt{5}}{2}$, luego $\varphi^n - \psi^n = \varphi^1 - \psi^1 = \frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2} = 2\frac{\sqrt{5}}{2} = \sqrt{5}$, y luego $\frac{\varphi^n - \psi^n}{\sqrt{5}} = 1 = a_0 = a_n$, que es lo que queríamos demostrar.
- Paso inductivo. Podemos asumir $P(n-1)$ y $P(n-2)$, y queremos probar $P(n)$. Vemos que $\varphi^{-1} = \frac{\sqrt{5}-1}{2}$, y de ahí vemos que $1 + \varphi^{-1} = \varphi$, y que $1 - \varphi = -\frac{1}{\varphi}$. Realmente ambos hechos son consecuencia de que φ es una de las raíces de $\varphi + 1 = \varphi^2$. Notemos que $\psi = 1 - \varphi = -\frac{1}{\varphi}$. Entonces:

$$\begin{aligned}
 a_n &= a_{n-1} + a_{n-2} \\
 &= \frac{\varphi^{n-1} - \psi^{n-1}}{\sqrt{5}} + \frac{\varphi^{n-2} - \psi^{n-2}}{\sqrt{5}} \\
 &= \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^{n-1} - \left(-\frac{1}{\varphi}\right)^{n-1} \right] + \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^{n-2} - \left(-\frac{1}{\varphi}\right)^{n-2} \right] \\
 &= \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^{n-1} - \left(-\frac{1}{\varphi}\right)^{n-1} \right] + \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^{n-1} \frac{1}{\varphi} - \left(-\frac{1}{\varphi}\right)^{n-1} (-\varphi) \right] \\
 &= \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^{n-1} - \left(-\frac{1}{\varphi}\right)^{n-1} + \varphi^{n-1} \frac{1}{\varphi} - \left(-\frac{1}{\varphi}\right)^{n-1} (-\varphi) \right] \\
 &= \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^{n-1} \left(1 + \frac{1}{\varphi}\right) - \left(-\frac{1}{\varphi}\right)^{n-1} (1 - \varphi) \right] \\
 &= \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^{n-1} \varphi - \left(-\frac{1}{\varphi}\right)^{n-1} \left(-\frac{1}{\varphi}\right) \right] \\
 &= \left(\frac{1}{\sqrt{5}}\right) \left[\varphi^n - \left(-\frac{1}{\varphi}\right)^n \right] \\
 &= \frac{\varphi^n - \psi^n}{\sqrt{5}}
 \end{aligned}$$



Ahora veamos qué pasa si no somos cuidadosos al usar la hipótesis inductiva.



Proposición falsa 1.4.13

En cualquier conjunto de caballos, todos los caballos son del mismo color.



Demostración incorrecta. Sea S un conjunto de caballos de n elementos. Si $n = 1$, S tiene un único caballo, y obviamente es del mismo color que todos los caballos de S . Si $n > 1$, entonces sea x cualquier caballo en S . Si sacamos a x de S , obtenemos un conjunto $S' = S \setminus \{x\}$ de $n - 1$ caballos. Por hipótesis inductiva, todos los caballos en S' son del mismo color. Ahora sea y otro caballo, distinto de x . Por hipótesis inductiva, en $T = S \setminus \{y\}$, todos los caballos son del mismo color. Como x e y tienen el mismo color que todos los otros caballos, entonces x e y también son del mismo color entre sí, y luego todos los caballos en S son del mismo color.

Por inducción, en cualquier conjunto de caballos, todos los caballos son del mismo color. \square

Sin embargo, han visto dos caballos de colores distintos. ¿Cómo puede ser? El error está en ser informal, pues al decir « x e y tienen el mismo color que todos los otros caballos», e intentar argumentar algo con eso, uno tiene que asegurarse de que el conjunto «todos los otros caballos» no es vacío, pues en ese caso no podemos inferir nada. Luego, esta demostración se cae en el caso $n = 2$. Al ser informal y razonar vagamente, miente.

Otra demostración errónea más. ¿Pueden encontrar el error?

Proposición falsa 1.4.14

Para todo $n \in \mathbb{N}$, $2^n = 1$.



Demostración incorrecta. Vamos a probar que vale $P(n)$ para todo $n \in \mathbb{N}$, con $P(n) : 2^n = 1$.

1. Caso base, $P(0)$. Es cierto que $2^0 = 1$, y luego $P(0)$ es cierta.
2. Paso inductivo. Asumimos que vale $P(j)$ para todo $j < n + 1$, y probemos $P(n + 1)$.

Manipulamos algebraicamente:

$$\begin{aligned}
2^{n+1} &= 2^{2n-(n-1)} \\
&= \frac{2^{2n}}{2^{n-1}} \\
&= \frac{2^n 2^n}{2^{n-1}} \\
&= 1 \times \frac{1}{1} \text{ por hipótesis inductiva, tres veces} \\
&= 1
\end{aligned}$$

que es lo que queríamos demostrar. □

El error vino de usar $P(n - 1)$. Esto no tiene sentido cuando estamos probando $P(1)$, porque entonces $1 = n + 1$ y entonces $n = 0$, y no tiene sentido decir $n - 1$. No fuimos cuidadosos al asumir que $n > 1$, lo cual nos hubiera marcado que debemos probar $P(1)$ por separado, no sólo $P(0)$.

¿Pueden detectar dónde está el error en la siguiente demostración?

Proposición falsa 1.4.15

Probar que para todo $n \in \mathbb{N}$, $5n = 0$.



Demostración incorrecta. Sea el predicado $P(n) : 5n = 0$. Vamos a probar $P(n) \forall n \in \mathbb{N}$ por inducción.

- $P(0)$. Queremos probar $P(0)$, que significa $5 \times 0 = 0$, y esto es cierto. Luego vale $P(0)$.
- $n > 0 \Rightarrow P(n)$. Sean $i, j \in \mathbb{N}$, con $i < n, j < n$, tales que $i + j = n$. Entonces por hipótesis inductiva vale $P(i)$ y $P(j)$, entonces $5i = 0$ y $5j = 0$. Entonces, $5n = 5(i + j) = 5i + 5j = 0 + 0 = 0$, lo cual prueba $P(n)$.



El error está en asumir que existen naturales $i < n, j < n$, con $i + j = n$. Esto sólo es cierto si $n \geq 2$, y entonces nuestra demostración falla para $n = 1$, y se cae la inducción. No fuimos cuidadosos, y nos faltó el caso base $n = 1$.

4.4.2 Correctitud de ciclos en algoritmos

Frecuentemente vamos a probar propiedades sobre algoritmos que usan ciclos. La herramienta principal que tenemos para esto es el teorema del invariante. Esto no es nada más que inducción en el número de iteraciones, con un formalismo al rededor para evitar que cometan errores (como, por ejemplo, olvidarse de probar que el ciclo efectivamente termina).

Para usar el teorema del invariante, necesitamos definir cinco cosas:

1. Una precondición P . Esto es algo que asumimos que vale antes de comenzar el ciclo.
2. Una postcondición Q . Esto es algo que queremos probar que vale al terminar el ciclo.
3. Un invariante I . Esto es algo que es válido antes de cada iteración, y después de cada iteración (pero no necesariamente *durante* una iteración).
4. Una guarda B , que nos dice si ejecutaremos la próxima iteración del ciclo, o no.
5. Una función variante v , que nos obliga a terminar el ciclo cuando llega a cero.

Y tenemos que demostrar las siguientes seis proposiciones:

1. La precondición vale antes de comenzar el ciclo.
2. La precondición implica el invariante.
3. La función variante decrece con cada iteración.
4. Si la función variante es cero, la guarda es falsa.
5. El invariante y la negación de la guarda implican la postcondición.
6. La guarda y el invariante al comenzar la iteración implican el invariante al terminar la iteración.



En Algoritmos 1, aprenden a especificar estas proposiciones, y a demostrar estas proposiciones. En la sección de ejemplos muestro varios, pero para mostrarles uno acá, vamos a probar la correctitud del algoritmo de exponenciación en tiempo logarítmico.

```

1: procedure EXP( $a \in \mathbb{N}, n \in \mathbb{N}$ )
2:   if  $n = 0$  then
3:     return 1
4:   end
5:    $k \leftarrow n$ 
6:    $y \leftarrow 1$ 
7:   while  $k > 1$  do
8:     if  $k \bmod 2 = 1$  then
9:        $y \leftarrow x \times y$ 
10:       $k \leftarrow k - 1$ 
11:    end
12:     $x \leftarrow x^2$ 
13:     $k \leftarrow k/2$ 
14:  end
15:  return  $x \times y$ 
16: end

```

Demostración. Llamemos x_0 al valor de x al comenzar el algoritmo. Queremos probar que el programa devuelve x_0^n .

1. La precondición del algoritmo es $n > 0, k = n, y = 1, x = x_0$.
2. La postcondición del ciclo es $x \times y = x_0^n$.
3. El invariante es $1 \leq k \leq n \wedge x^k y = x_0^n$.
4. La guarda es $k > 1$.
5. La función variante es $k - 1$.

Probemos el teorema del invariante para este ciclo.

1. Estamos asignando $k \leftarrow n, y \leftarrow 1$, luego valen $k = n, y = 1$. Lo primero que hace nuestro programa es salir si $n = 0$, en cuyo caso devuelve la respuesta correcta y no ejecuta nada más. Luego, al comenzar el ciclo, sabemos que $n \in \mathbb{N}, n \neq 0$, y luego $n > 0$. Finalmente, antes de comenzar el ciclo no modificamos x , con lo cual sigue valiendo $x = x_0$.
2. Como vale la precondición, entonces $k = n$, y por ende $k \leq n$. Asimismo, la precondición nos dice que $n > 0$, y por ende $k > 0$, o lo que es equivalente, $k \geq 1$. Juntando las dos oraciones anteriores, tenemos que $1 \leq k \leq n$. Finalmente, como la precondición nos dice que $y = 1$ y $x_0 = x$, tenemos $x^k y = x^n y = x^n = x_0^n$, que prueba el invariante.

3. En cada iteración, estamos dividiendo a k por 2, y quizás restándole 1, con lo cual siempre va a decrecer, porque $k > 1$ (sólo podría no-decrecer si $k = 0$, porque entonces $k/2 = k$). Luego, como k decrece, también $k - 1$, la función variante, decrece.
4. Si la función variante es cero, entonces $k - 1 = 0$, y luego $k = 1$. Como la guarda es $k > 1$, el que la función variante sea cero obliga a que la guarda no se cumpla.
5. La negación de la guarda nos dice que $k \leq 1$. El invariante nos dice que $1 \leq k$. Luego, sabemos que $k = 1$. El invariante también nos dice que $x^k \times y = x_0^n$, entonces sabemos que $x^1 \times y = x \times y = x_0^n$, que es la postcondición.
6. Sabemos que $k > 1$ porque vale la guarda. Partimos en casos, dependiendo de si $k \bmod 2 = 0$ o $k \bmod 2 = 1$.
 - Si $k = 2k'$, entonces entramos al condicional. Lo que hacemos es cuadrar x obteniendo x' , y dividir k por dos, obteniendo k' . Como vale la guarda, sabemos que $k > 1$, y como $k \in \mathbb{N}$, tenemos $k \geq 2$. Como $k = 2k'$, vemos que $k' \geq 1$, que es la primer parte del invariante. Empezamos la iteración con $x^k y = x_0^n$, es decir $x^{2k'} y = x_0^n$. Usando el álgebra de exponentiación, vemos que $x^{2k'} = (x^2)^{k'}$, y $k' = k/2$. Luego, vemos que vale $(x^2)^{k'} y = x_0^n$, o $x'^{k'} y = x_0^n$, que es el invariante al terminar la iteración.
 - Si $k = 2k' + 1$, entonces entramos al condicional. El efecto de la iteración en este caso es transformar k en k' , x en $x' = x^2$, y $y' = x \times y$. Como $k > 1$, y $k \in \mathbb{N}$, sabemos que $k \geq 2$, pero como k es impar, $k \geq 3$. Como $k = 2k' + 1$, entonces, tenemos $2k' \geq 2$, y luego $k' \geq 1$, que es la primer parte del invariante. Como vale el invariante al comenzar la iteración, sabemos que $x^{2k'+1} y = x_0^n$. Nuevamente usando álgebra de exponentiación, obtenemos $x_0^n = x^{2k'+1} y = x^{2k'} xy = (x^2)^{k'} xy = (x^2)^{k'} y' = x'^{k'} y'$, y por tanto vale el invariante al terminar la iteración.

Concluimos la postcondición, que junto con la salida temprana en el caso de $n = 0$ nos deja concluir que el valor de retorno de nuestro programa es efectivamente x_0^n , con lo cual es un algoritmo de exponentiación correcto. \square

Nota

Noten cómo los algoritmos con estado son más engorrosos de demostrar correctos, pues necesitamos más formalidad sobre las transiciones de estado para evitar cometer errores. Un argumento poco riguroso sería decir «En cada iteración, $x^k y = x_0^n$ », pero esto abre la puerta a errores (¿en qué momento de la iteración? ¿por qué es cierto eso?). El teorema del invariante es una herramienta formal para obtener rigor.

4.4.3 Correctitud de algoritmos recursivos

Si nuestro algoritmo es recursivo, en general vamos a usar inducción para probar su correctitud. Veamos una versión recursiva del algoritmo `Exp`.

```

1: procedure Exp( $a \in \mathbb{N}, n \in \mathbb{N}$ )
2:   if  $n = 0$  then
3:     return 1
4:   end
5:    $b \leftarrow \text{Exp}(a, \lfloor \frac{n}{2} \rfloor)$ 
6:    $c \leftarrow b^2$ 
7:   if  $n \bmod 2 = 1$  then
8:      $c \leftarrow c \times a$ 
9:   end

```

```

10:   return c
11: end

```

Para probar su correctitud, vamos a definir una noción de «tamaño» de entrada, y probar la correctitud de nuestro algoritmo para todas las entradas de tamaño menor o igual a n , para todo $n \in \mathbb{N}$.

Proposición 1.4.16

El algoritmo recursivo devuelve a^n .



Demostración. Definamos una propiedad $P(n)$: Para todo $a \in \mathbb{N}$, $\text{Exp}(a, n) = a^n$.

- $P(0)$. Queremos ver que $\text{Exp}(a, 0) = a^0 = 1$ para todo $a \in \mathbb{N}$. Si $n = 0$, entonces entramos en el **if** de la línea 2, y devolvemos 1, que es la respuesta correcta.
- $\forall n \in \mathbb{N}. (n > 0 \wedge (\forall k \in \mathbb{N}. k < n \Rightarrow P(k))) \Rightarrow P(n)$. Vamos a usar inducción global. Si $n > 0$, entonces no entramos en el **if** de la línea 2. Llamamos a $\text{Exp}(a, \lfloor \frac{n}{2} \rfloor)$. Como $\lfloor \frac{n}{2} \rfloor < n$, podemos usar la hipótesis inductiva para ver que $b = a^{\lfloor \frac{n}{2} \rfloor}$. Ahora partimos en casos:
 - ▶ Si $n \equiv 0 \pmod{2}$, entonces $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$. Luego, $a^n = (a^{\frac{n}{2}})^2$. Como asignamos $b \leftarrow a^{\frac{n}{2}}$, y luego $c \leftarrow b^2$, vemos que $c = a^n$. Como $n \equiv 0 \pmod{2}$, no entramos en el **if** de la línea 7. Luego, cuando devolvemos c en la línea 10, estamos devolviendo a^n , que es la respuesta correcta.
 - ▶ Si $n \equiv 1 \pmod{2}$, entonces $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$. Luego, $a^n = a^{2\frac{n-1}{2}+1} = a^{2\frac{n-1}{2}} \times a = a^{(\lfloor \frac{n}{2} \rfloor)^2} \times a$. Como asignamos $b \leftarrow a^{\lfloor \frac{n}{2} \rfloor}$, y luego $c \leftarrow b^2$, vemos que $c = a^{2\frac{n-1}{2}}$. Como $n \equiv 1 \pmod{2}$, entonces entramos al **if** de la línea 7, y multiplicamos a c por a , obteniendo $c = a^{2\frac{n-1}{2}} \times a = a^{2\frac{n-1}{2}+1} = a^{n-1+1} = a^n$. Luego, cuando devolvemos c en la línea 10, estamos devolviendo a^n , que es la respuesta correcta.



Vemos como es más fácil demostrar esto que un algoritmo iterativo, que cambia estados. Esto es cierto en general, y es parte del motivo por el cual la gente usa algoritmos y lenguajes de programación funcionales.

4.4.4 Definiciones equivalentes

Si tenemos definiciones equivalentes para nuestro objeto, podemos hacer uso de cualquiera de ellas. Por ejemplo:

Ejercicio 1.4.17

Recordemos que una función $f : \mathbb{R} \rightarrow \mathbb{R}$ es continua en un punto $x_0 \in \mathbb{R}$ si cumple cualquiera de las siguientes definiciones equivalentes:

- Para todo $\varepsilon > 0$, existe $\delta > 0$ tal que para todo $x \in \mathbb{R}$, si $|x - x_0| < \delta$, entonces $|f(x) - f(x_0)| < \varepsilon$.
- Para toda sucesión $(x_n)_{\{n \in \mathbb{N}\}}$ en \mathbb{R} que converge a x_0 , la sucesión $(f(x_n))_{\{n \in \mathbb{N}\}}$ converge a $f(x_0)$.

Sea f la función definida por:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$
$$f(x) = \begin{cases} 1 & \text{si } x \in \mathbb{Q} \\ 0 & \text{si } x \notin \mathbb{Q} \end{cases}$$

Demostrar que f no es continua en ningún punto.



Demostración. La primer noción de continuidad parece difícil de usar acá, pues para cualquier $\delta > 0$, podemos encontrar un número racional y un número irracional dentro del intervalo $(x_0 - \delta, x_0 + \delta)$, y luego no podemos encontrar un único δ que funcione para todos los x en ese intervalo. La segunda definición parece más manejable.

Vamos a encontrar dos secuencias que convergen a x_0 , una (q_n) de números racionales y otra (i_n) de números irracionales. Por ejemplo, podemos tomar la sucesión $q_n = \frac{\lfloor 10^n x_0 \rfloor}{10^n}$, que es truncar x_0 a n decimales, y la sucesión $i_n = q_n + \frac{\sqrt{2}}{10^n}$. Como q_n es racional para todo $n \in \mathbb{N}$, i_n es irracional, y la secuencia (i_n) también converge a x_0 . Entonces, tenemos que $f(q_n) = 1$ para todo $n \in \mathbb{N}$, y $f(i_n) = 0$ para todo $n \in \mathbb{N}$.

Si asumimos que f es continua en x_0 , ambas sucesiones $(f(q_n))$ y $(f(i_n))$ convergen a $f(x_0)$. Sin embargo, la primera sucesión es constantemente 1, y luego converge a 1, y la segunda sucesión es constantemente 0, y luego converge a 0. Luego, $f(x_0) = 1$ y $f(x_0) = 0$, lo cual es una contradicción. Luego, f no es continua en ningún punto $x_0 \in \mathbb{R}$. \square

Si tenemos que probar condiciones equivalentes, frecuentemente nos va a ser útil ordenarlas de manera que cada una implique la próxima, y la última implique la primera, estableciendo así la equivalencia entre todas, usando sólo implicaciones.

Ejercicio 1.4.18

Sea X un conjunto, y \sim una relación de equivalencia en X . Sean $a, b \in X$. Entonces son equivalentes:

1. $a \sim b$
2. $[a] \cap [b] \neq \emptyset$
3. $[a] = [b]$



Demostración.

- $1 \Rightarrow 2$: Como $a \sim b$, entonces b es un elemento de $[a]$. También b es un elemento de $[b]$. Entonces $b \in [a] \cap [b]$, y luego $[a] \cap [b] \neq \emptyset$.
- $2 \Rightarrow 3$: Sea $y \in [a] \cap [b]$. Luego $y \sim a$ como también $y \sim b$. Entonces para todo $x \in [a]$, tenemos por transitividad que $x \sim y \sim b$, y luego $x \in [b]$, y luego $[a] \subseteq [b]$. De la misma manera, si $x \in [b]$, tenemos que $x \sim b \sim y \sim a$, y luego $x \in [a]$, y $[b] \subseteq [a]$. Entonces $[a] = [b]$.
- $3 \Rightarrow 1$: Como $[a] = [b]$ y $a \in [a]$, entonces $a \in [b]$, y luego $a \sim b$.

□

4.4.5 Contrarecíproco y contradicción

Si tenemos que probar una implicación, es decir una proposición de la forma $P \Rightarrow Q$, podemos probar algo equivalente, que es el contrarecíproco de esa implicación. El contrarecíproco de $P \Rightarrow Q$ es $(\neg Q) \Rightarrow (\neg P)$. Hacemos esto cuando nos es más cómodo tener de antecedente $\neg Q$, por ejemplo porque ya probamos $\neg Q$ y queremos usar esto en un *modus ponens* para probar $\neg P$.

Esto también se puede usar para probar proposiciones en general, aún cuando no sean inmediatamente implicaciones. Si tenemos la proposición P , esta es equivalente a la proposición $\top \Rightarrow P$. Luego, es equivalente a su contrarecíproco, que es $(\neg P) \Rightarrow \perp$. Esto se llama probar P por contradicción, o a veces, «por absurdo».

Aún siendo equivalentes, a veces una va a ser más fácil de probar que la otra. Por ejemplo, si P es de la forma $P : \neg(\exists y.Q(y))$, parece difícil probarla, porque es difícil probar la no-existencia de algo, uno parece carecer de herramientas. Al negarla obtenemos $\exists y.Q(y)$. Eso nos da otro objeto (y), otro sustantivo del cual hablar, y con el cual razonar. Cambiamos nuestra misión a ver si podemos obtener algo internamente contradictorio sobre y , y queremos llegar a \perp . Lo mismo sucede si tenemos una proposición de la forma $P : \forall x.Q(x)$. Probar esto por contradicción es probar $(\neg(\forall x.Q(x))) \Rightarrow \perp$, que es lo mismo que decir $(\exists x.\neg Q(x)) \Rightarrow \perp$. Eso otra vez nos da un objeto, x , del cual hablar.

Por ejemplo, el siguiente teorema era probablemente sabido por Aristóteles, y aparece en «Elementos» de Euclides:

Teorema 7

$\sqrt{2} \notin \mathbb{Q}$.

♥

Demostración. Asumamos $\sqrt{2} \in \mathbb{Q}$. Luego, existen $a, b \in \mathbb{Z}$ tal que $\sqrt{2} = \frac{a}{b}$, con $b \neq 0$. Podemos elegir a a, b coprimos, dado que si no lo fueran, tomamos $\frac{a}{b} = \frac{\frac{a}{d}}{\frac{b}{d}}$, con $d = \gcd(a, b)$, y ahora teniendo el numerador y denominador coprimos.

Como $\sqrt{2} = \frac{a}{b}$, tenemos que $\frac{a^2}{b^2} = 2$, y luego $a^2 = 2b^2$. Luego a es par, pues a^2 lo es, y el cuadrado de un número impar sería impar.

Entonces existe $k \in \mathbb{Z}$, tal que $a = 2k$. Usando esto obtenemos que $2b^2 = (2k)^2 = 4k^2$, y luego $b^2 = 2k^2$. Luego, b también es par, porque su cuadrado es par.

Luego ambos a y b son pares. Esto no puede suceder, porque dijimos que eran coprimos.
 Luego, lo que asumimos es falso, y no es cierto que $\sqrt{2} \in \mathbb{Q}$. Esto prueba que $\sqrt{2} \notin \mathbb{Q}$, que es lo que queríamos demostrar. \square

El motivo por el que la demostración es por contradicción es porque el que x no esté en \mathbb{Q} nos dice poco, sólo que $x \in \mathbb{R} \setminus \mathbb{Q}$, pero no es cómodo hablar sobre los irracionales. El demostrar por contradicción nos deja «imaginar» que $x \in \mathbb{Q}$, y el objetivo es llegar a \perp , algo falso. Usando esta hipótesis concluimos cosas útiles sobre x , como que $x = \frac{n}{m}$ con $n, m \in \mathbb{Z}, m \neq 0$.

Advertencia

Frecuentemente vemos alumnos empezar usando esta estrategia automáticamente, sin pensar en por qué se lo está haciendo. Imaginamos que es porque les da algo que escribir, y «se parece a progreso». Pero realmente no sugerimos hacer esto sin tener una razón específica, muchas veces se confunden con la cantidad (y paridad) de negaciones, ya que cada vez que hacen esto agregan una negación más. Eventualmente cometan algún error, llegan a un absurdo, y dicen «¡Listo, terminé!», pero el absurdo vino de hacer otra cosa mal en el medio.

Las demostraciones por contradicción son una fuente clásica de errores de alumnos.[4]

Si van a usar esta estrategia, recuerden las reglas de negación.

$$\begin{aligned}\neg(\forall x \in X.P(x)) &\Leftrightarrow \exists x \in X.\neg(P(x)) \\ \neg(\exists x \in X.P(x)) &\Leftrightarrow \forall x \in X.\neg(P(x)) \\ \neg(P \Rightarrow Q) &\Leftrightarrow P \wedge (\neg Q) \\ \neg(P \vee Q) &\Leftrightarrow (\neg P) \wedge (\neg Q) \\ \neg(P \wedge Q) &\Leftrightarrow (\neg P) \vee (\neg Q) \\ \neg\perp &\Leftrightarrow \top \\ \neg\top &\Leftrightarrow \perp\end{aligned}$$

Vemos frecuentemente el error $\neg(P \Rightarrow Q) \Leftrightarrow \neg P \Rightarrow \neg Q$, y $\neg(P \Rightarrow Q) \Leftrightarrow Q \Rightarrow P$. A continuación hay varios ejemplos de usos de negación y contradicción, sólo algunos son correctos.

<p>Si la inflamación no se va, el dolor vuelve. Luego, voy a tomar Anaflex, porque saca la inflamación.</p>	<p>$P =$ La inflamación se va, $Q =$ El dolor vuelve, $R =$ Tomo Anaflex. Asumimos que $(\neg P) \Rightarrow Q$, y que $R \Rightarrow P$. No podemos concluir que $R \Rightarrow \neg Q$. Perfectamente puede ser que $Q = \top$, y el dolor siempre vuelve. Tomar Anaflex no hace nada para el dolor. La «demostración» asume que $(\neg P \Rightarrow Q) \Leftrightarrow (P \Rightarrow \neg Q)$, que es mentira¹¹.</p>
<p>Sea A un conjunto que contiene a todos los conjuntos. Sea $B = \{x \in A \mid x \notin x\}$. Si $B \in B$, entonces por definición de B, $B \notin B$, que no puede suceder. Si no, $B \notin B$, y por definición</p>	<p>Esto es correcto. En ambas ramas, llegamos a una contradicción. Si $P \Rightarrow Q$ y $\neg(P) \Rightarrow Q$, entonces vale Q. En este caso, $Q = \perp$, $P = B \in$</p>

¹¹El autor de este documento ha odiado esa publicidad más de 20 años, precisamente por ser un mal uso de operaciones lógicas.

<p>de B, $B \in B$, que no puede suceder. Luego, A no puede existir.</p>	<p>B. Luego, si asumimos que A existe, probamos \perp, y por ende concluimos que A no existe.¹²</p>
<p>Queremos ver si vale la siguiente proposición: «Sea $k \geq 1$, con $k \in \mathbb{Z}$. Si k es tal que $2^k \equiv 0 \pmod{3}$, entonces $8 \equiv 1 \pmod{3}$». Vemos que hay un contraejemplo, con $k = 1$, tenemos $8^1 = 8 \not\equiv 3 \pmod{2}$, pues $8 = 2 \times 3 + 2 \equiv 2 \pmod{3}$.</p>	<p>Esto está mal. Tenemos una proposición $P(k)$: $Q(k) \Rightarrow R(k)$ sobre todos los $k \in \mathbb{Z}, k \geq 1$, con $Q(k) : 2^k \equiv 0 \pmod{3}$, y $R(k) : 8 \equiv 1 \pmod{3}$. Lo que encontramos es un contraejemplo a $R(k)$, pero esto no implica que $P(k)$ sea falsa. De hecho $P(k)$ siempre es cierta, pues $Q(k)$ es falsa para todo tal k. $P(k)$ es equivalente a $\neg Q(k) \vee R(k)$, y como $Q(k)$ es siempre \perp, entonces $P(k)$ es equivalente a $\top \vee R(k)$, que es equivalente a \top. Luego, $P(k)$ siempre es cierta para tales k.</p>
<p>Queremos ver que si $a^2 = 0$, con $a \in \mathbb{R}$, entonces $a = 0$. Supongamos que $a \neq 0$. Entonces existe $a^{-1} \in \mathbb{R}$. Luego, $a^2 = 0 \Rightarrow a^{-1}a^2 = 0 \Rightarrow a = 0$, con lo cual concluimos que $a = 0$, una contradicción pues asumimos que $a \neq 0$. Luego, lo que asumimos no puede suceder, y tenemos que $a = 0$.</p>	<p>Está bien. Asumimos $\neg P$, y llegamos a una contradicción. En particular, llegamos a P. Luego, no puede suceder que valga $\neg P$, y efectivamente tenemos que vale P sin asumir nada.</p>
<p>Sea $n \in \mathbb{N}$, y $f : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$. Queremos probar que si f es inyectiva, entonces es sobreyectiva. Asumimos, entonces, que f no es inyectiva. Luego existen $1 \leq a, b \leq n$ tales que $f(a) = f(b)$. Como a algún elemento del codominio le llegan dos flechas, hay algún elemento del codominio al que no le llega ninguna. Luego f no es sobreyectiva.</p>	<p>Esto confunde $A \Rightarrow B$ con $\neg A \Rightarrow \neg B$. Lo que el autor quiso probar, quizás, es $\neg B \Rightarrow \neg A$, que sí es equivalente a $A \Rightarrow B$. Se confundió, quizás, por no ser suficientemente formal.</p>
<p>Queremos ver que existe un irracional $x \in \mathbb{R}$ tal que x^x es racional. Sea x una solución real a $x^x = 2$, que existe en $(1, 2)$ por el teorema del valor medio, pues $1^1 = 1$ y $2^2 = 4$, con x^x continua. Supongamos que $x = \frac{p}{q}$, con p y q enteros positivos coprimos. Como $x > 1$, entonces $p > q$, y luego $p - q > 0$. Tomando q-ésimas potencias a ambos lados de $x^x = 2$, obtenemos $\left(\frac{p}{q}\right)^p = 2^q$, y luego $p^p = q^p 2^q$. Luego p es par, $p = 2k$ para algún $k \in \mathbb{N}$. Tenemos $(2k)^{2k} = q^p 2^q \Rightarrow 2^{2k} k^{2k} = q^p 2^q \Rightarrow 2^{2k-q} k^{2k} = q^p \Rightarrow 2^{p-q} k^{2k} = q^p$. Como una potencia de q es par, q es par, lo cual contradice que p y q eran coprimos. Luego no pueden existir p, q, y x es irracional.</p>	<p>Está bien. Asumimos $P : \exists p, q \in \mathbb{N}. \gcd(p, q) = 1 \wedge p, q > 0 \wedge x = \frac{p}{q}$, que es equivalente a que x es un racional positivo. Llegamos a un absurdo, pues $2 \mid \gcd(p, q)$, y $2 \nmid 1$. Luego no puede ser que valga P, o equivalentemente, no puede ser que $x \in \mathbb{Q}$. Como $x \in \mathbb{R}$, entonces $x \in \mathbb{R} \setminus \mathbb{Q} = \mathbb{I}$.</p>
<p>Queremos probar que si x e y son racionales, entonces $x + y$ es racional. Escribimos $x =$</p>	<p>Esta demostración no está «mal», pero no usa la contradicción en ningún momento. Es de la</p>

¹²Esta es la paradoja de Russell[5].

<p>$\frac{a}{b}, y = \frac{c}{d}$, con $b \neq 0, c \neq 0$, y $a, b, c, d \in \mathbb{Z}$. Asumimos que $x + y \notin \mathbb{Q}$. Luego vemos que $x + y = \frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$. Como $b \neq 0$, y $d \neq 0$, entonces $bd \neq 0$. Como $a, b, c, d \in \mathbb{Z}$, entonces $ad + bc \in \mathbb{Z}$, y $bd \in \mathbb{Z}$, $bd \neq 0$. Luego $x + y \in \mathbb{Q}$, que contradice que $x + y$ era irracional. Luego lo que asumimos no puede suceder, y concluimos que $x + y \in \mathbb{Q}$.</p>	<p>forma «Asumo $\neg P$. Pruebo P sin usar $\neg P$. Esto contradice que $\neg P$, luego vale P.» Pero P vale porque probamos P, no por la contradicción con $\neg P$. La contradicción es enteramente superflua, y sólo hace más difícil leer la demostración, ambos para el que la corrige, y para el alumno que intenta ver si cometió un error.</p> <p>Esto pasa cuando los alumnos mecánicamente intentan usar contradicción, sin pensar por qué lo está haciendo.</p>
<p>Queremos ver que existen dos irracionales $x, y \in \mathbb{R} \setminus \mathbb{Q}$, tales que $x^y \in \mathbb{Q}$. Sea $A = \sqrt{2}^{\sqrt{2}}$. Si A es racional, terminamos, pues $x = \sqrt{2}, y = \sqrt{2}$ resuelve lo pedido. Si no, A es irracional. Pero luego, $A^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2 \in \mathbb{Q}$, y luego $x = A, y = \sqrt{2}$ resuelve lo pedido.</p>	<p>Está bien. Si $P : A \in \mathbb{Q}$, y $Q : \exists x, y \in \mathbb{I}. x^y \in \mathbb{Q}$, entonces probamos que $P \Rightarrow Q$, y que $(\neg P) \Rightarrow Q$. Juntando ambas oraciones, tenemos que $(P \vee \neg P) \Rightarrow Q$, que es equivalente a $\top \Rightarrow Q$, que es equivalente a Q. Luego, probamos Q. Notar que esta demostración no prueba que A es racional.¹³</p>
<p>Queremos probar que para todo $n \in \mathbb{N}$, si $2n + 1 \equiv 0 \pmod{3}$, entonces $n^2 + 1 \equiv 0 \pmod{3}$. Por contrarecíproco, asumimos que $2n + 1 \not\equiv 0 \pmod{3}$. Entonces partimos en casos:</p> <ul style="list-style-type: none"> Si $n = 3k + 1$ con $k \in \mathbb{N}$, entonces $n^2 + 1 = 9k^2 + 6k + 2 \equiv 2 \not\equiv 0 \pmod{3}$. Si $n = 3k + 2$ con $k \in \mathbb{N}$, entonces $n^2 + 1 = 9k^2 + 12k + 5 \equiv 2 \not\equiv 0 \pmod{3}$. <p>En ambos casos, $n^2 + 1 \not\equiv 0 \pmod{3}$. Por contrarecíproco, si $2n + 1 \equiv 0 \pmod{3}$, entonces $n^2 + 1 \equiv 0 \pmod{3}$, que es lo que queríamos demostrar.</p>	<p>Esto está mal, confunde $A(n) \Rightarrow B(n)$ con $\neg A(n) \Rightarrow \neg B(n)$, donde $A(n) : 2n + 1 \equiv 0 \pmod{3}$, y $B(n) : n^2 + 1 \equiv 0 \pmod{3}$. Lo que queríamos probar es $A(n) \Rightarrow B(n)$, pero lo que esto prueba es $\neg A(n) \Rightarrow \neg B(n)$.</p>
<p>Sean u, v, w vectores linealmente independientes en un espacio vectorial real V, y $T : V \rightarrow W$ una transformación lineal inyectiva. Queremos probar que $\{T(u), T(v), T(w)\}$ es linealmente independiente. Asumamos que no. Luego, como $\{T(u), T(v), T(w)\}$ es linealmente dependiente, existen $\alpha, \beta, \gamma \in \mathbb{R}$, no todas cero, tal que $\alpha T(u) + \beta T(v) + \gamma T(w) = 0$. Asumamos sin pérdida de generalidad que $\alpha \neq 0$. Luego $T(u) = -\frac{\beta}{\alpha}T(v) - \frac{\gamma}{\alpha}T(w)$. Esto es lo mismo que decir que $T(u) = T(-\beta\frac{v}{\alpha} - \gamma\frac{w}{\alpha})$. Pero T es inyectiva, luego $u = -\beta\frac{v}{\alpha} - \gamma\frac{w}{\alpha}$. Esto no puede suceder, pues u sería una combinación</p>	<p>Está bien. Notar cómo asumimos $\alpha \neq 0$ sin pérdida de generalidad. Esto significa que como alguno de los tres coeficientes no es cero, podemos renombrar las variables para que se coeficiente sea α. Nada en la demostración depende de cuál exactamente es u, v, o w, o α, β, o γ.</p> <p>Asumimos que vale $\neg(\{T(u), T(v), T(w)\}$ es linealmente independiente), y llegamos a una contradicción. Esto nos dice que $\{T(u), T(v), T(w)\}$ es linealmente independiente, que es lo que queríamos probar.</p>

¹³Uno puede usar el teorema de Gelfond-Schneider[6] para probar que A no sólo es irracional, sino que es transcendental.

lineal de v y w , y sabemos que $\{u, v, w\}$ son linealmente independientes.	
Queremos ver que para todo $x \in \mathbb{R}$, $x^2 \geq 0$. Asumimos, por contradicción, que para todo $x \in \mathbb{R}$, $x^2 < 0$. Tomemos $x = 3$, y vemos que $3^2 = 9 \geq 0$. Esto contradice lo que asumimos, y por lo tanto $x^2 \geq 0$ para todo $x \in \mathbb{R}$.	Esto está mal. Intenta negar $\forall x \in \mathbb{R}. x^2 \geq 0$, y dice que eso es $\forall x \in \mathbb{R}. x^2 < 0$. La negación correcta es $\exists x \in \mathbb{R}. x^2 < 0$. La demostración no prueba lo pedido.
<p>Queremos ver que no existe un programa H que, dado cualquier programa P, devuelve TRUE si y sólo si $P()$ se detiene, y FALSE si no. Asumamos que H existe. Sea A el siguiente program:</p> <pre> 1: procedure A() 2: if H(A) then 3: while TRUE do 4: end 5: end 6: end </pre> <p>Consideremos $H(A)$. Si $H(A) = \text{True}$, entonces $A()$ debe detenerse, con lo cual nunca entramos al ciclo infinito, pero entonces $\neg H(A)$, que no puede suceder pues asumimos $H(A)$. Por otro lado, si $H(A) = \text{False}$, entonces A tuvo que entrar al ciclo, y luego $H(A)$, que no puede suceder pues asumimos $\neg H(A)$.</p> <p>Luego H no puede existir.</p>	Está bien. Partimos en casos, dependiendo del valor de $H(A)$. En ambas ramas, llegamos a una contradicción asumiendo la rama. Luego lo que asumimos inicialmente es falso, es decir, H no puede existir. Este es un caso particular del «halting theorem»[7].

4.4.6 Si y sólo si

Si tenemos que probar un si-y-sólo-si (\Leftrightarrow), podemos probar por separado \Rightarrow y \Leftarrow . Es muy común que uno de los dos sea muy fácil, y el otro sea más difícil. Por ejemplo:

Teorema 8 (Cantor-Schröder-Bernstein)

Sean A y B dos conjuntos. Entonces son equivalentes:

- Existe una función biyectiva $h : A \rightarrow B$.
- Existe una función inyectiva $f : A \rightarrow B$, y una función inyectiva $g : B \rightarrow A$.

La vuelta de este teorema es extremadamente molesta de probar, mientras que la ida es trivial.

Demostración.

- \Rightarrow) Asumamos que existe una función biyectiva $h : A \rightarrow B$. Entonces, h es inyectiva, y su inversa $h^{-1} : B \rightarrow A$ también es inyectiva. Luego, existen funciones inyectivas $f = h : A \rightarrow B$ y $g = h^{-1} : B \rightarrow A$.

- \Leftarrow) Esta demostración es de Dedekind[8]. Sea $C_0 = A \setminus g(B)$, y para $n \geq 0$, definimos recursivamente $C_{n+1} = g(f(C_n))$. Definimos $C = \bigcup_{n \in \mathbb{N}} C_n$. Definimos la función $h : A \rightarrow B$ como:

$$h(x) = \begin{cases} f(x) & \text{si } x \in C \\ g^{-1}(x) & \text{si } x \notin C \end{cases}$$

Para ver que h está bien definida, sea $x \in A$. Si $x \notin C$, entonces $x \notin C_0$, con lo cual $x \in g(B)$, y luego existe $g^{-1}(x) \in B$. Luego, $h(x)$ está bien definida.

Ahora veamos que $g^{-1}(A \setminus C)$ siempre cae en $B \setminus f(C)$. Asumamos lo contrario, y supongamos que existe un $x \in A \setminus C$ tal que $g^{-1}(x) = f(z)$ para algún $z \in C$. Llamemos $y = f(z)$. Como $C = \bigcup_{n \in \mathbb{N}} C_n$, existe n tal que $z \in C_n$. Como $y = f(z)$ y $x = g(y)$ (pues $y = g^{-1}(x)$), y entonces $x = g(f(z)) \in C_{n+1} \subseteq C$, lo cual contradice que $x \in A \setminus C$. Luego, $g^{-1}(A \setminus C) \subseteq B \setminus f(C)$.

Con esto, veamos que h es inyectiva. Sabemos que f y g^{-1} son inyectivas. Luego, si h fuera no-inyectiva, tendríamos que tener $h(x) = h(y)$ con $x \in C$ e $y \notin C$, es decir, $f(x) = g^{-1}(y)$. Sin embargo, f manda elementos de C a elementos de $f(C)$, mientras que, como vimos, g^{-1} manda elementos de $A \setminus C$ a elementos de $B \setminus f(C)$. Luego las imágenes de f y g^{-1} no intersecan, y nunca podemos tener $f(x) = g^{-1}(y)$. Luego, h es inyectiva.

Ahora vamos a mostrar que si $y \in B$ es tal que $g(y) \in C$, entonces $y \in f(C)$. Como $g(y) \in C = \bigcup_{n \in \mathbb{N}} C_n$, existe n tal que $g(y) \in C_n$. Si $n = 0$, entonces $g(y) \in C_0 = A \setminus g(B)$, lo cual no puede suceder. Luego, $n \geq 1$, y entonces $g(y) \in C_n = g(f(C_{n-1}))$. Como g es inyectiva, tenemos que $y \in f(C_{n-1}) \subseteq f(C)$. Luego, si $g(y) \in C$, entonces $y \in f(C)$.

Para ver que h es sobreyectiva, consideremos cualquier $y \in B$. Queremos decir que $y \in h(A)$. y puede estar en $f(C)$, o no. Si $y \in f(C)$, entonces existe un $x \in C \subseteq A$ tal que $f(x) = y$, y por lo tanto $h(x) = y$. Si $y \notin f(C)$, por el párrafo anterior sabemos que $g(y) \in A \setminus C$. Luego, $h(g(y)) = g^{-1}(g(y)) = y$. En ambos casos, existe un $x \in A$ tal que $h(x) = y$. Luego, h es sobreyectiva.

□

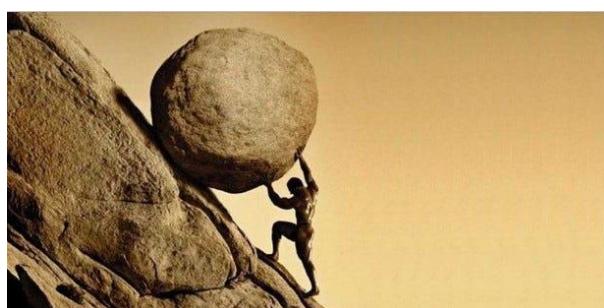


Figura 1: Sísifo probando la vuelta de Cantor-Schröder-Bernstein.

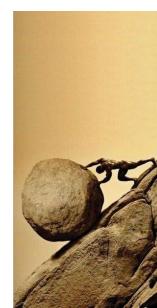


Figura 2: Sísifo probando la ida de Cantor-Schröder-Bernstein.

⚠ Advertencia

Uno puede probar un si-y-sólo-si mediante una cadena de \Leftrightarrow , pero tiene que tener cuidado que absolutamente todos los pasos que uno haga a las proposiciones que está manejando, sean

equivalencias, y no sólo implicaciones. Por esto frecuentemente es más fácil hacer cada implicación por separado, y luego si uno se da cuenta que se puede hacer ambas al mismo tiempo, reescribirlo de esa forma.

Por ejemplo, la siguiente demostración es incorrecta porque usa una implicación pero dice usar un si-y-sólo-si.

Proposición 1.4.19

$$-2 = 2.$$



Demostración. Sea $x = -2$. Entonces:

$$\begin{aligned}x &= -2 \Leftrightarrow \\x^2 &= 4 \Leftrightarrow \\\sqrt{x^2} &= \sqrt{4} \Leftrightarrow \\x &= 2\end{aligned}$$



El error está en que $x = 2$ implica $\sqrt{x^2} = \sqrt{4}$, pero la vuelta no vale. Para probar una cadena de si-y-sólo-si, absolutamente todos los pasos deben ser si-y-sólo-si. Con que haya una implicación sin vuelta cierta, todo está mal.

Sucede lo mismo dando vuelta el argumento:

Demostración. Sea $x = 2$. Entonces:

$$\begin{aligned}x &= 2 \Leftrightarrow \\x^2 &= \sqrt{4} \Leftrightarrow \\x^2 &= 4\end{aligned}$$

Por lo tanto cualquier solución a $x^2 = 4$ es solución de nuestra ecuación, y en particular $-2 = 2$.



4.4.7 Partir en casos

Si tenemos que probar $\forall x \in X. P(x)$ y podemos dividir el dominio X de forma productiva, donde cada subconjunto de X tiene una demostración de P simple pero mayormente independiente, podemos partir en casos. Por ejemplo:

Ejercicio 1.4.20

Sea $n \in \mathbb{Z}$. Entonces $n(n + 1)$ es par.



Demostración. Partimos en casos.

- Si n es par, entonces existe $k \in \mathbb{Z}$ tal que $n = 2k$. Luego, $n(n+1) = 2k(2k+1)$. Llamando $m = k(2k+1)$, vemos que $n(n+1) = 2m$, con lo cual $n(n+1)$ es par.
- Si n es impar, entonces existe $k \in \mathbb{Z}$ tal que $n = 2k+1$. Luego, $n(n+1) = (2k+1)(2k+1+1) = (2k+1)(2k+2) = (2k+1)2(k+1)$. Llamando $m = (k+1)(2k+1)$, tenemos que $n(n+1) = 2m$, con lo cual $n(n+1)$ es par.

□

Cuando hacemos esto, es importante tener en cuenta cuán difícil es la demostración en cada caso. Si partimos en, por ejemplo, $n = 0$ y $n \neq 0$, pero uno de los casos es mucho más difícil que el otro, entonces el caso que es más difícil requiere mucho más esfuerzo y detalle. A veces vemos alumnos que hacen el caso simple con detalle, y el caso complejo lo dejan vago, lo cual es bastante inútil.

4.4.8 Unicidad

Si tenemos que probar que «existe un único x en X tal que $P(x)$ », una estrategia común es tomar dos objetos que cumplen $P(x)$, y concluir que son el mismo.

Ejercicio 1.4.21

Probar que existe una única matriz I en $M_n(\mathbb{Z})$, el conjunto de matrices $n \times n$ con entradas con coeficientes enteros, tal que para toda matriz $A \in M_n(\mathbb{Z})$, vale $IA = AI = A$.

♠

Demostración. Sean I, J dos matrices en $M_n(\mathbb{Z})$ tales que para toda matriz $A \in M_n(\mathbb{Z})$, vale $IA = AI = A$ y $JA = AJ = A$. Queremos ver que $I = J$. Consideremos la matriz $A = J$. Entonces, por definición de I , tenemos $JI = J$. Por otro lado, considerando $A = I$, por definición de J tenemos que $JI = I$. Luego, $I = JI = J$. □

A veces vamos a usar el contrarecíproco para probar unicidad, postulando que existen dos objetos distintos que cumplen una propiedad, y llegando a un absurdo. Les reitero que no usan el contrarecíproco mecánicamente, pero sí que lo conozcan como herramienta.

4.5 Pasar en limpio

Gran parte de una demostración es jugar con los objetos, e intentar ver qué sucede. Eventualmente, uno llega a un argumento formal sólido. Sin embargo, al comunicarle este argumento a alguien, no hace falta comunicar todas las cosas que pensamos, las ecuaciones que no llevaron a nada, los errores que cometimos, los ejemplos que intentamos, los dibujos que nos confundieron, etcétera.

Es difícil comunicar el patrón incoherente de ideas que pasan por la cabeza mientras uno juega, pero el siguiente es un intento de mostrarlo, y luego la demostración final que uno pasa en limpio, obviando todos los caminos sin salida. No se supone que el texto a continuación sea totalmente comprensible, es sólo un camino vueltero que pueden transitar al jugar.

Ejercicio 1.4.22

Demostrar que para todo $n \in \mathbb{N}$, $n > 0$, todo tablero de $2^n \times 2^n$ casillas con una casilla removida, puede ser cubierto completamente con triominos (pieza que cubre tres casillas, en forma de L).

♠

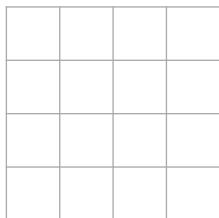
Hrm, dibujemos algunos ejemplos. Con $n = 1$, hay realmente sólo una grilla, salvo rotación:



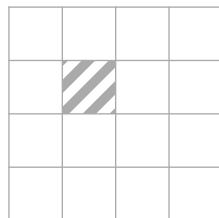
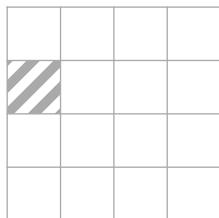
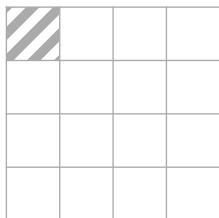
Y no hay más de una forma de poner un triomino:



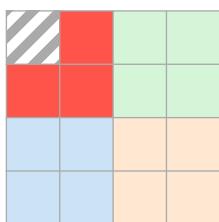
Con $n = 2$, la grilla es:



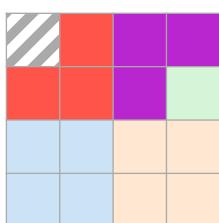
Y realmente hay solo tres formas de poner un cuadrado faltante, salvo rotaciones y reflecciones:



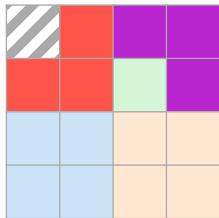
Parece enorme el espacio de cosas para hacer... no parece haber mucha estructura si empiezo a poner triominos al azar. El que me digan $2^n \times 2^n$ me hace pensar en dividir y conquistar, quizás dividiendo el tablero en cuatro tableros $2^{n-1} \times 2^{n-1}$... pero no parece fácil, porque los tableros chicos no tienen una casilla removida cada uno... por ejemplo, para la primera:



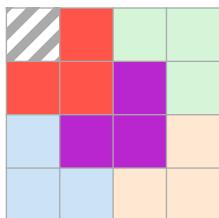
Cubrí el primero perfectamente, pero arruiné los otros tres, porque ya no se parecen al caso $n = 1$, tienen cero bloques faltantes, no uno. Y si sigo poniendo triominos parece que me voy a quedar corto...



Voy a estar en problemas cuando intente cubrir la casilla verde... a ver de otra manera:

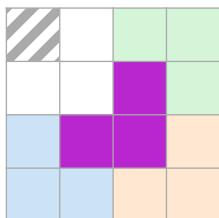


Mismo problema... no puedo entonces cubrir la casilla verde poniendo un triomino enteramente en ella. Hrm... pero podría poner otro triomino en el centro, que cubra una casilla de cada uno de los otros tres tableros... así:

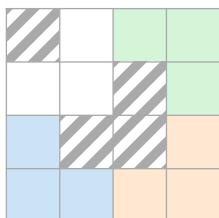


Hrm eso parece funcionar... pero no fue algo divide-and-conquer, sólo tuve suerte... cómo puedo hacer esto mediante divide-and-conquer?

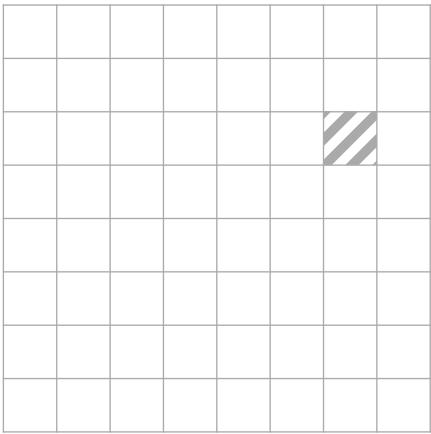
Hrm y si lo hago en el otro orden? Puedo poner el triomino violeta primero:



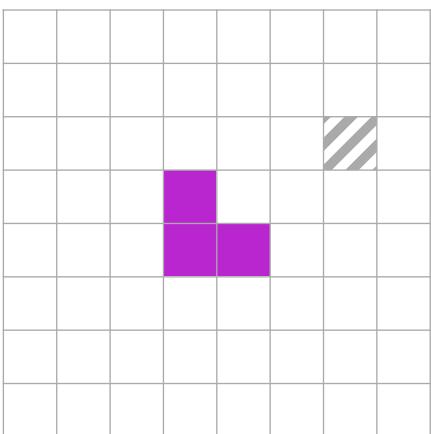
Y ahora me quedaron cuatro sub-grillas, donde cada una tiene una casilla removida! Este tablero es equivalente a tener:



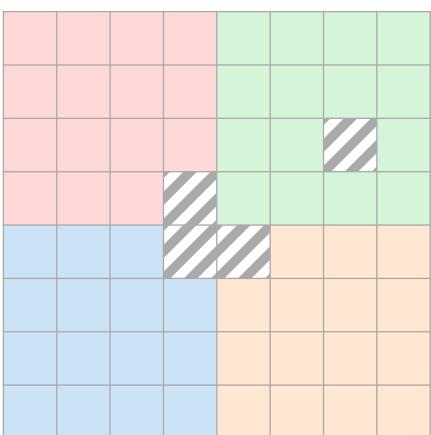
Y esto sí tiene estructura recursiva, porque en cada sub-grilla de 2×2 tengo el mismo problema anterior. A ver cómo se ve esto en tableros más grandes, con $n = 3$...



Puedo poner un triomino en el centro, cubriendo una casilla de cada uno de los sub-tableros, excepto en el que se encuentra la ya-faltante:



Y ahora tenemos cuatro sub-tableros, cada uno con una casilla removida:



Y podemos aplicar divide-and-conquer. Bueno, ahora a formalizar...

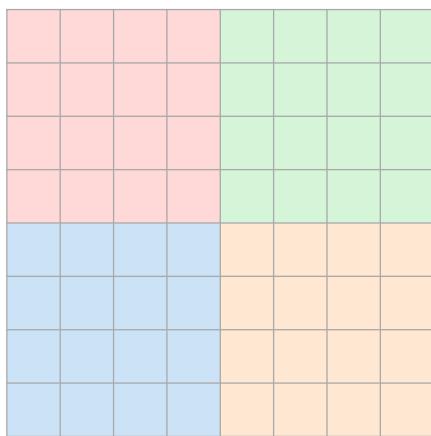
Demostración. Sea $n \in \mathbb{N}$, con $n > 0$. Vamos a demostrar por inducción en n que todo tablero de $2^n \times 2^n$ casillas con una casilla removida, puede ser cubierto completamente con triominos.

Si $n = 1$, el tablero tiene $2 \times 2 = 4$ casillas, y al remover una casilla quedan exactamente tres casillas, que pueden ser cubiertas con un triomino. Los cuatro casos posibles son:

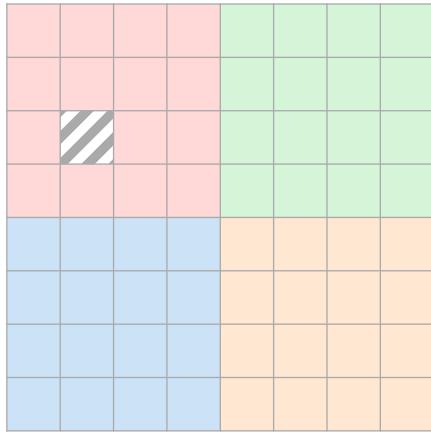


Esto prueba el caso base.

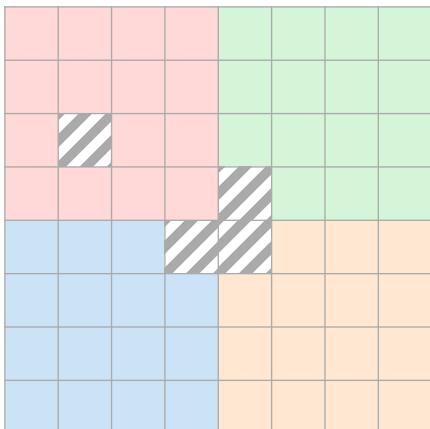
Supongamos que la afirmación es cierta para todo $k < n$, y probémosla para n . Sea T un tablero de $2^n \times 2^n$ casillas con una casilla removida. Dividimos T en cuatro sub-tableros T_1, T_2, T_3, T_4 , cada uno de tamaño $2^{n-1} \times 2^{n-1}$, ubicados en las cuatro esquinas de T . Como ejemplo, para $n = 3$ se ve así:



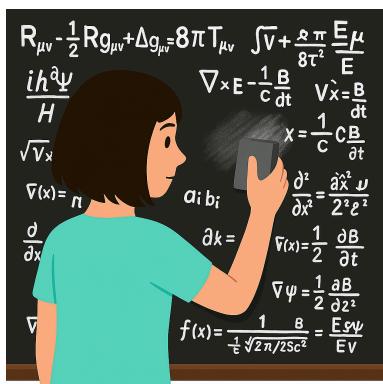
Sin pérdida de generalidad, supongamos que la casilla removida está en T_1 , aunque no sabemos dónde en T_1 . A modo de ilustración, podría ser esta casilla:



Podemos colocar un triomino en el centro de T , cubriendo una casilla de cada uno de los otros tres sub-tableros, T_2, T_3, T_4 . De esta forma, cada uno de los sub-tableros T_1, T_2, T_3, T_4 tiene exactamente una casilla removida. En este ejemplo quedaría así:



Usando la hipótesis inductiva $P(n - 1)$, como cada uno de esos cuatro sub-tableros tiene tamaño $2^{n-1} \times 2^{n-1}$, y tiene exactamente una casilla faltante, cada uno de ellos puede ser cubierto completamente con triominos. Juntando estas coberturas disjuntas de las cuatro sub-grillas de T , obtenemos una cobertura de T por triominos, que es lo que buscábamos. \square



Hrm, quedó medio desordenado eso. Mejor lo emprolijo un poco. Puedo ponerle nombre a los tableros, y enunciar bien mi hipótesis inductiva...

Si sólo ven la demostración final, parece compacta, no comete errores, no intenta varias cosas, no nombra cosas que no usa, no deja cosas sin demostrar para después, tiene notación sensible, y hasta tiene estructura, planteando una inducción formal. No piensen que la demostración nació así - como ven, uno pasa por jugar, probar cosas, planear, y emprolijar. No se frustren si sus demostraciones no se ven como esta última, en su primer pasada.

💡 Consejo

Las siguientes son cosas que pueden hacer al pasar en limpio una demostración:

- Introducir notación útil. A veces un argumento complejo puede ser simplificado introduciendo un símbolo y usándolo repetidamente.
- Extraer sub-lemas. Si en el medio de su demostración tienen que demostrar un lema sobre algún objeto, pueden extraerlo como un sub-lemma, que se puede entender por separado. Al extraerlo, tengan cuidado que las variables que mencionan en la demostración del lema, sean parte del enunciado lema, y estén cuantificadas correctamente en el mismo.

Esto a veces acorta nuestras demostraciones mucho, pues podemos reutilizar el mismo lema varias veces con distintos objetos en la misma demostración.

- Evitar usar simbolismo innecesario. Si no están trabajando explícitamente con fórmulas lógicas, usen «para todo» en vez de \forall , «existe» en vez de \exists , «entonces» en vez de \Rightarrow , etcétera. Su lector tiene décadas de experiencia usando el idioma español, sabemos leer oraciones mucho más rápidamente en su idioma que en notación simbólica de lógica formal.
- Usar conectores lógicos y explicaciones entre sus ecuaciones. Una demostración es un argumento, no una serie de ecuaciones sin semántica.
- Si no necesitan usar contrarecíproco o contradicción, intenten estructurar su demostración para argumentar de forma directa. Es muy difícil leer un argumento que contiene contradicciones anidadas.
- Revisar si en algún lugar dijeron que algo es «obvio», si realmente pueden asumir que el lector lo va a considerar obvio. Es tentador decir que algo es «obvio» como táctica de intimidación para que el lector acepte nuestras proposiciones, pero no va a funcionar en una instancia de evaluación, y es de mal gusto al escribir a un par científico.

5 Errores comunes

5.1 Ser informal

Este es **de lejos** el error que más cometan los alumnos. En este momento de su educación, todavía no les recomiendo dejar de lado la formalidad, y hacer argumentos informales. Un argumento informal puede ser riguroso, pero requiere experiencia hacer esto sin cometer errores. Todavía no tienen esa experiencia. Por ende, a la hora de argumentar, sean formales. Algunas recomendaciones sobre formalidad:

1. Ponganle nombre a todo. Si un sustantivo no tiene nombre, no podemos hablar de él claramente. Muchas veces se quedan «sin saber cómo seguir», porque no tienen a mano suficientes sustantivos para ver relaciones entre ellos, o ver qué cosas cumple cada uno.

Si se encuentran usando preposiciones como «el vértice vecino de ...» o «la lista l pero sin el i -ésimo elemento», deténganse y ponganle nombre a esos objetos. Por ejemplo, «sea u el vértice vecino de ...», o «sea $l' = l \cdot \text{remove}(i)$ ». Ahora pueden hablar de u y l' directamente, y ver qué propiedades y relaciones cumplen.

1. No usen el mismo nombre para dos cosas distintas. Si están modificando un objeto, no usen el mismo nombre para el objeto antes y después de modificarlo.

Ejercicio 1.5.1

Sean $a, b \in \mathbb{Z}$, tal que $a \equiv 1 \pmod{3}$ y $b \equiv 2 \pmod{3}$. Probar que $a + b \equiv 0 \pmod{3}$.

Demostración. Como $a \equiv 1 \pmod{3}$, entonces existe un $k \in \mathbb{Z}$ tal que $a = 3k + 1$. Como $b \equiv 2 \pmod{3}$, existe un $k \in \mathbb{Z}$ tal que $b = 3k + 2$. Luego, $a + b = (3k + 1) + (3k + 2) = 6k + 3 = 3(2k + 1)$, y luego $a + b \equiv 0 \pmod{3}$. \square

Esto está mal, porque usa k para dos cosas distintas. En particular, esto asume que $b = 3k + 2 = (3k + 1) + 1 = a + 1$, lo cual no podemos asumir.

2. Si el objeto X depende de un objeto Y , nómbrénlo X_Y o $X(Y)$, para recordar la dependencia. A veces se olvidan, al crear un objeto, de qué depende, y terminan concluyendo algo falso. Un ejemplo tonto:

Demostración. Queremos ver que el conjunto de los números reales está acotado. Sea $x \in \mathbb{R}$. Elegimos $M = x + 1$. Claramente, $x < x + 1$, es decir, $x < M$. Luego para cualquier $x \in \mathbb{R}$, x está acotado por M , y \mathbb{R} está acotado. □

Formalmente, el error es que esto prueba $\forall x \in \mathbb{R}. \exists M \in \mathbb{R}. x < M$, mientras que lo pedido es $\exists M \in \mathbb{R}. \forall x \in \mathbb{R}. x < M$ (y esto último es obviamente falso).

3. Si terminan definiendo un sustantivo y no lo usan para su conclusión, o no es necesario, pueden removerlo al terminar. Pero si no empezamos dándole nombre, seguro no lo podemos usar.

Nombrar los objetos que usamos nos deja ser creativos al ver relaciones entre ellos, y crear aún más objetos a partir de ellos.



Figura 3: Neo intentando probar algo sin darle nombre a todas las cosas.

2. Cuantifiquen todo.
 1. Si usan una variable, cuantifíquenla. Una variable sin cuantificar es inútil. « G es conexo.» ¿Quién es G ? ¿Vale para todo G ? ¿Existe algún G ? ¿Es un G particular que definimos nosotros?
 2. Presten atención al anidado de cuantificadores. En $\forall x \in X. \exists y \in Y. P(x, y)$, y puede depender de x , pero x no puede depender de y . La oración « $\exists x \in X. \forall y \in Y. P(x, y)$ » es completamente distinta, no tienen nada que ver una con la otra. Recuerden la Sección 4.2, donde interpretamos demostraciones como una conversación entre nosotros y alguien que no está pidiendo demostrarles algo.
 3. Usen ecuaciones y desigualdades. En vez de decir «El peor caso es que $m = n$ », digan explícitamente que $m \leq n$, o $m \geq n$, sea cual fuere el caso. Muchas veces cometen el error de asumir que un objeto es «un peor caso» (y luego basta probar lo que tienen que probar sólo para ese objeto), pero están confundiéndose con la dirección de la desigualdad. Razonen formalmente, usen ecuaciones y desigualdades.
 4. Usen lenguaje formal, cuando existe. La oración «La función seno se ve igual cada 2π .» es vaga. ¿Qué significa «se ve igual»? ¿Quién la «ve», y cómo? Escribir esto con precisión resultaría en «Para todo $x \in \mathbb{R}$, $\sin(x) = \sin(x + 2\pi)$ », que es preciso, y nos da una ecuación con la cual trabajar y reemplazar en el futuro.
 5. Sean claros en qué es lo que afirman, qué es lo que asumen, y qué es lo que quieren probar. Ver un montón de oraciones donde todas son afirmaciones dificulta la comprensión. Usen conectores lógicos, como «porque», «luego», «si», y «entonces». Pueden usar frases como «Vamos a probar que», «Acá usamos la hipótesis tal», «Asumimos por contradicción que tal cosa», «Vamos a usar tal estrategia (inducción, partir en casos, etcétera)».

5.2 No decir nada

Si la demostración les salió en una oración, está mal. Generalmente veo esto cuando sólo están reiterando el enunciado, o reiterando la conclusión, y no hay ningún argumento en el medio que los conecte. Saben que tienen que ir de P a Q , entonces saben que al menos van a tener que mencionar a P y a Q , se confunden, y sólo dicen « P . Luego Q ». o sólo « Q ». Si la demostración fuera una sola

oración, no sería un ejercicio de una materia universitaria. Si un ejercicio les resultó totalmente trivial, probablemente lo hicieron mal. Vuelvan a leer la Sección 4.2, sobre comprender qué nos están pidiendo.

⚠️ Advertencia

Esto es un ejemplo de un alumno, donde sólo se reitera lo que hay que probar.

Ejercicio 1.5.2

Dado un grafo $G = (V, E)$ y un vértice $v \in V$, un árbol generador T de G es v -geodésico si $\text{dist}_G(v, w) = \text{dist}_T(v, w)$ para todo $w \in W$.

Sea T un árbol que genera BFS al comenzar desde un vértice v . Probar que T es v -geodésico. ↗

Demostración. El árbol que queda explícitamente definido después de correr BFS en G con el vértice v cumple que $\text{dist}_T(v, w)$ es igual a $\text{dist}_G(v, w)$ para todo $w \in V$. Por lo tanto si T es el árbol de BFS en G enraizado en v , entonces queda probado que T es v -geodésico. □

Esto no prueba nada. Sólo reitera la conclusión.

Esto sucede también cuando afirman proposiciones sin probarlas, a veces simplemente diciendo que «es obvio». La proposición «Todo árbol de al menos dos vértices tiene al menos dos hojas» es «obvia», e imagino que la mayoría no la puede probar fácilmente. Frecuentemente piensan en varios ejemplos, todos cumplen una propiedad, y concluyen que «es obvio». Lo que es obvio es *que esos ejemplos la cumplen*, lo que no es obvio es *cómo demostrar que todos los objetos la cumplen*. Después de todo, esta conjeta:

Conjetura 1.5.3 (Goldbach)

Para todo $n \in \mathbb{N}$ par, $n > 2$, existen dos números primos $p, q \in \mathbb{N}$, tal que $n = p + q$. ♡

jamás ha sido probada, aún después de cientos de años de intentos. Sin embargo, absolutamente todos los números naturales pares que ustedes piensen van a cumplirla, porque no se le conocen contraejemplos. Sólo afirmarla porque no se nos ocurren contraejemplos no dice nada.

En general, pueden usar sin probar (pero mencionando qué es lo que están usando!) lo que hayan visto en materias correlativas, y en el material demostrado en clase. Si quieren usar algo más, pregúnten si lo pueden usar. La respuesta muchas veces va a ser «Lo podés usar sólo si lo podés probar», que es lo mismo que «No».

5.3 Empezar con la conclusión

No empiecen con la conclusión e intenten probar la premisa. Si logran hacer esto, de milagro, usando sólo implicaciones bilaterales (\Leftrightarrow), en el mejor caso es una pobre y confusa exposición de la implicación pedida. En el peor caso, casi siempre van a cometer el error de usar una implicación que no tiene vuelta válida, y su demostración no va a decir nada.

⚠️ Advertencia

Asumir la conclusión y probar algo cierto no dice nada.

Postulado 1.5.4

$$1 = -1$$



Demostración.

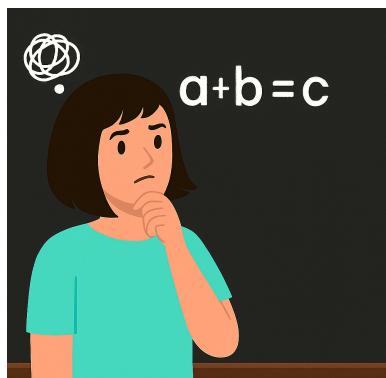
$$\begin{aligned} 1 &= -1 && \text{hago lo mismo en ambos lados} \\ 1^2 &= (-1)^2 && \text{simplifico} \\ 1 &= 1 \end{aligned}$$

Como llegamos a algo cierto, debemos haber empezado con algo cierto. □

Si llegaron a algo razonando así, este es el momento para emprolijar su idea, y empezar desde el principio. No se preocupen porque «parezca un galerazo» que su objeto justo cumpla lo pedido al final. Esa es una preocupación pedagógica del docente, no de ustedes.

5.4 No entender qué estamos asumiendo

Frecuentemente vemos que usan algo sin saber que lo están usando. Por ejemplo, «sean $u, v \in V$ tales que hay un camino entre u y v ». Nada en esto nos deja concluir que $u \neq v$. Si luego asumimos que el camino tiene longitud mayor a cero, por ejemplo diciendo «sea $P = [u, x, \dots, v]$ el camino entre u y v , y tomemos la arista (u, x) » (que no tiene por qué existir), estamos asumiendo algo sin siquiera entender que lo estamos haciendo.



Mismo cuando decimos «sea x el vértice tal que $P(x)$ ». El decir *el* vértice implica que existe un único tal vértice que cumple P . Esto ambos requiere probar que existe, y que no hay otro que cumpla P . Si sólo queremos decir que existe (y lo demostramos anteriormente), podemos decir «sea x un vértice tal que $P(x)$ ».

Finalmente, a veces lo que asumen no es explícito. Por ejemplo, si se les pide probar que a y b comutan (es decir, que $ab = ba$), y usan $(ab)^2 = a^2b^2$, están precisamente asumiendo commutatividad para probar commutatividad.

Parte II: Fundamentos matemáticos

1 Lógica

El área de lógica es rica y profunda. En este libro vamos a ver los conceptos necesarios para entender las demostraciones matemáticas que se encuentran en el mismo. Para alumnos interesados en ver más profundamente el tema, y cómo se relaciona con la computación, recomiendo el libro [9]. Para una mirada más matemática, incluyendo los famosos teoremas de completitud e incompletitud de Gödel, recomiendo [10].

Definición 2

Una proposición es una afirmación que puede ser verdadera o falsa.



Los siguientes son ejemplos de proposiciones:

- $2 + 2 = 4$.
- El número π es irracional.
- Para todo número natural n , $n + 0 = n$.
- Existe un número primo par mayor que 2.

Mientras tanto, las siguientes no son proposiciones:

- $x + 2 = 5$ (no sabemos si es verdadera o falsa, porque no sabemos qué es x).
- «Cierre la puerta.» (no es una afirmación, es una orden).
- «¿Cuánto es $2 + 2$?» (no es una afirmación, es una pregunta).

Podemos asignarle nombres a proposiciones, por ejemplo escribiendo «Sea P : $2 + 2 = 4$.» Luego podemos usar estos nombres para construir proposiciones más complejas, usando conectores lógicos. Por ejemplo, si decimos «Sea P : $2 + 2 = 4$ y Q : $3 + 3 = 6$.», entonces podemos construir la proposición $P \wedge Q$, que es verdadera sólo si ambas P y Q son verdaderas. Otros conectores lógicos son:

Símbolo	Definición
\wedge	«Y». La expresión $A \wedge B$ significa que valen ambas proposiciones A y B .
\vee	«O». La expresión $A \vee B$ significa que vale al menos una de las proposiciones A o B . En particular, si vale A , entonces vale $A \vee B$, sin importar si vale o no B . Lo mismo si vale B .
\neg	«No». La expresión $\neg A$ significa que no vale la proposición A . Si vale A , entonces no vale $\neg A$. Si no vale A , entonces vale $\neg A$. Notar que \neg liga fuertemente a una variable o expresión, luego $\neg A \vee B$ significa $(\neg A) \vee B$.
\Rightarrow	«Implica». La expresión $A \Rightarrow B$ significa que si vale la proposición A , entonces vale la proposición B . Si no vale A , entonces no hay nada que probar, y la expresión es cierta. Si vale A , tenemos que probar que vale B . Notar que esto es equivalente a decir que «o no vale A , o vale B », es decir, que $\neg A \vee B$ es equivalente a $A \Rightarrow B$.
\Leftrightarrow	«Si y sólo si». La expresión $A \Leftrightarrow B$ significa que ambas proposiciones son equivalentes: Si vale una, entonces vale la otra, y viceversa. Es decir, $A \Leftrightarrow B$ es lo mismo que $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

A veces vamos a querer referirnos a los valores de verdad de proposiciones complejas. Para esto, usamos los siguientes símbolos:

Símbolo	Definición
\top	«Verdadero». La proposición \top es siempre verdadera.

\perp

«Falso». La proposición \perp es siempre falsa.

Aún cuando no sepamos la veracidad de algunas proposiciones, podemos razonar sobre su composición usando tablas de verdad. En la proposición $Q : P \Rightarrow P$, no sabemos quién es P , pero podemos afirmar que Q es siempre verdadera, usando la siguiente tabla:

P	$P \Rightarrow P$
\top	\top
\perp	\top

Si tenemos más proposiciones, podemos agregar columnas a la tabla. Por ejemplo, para la proposición $R : (P \wedge Q) \Rightarrow P$, tenemos:

P	Q	$P \wedge Q$	$(P \wedge Q) \Rightarrow P$
\top	\top	\top	\top
\top	\perp	\perp	\top
\perp	\top	\perp	\top
\perp	\perp	\perp	\top

Definición 3

Un predicado es una proposición que depende de una o más variables.



Por ejemplo, « x es par» es un predicado que depende de la variable x . Si le damos un valor a x , obtenemos una proposición. Por ejemplo, si $x = 4$, entonces el predicado « x es par» se vuelve la proposición «4 es par», que es verdadera. Si P es un predicado que depende de una variable, podemos escribir $P(x)$ para referirnos a la proposición que resulta al darle el valor x a la variable. Por ejemplo, si $P(x) : x$ es par, entonces $P(4)$ es la proposición «4 es par», y $P(5)$ es la proposición «5 es par».

También tenemos cuantificadores, que nos permiten introducir variables en nuestras proposiciones.

Símbolo	Definición
\forall	«Para todo», o «Sea». La oración $\forall x.P(x)$ significa que el predicado P vale para todo x . Algo común es escribir $\forall x \in X.P(x)$, que es una abreviación de $\forall x.x \in X \Rightarrow P(x)$. Notar cómo el \forall captura todo lo que viene después del «.» que le sigue al símbolo, luego no es necesario aclarar que la oración anterior es lo mismo que $\forall x.(x \in X \Rightarrow P(x))$.
\exists	«Existe». La oración $\exists y.P(y)$ significa que hay al menos un y tal que el predicado P vale para y . Algo común es escribir $\exists y \in Y.P(y)$, que es una abreviación de $\exists y.y \in Y \wedge P(y)$. Al igual que \forall , el símbolo \exists captura todo lo que viene después del «.» que le sigue al símbolo, luego no es necesario aclarar que la oración anterior es lo mismo que $\exists y.(y \in Y \wedge P(y))$. Como abreviación, se usa $\exists!x \in X.P(x)$ para significar «Existe un único x en X , tal que $P(x)$ ». Puede haber otros x que cumplan $P(x)$, pero en X sólo hay uno.

Crear variables nos da muchísimo poder, y nos lleva de la lógica proposicional, donde sólo tenemos proposiciones y predicados, a la lógica de primer orden, donde tenemos variables y cuantificadores.

La lógica de primer orden es la base de las demostraciones matemáticas, y es lo que vamos a usar en este libro.

Oración	Significado	Cómo probarla
Para todo $x \in \mathbb{R}$, tenemos que $x^2 \geq 0$.	Para cualquier x que elijamos en los reales, x^2 es mayor o igual a cero. Otra forma de escribir esto es $\forall x. (x \in \mathbb{R} \Rightarrow x^2 \geq 0)$. Es decir, para todo x , si x está en los reales, entonces $x^2 \geq 0$.	Nos van a dar un x , y sabemos que $x \in \mathbb{R}$. Tenemos que probar que $x^2 \geq 0$. Podemos partir en casos, dependiendo de si $x \geq 0$ o $x < 0$. En el primero, el producto de dos números no-negativos es no-negativo, y en el segundo caso, $x = -y$ con $y > 0$, y luego $x^2 = (-y)(-y) = y^2 > 0$, y luego en ambas ramas tenemos $x^2 \geq 0$.
Para todo $x \in \mathbb{R}$, existe un $y \in \mathbb{N}$, tal que $y > x$.	Para cualquier x que elijamos en el conjunto \mathbb{R} , hay algún y en \mathbb{N} que es más grande. Otra forma de escribir esto es $\forall x. (x \in \mathbb{R} \Rightarrow (\exists y. (y \in \mathbb{N} \wedge y > x)))$. Es decir, para todo x , si x está en \mathbb{R} , entonces existe un y , tal que y está en \mathbb{N} , y también $y > x$.	Nos van a dar un x , y sabemos que $x \in \mathbb{R}$. Tenemos que mostrar que existe un y tal que $y \in \mathbb{N}$, e $y > x$. A veces vamos a poder encontrar y explícitamente, otras veces sólo vamos a saber que existe. y puede depender de x . En este caso, podemos elegir $y = \lceil x \rceil + 1$, y sabiendo que $\lceil x \rceil \geq x$, y que $\lceil \cdot \rceil : \mathbb{R} \rightarrow \mathbb{N}$, concluimos que $y = \lceil x \rceil + 1 > x$, con $y \in \mathbb{N}$. Notar que puede haber otros y posibles, por ejemplo $\lceil x \rceil + 5$, pero basta con encontrar uno y estamos.
Existe un $x \in \mathbb{R}$, tal que para todo $y \in \{0, 1, \dots, 8\}$, $y > x$.	Hay alguien en \mathbb{R} que es menor a todo elemento de $\{0, 1, \dots, 8\}$.	Tenemos que probar que existe un tal x . A veces vamos a poder decir quién es x explícitamente, otras veces sólo vamos a poder probar que existe. Tenemos que mostrar que para este x que elegimos, sin importar qué $y \in \{0, 1, \dots, 8\}$ alguien elija, tendremos $y > x$. Podemos elegir $x = -1$, y vemos que $-1 < 0, -1 < 1, \dots, -1 < 8$, y por lo tanto $x < y$ para todo $y \in \{0, 1, \dots, 8\}$.

Existe un $x \in \mathbb{R}$, tal que para todo $y \in \{0, 1, \dots, \lceil x \rceil\}, x > y$.	Hay algún real x , tal que x es más grande que cualquier elemento en $\{0, 1, \dots, \lceil x \rceil\}$.	<p>Tenemos que dar un tal $x \in \mathbb{R}$. Llamemos $X = \{0, 1, \dots, \lceil x \rceil\}$. Veamos a quién podemos elegir:</p> <ul style="list-style-type: none"> • Si elegimos un $x > -1$, entonces X tiene al menos 1 elemento ($\lceil x \rceil \geq 0$). Como $\lceil x \rceil \geq x$, y $\lceil x \rceil \in X$, nunca vamos a poder probar que $x > y$ para todo $y \in X$, pues alguien podría darnos $y = \lceil x \rceil \geq x$. • Si elegimos un $x \leq -1$, entonces $X = \{0, 1, \dots, \lceil x \rceil\}$ con $\lceil x \rceil < 0$, y luego $X = \emptyset$. Luego, «para todo $y \in \emptyset, x > y$» es trivialmente cierto sobre x, puesto que no hay ningún tal $y \in \emptyset$. Luego, podemos elegir $x = -1$ (o si quisieramos, $x = -47$), y vemos que este x cumple lo pedido.
Para todo $n \in \mathbb{N}$ tal que $n > 1$, para todo $m \in \mathbb{N}$ tal que $m \geq 2n$, existe un $p \in \mathbb{N}$ tal que $n < p < m$, y p es primo.	Esto nos dice que siempre que tengamos dos números naturales n y m , si sabemos que $n > 1$ y $m \geq 2n$, entonces vamos a poder encontrar un primo entre n y m .	<p>Nos van a dar dos números naturales, n y m, y sabemos que $n > 1$ y $m \geq 2n$. Tenemos que probar que existe un primo p tal que $n < p < m$.</p> <p>Esta proposición es un corolario del postulado de Bertrand.</p>

1.1 Ejercicios

Ejercicio 2.1.1

Sea P la proposición «Está lloviendo», Q la proposición «Voy a llevar un paraguas», y R la proposición «Me voy a mojar». Escribir las siguientes proposiciones usando P, Q, R y los conectores lógicos:

- «Está lloviendo y voy a llevar un paraguas.»
- «Si está lloviendo, entonces voy a llevar un paraguas.»
- «No me voy a mojar si y sólo si llevo un paraguas.»
- «No está lloviendo, o me voy a mojar.»
- «Si no llevo un paraguas, entonces me voy a mojar.»

Ejercicio 2.1.2

Dados $P : \top$, $Q : \perp$, y $R : \top$, determinar el valor de verdad de las siguientes proposiciones, usando una tabla de verdad:

- $P \wedge Q$
- $(\neg P) \vee Q$
- $P \Rightarrow Q$
- $(P \wedge R) \Rightarrow \neg Q$
- $\neg(P \vee Q) \Leftrightarrow ((\neg P) \wedge (\neg Q))$

Ejercicio 2.1.3

Sean P, Q proposiciones. Demostrar que $(P \Rightarrow Q) \vee (Q \Rightarrow P)$.

Ejercicio 2.1.4

Sean P, Q proposiciones. Probar que $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$.

Esta fórmula se conoce como la Ley de Peirce[11].

Ejercicio 2.1.5

Para cada una de las siguientes proposiciones:

1. $\exists x.x^2 = 2$.
2. $\forall x.\exists y.x^2 = y$.
3. $\forall y.\exists x.x^2 = y$.
4. $\forall x.(x \neq 0 \Rightarrow \exists y.xy = 1)$.
5. $\forall x\exists y.2x - y = 0$
6. $\forall x\exists y.x - 2y = 0$
7. $\forall x.((x > 10) \Rightarrow (\forall y.(y < x \Rightarrow y < 9)))$
8. $\forall x.\exists y.(y > x \wedge \exists z.(y + z = 100))$
9. $\exists x.\exists y.x + 2y = 2 \wedge 2x + 4y = 5$

Indicar cuáles son **falsas** cuando x, y, z son:

1. Números naturales.
2. Números enteros.
3. Números reales.

Ejercicio 2.1.6

Mostrar que la siguiente proposición es falsa para algún conjunto D , y alguna proposición P :

$$(\forall x \in D.\exists y \in D.P(x, y)) \Rightarrow (\forall z \in D.P(z, z))$$

2 Conjuntos

Definición 4

Un conjunto es una colección no-ordenada de elementos distintos.

Si A es un conjunto, y a es un elemento del conjunto, escribimos $a \in A$. Si b no es un elemento del conjunto, escribimos $b \notin A$. Podemos escribir conjuntos usando llaves, por ejemplo, $A = \{1, 2, 3, 4\}$ es el conjunto cuyos elementos son exactamente 1, 2, 3, 4, y ningún otro. Como los conjuntos no son ordenados, $\{4, 3, 2, 1\}$ es el mismo conjunto que el anterior. Asimismo, como los elementos deben ser distintos, $\{1, 2, 2, 3, 4, 4\}$ es también el mismo conjunto.

Definición 5

Dos conjuntos son iguales cuando tienen los mismos elementos. Escribimos $A = B$ si y sólo si para todo x , $x \in A$ si y sólo si $x \in B$.



También podemos escribir conjuntos por comprensión, diciendo que son todos los elementos que cumplen alguna propiedad. Por ejemplo, $C = \{x \in \{1, 2, 3, 4\} \mid x \text{ es par}\}$ es el conjunto de todos los elementos pares en $\{1, 2, 3, 4\}$, es decir, $C = \{2, 4\}$.

Si todos los elementos de A están en B , notamos $A \subseteq B$. Si la inclusión es estricta, es decir, si $A \subseteq B$ y existe algún elemento en B que no está en A , notamos $A \subset B$.

Finalmente, notamos al conjunto vacío, que no tiene elementos, como \emptyset . Otros conjuntos conocidos son el conjunto de los números naturales, \mathbb{N} , el conjunto de los enteros, \mathbb{Z} , el conjunto de los racionales, \mathbb{Q} , y el conjunto de los reales, \mathbb{R} .

Veamos algunas propiedades fundamentales sobre conjuntos.

Proposición 2.2.1

Sean A y B conjuntos. Entonces $A = B$ si y sólo si $A \subseteq B$ y $B \subseteq A$.



Demostración. Para probar un si-y-sólo-si, probamos ambas implicaciones.

- \Rightarrow) Asumimos que $A = B$, queremos probar que $A \subseteq B$ y $B \subseteq A$. Sea $x \in A$. Como $A = B$, entonces $x \in B$. Luego, por definición de inclusión, $A \subseteq B$. Análogamente, si $y \in B$, como $A = B$, entonces $y \in A$, y luego $B \subseteq A$.
- \Leftarrow) Asumimos que $A \subseteq B$ y $B \subseteq A$, queremos ver que $A = B$. Sea x un elemento cualquiera. Vamos a probar que $x \in A$ si y sólo si $x \in B$.
 1. \Rightarrow) Asumimos que $x \in A$. Como $A \subseteq B$, entonces $x \in B$.
 2. \Leftarrow) Asumimos que $x \in B$. Como $B \subseteq A$, entonces $x \in A$.

Luego, x está en A si y sólo si x está en B . Como x era arbitrario, entonces $A = B$.



Proposición 2.2.2

Sean A , B , y C conjuntos. Si $A \subseteq B$ y $B \subseteq C$, entonces $A \subseteq C$.



Demostración. Para probar que $A \subseteq C$ tenemos que probar que cualquier elemento de A está también en C . Sea entonces $x \in A$. Como $A \subseteq B$, entonces $x \in B$. Luego, como $B \subseteq C$, tenemos que $x \in C$. Luego, cualquier elemento de A está en C , que es la definición de $A \subseteq C$.

Notar que la vuelta no vale necesariamente. Por ejemplo, si $A = \{1\}$, $B = \{3\}$, y $C = \{1, 2\}$, entonces $A \subseteq C$, pero no valen ni $A \subseteq B$ ni $B \subseteq C$. \square

Dados dos conjuntos, vamos a definir operaciones útiles.

Definición 6

Dados dos conjuntos A y B , definimos:

- La intersección de A y B , notada $A \cap B$, como el conjunto de todos los elementos que están en A y en B . Es decir, $A \cap B = \{x \mid x \in A \text{ y } x \in B\}$.
- La unión de A y B , notada $A \cup B$, como el conjunto de todos los elementos que están en A o en B . Es decir, $A \cup B = \{x \mid x \in A \text{ o } x \in B\}$. Notar que el «o» no es exclusivo, es decir, si x está en ambos conjuntos, está en la unión. Si queremos comunicar que la unión es de conjuntos disjuntos (es decir, que $A \cap B = \emptyset$) escribimos $A \sqcup B$.
- La diferencia entre A y B , notada $A \setminus B$, como el conjunto de todos los elementos que están en A pero no en B . Es decir, $A \setminus B = \{x \mid x \in A \text{ y } x \notin B\}$. Notar que la diferencia no es simétrica, es decir, en general $A \setminus B \neq B \setminus A$.
- La diferencia simétrica entre A y B , notada $A \triangle B$, es $(A \setminus B) \cup (B \setminus A)$.
- El complemento de A , notado A^c , como el conjunto de todos los elementos que no están en A . Es decir, $A^c = \{x \mid x \notin A\}$.



Hay reglas muy útiles que relacionan estas operaciones entre sí, llamadas las Leyes de De Morgan[12].

Proposición 2.2.3 (Leyes de De Morgan)

Sean A, B, C conjuntos. Entonces:

$$\begin{aligned} A \cup (B \cap C) &= (A \cup B) \cap (A \cup C) \\ A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \end{aligned}$$



Demostración. Para la primer ecuación:

$$\begin{aligned} A \cup (B \cap C) &= \{x \mid x \in A \vee x \in (B \cap C)\} \\ &= \{x \mid x \in A \vee (x \in B \wedge x \in C)\} \\ &= \{x \mid (x \in A \vee x \in B) \wedge (x \in A \vee x \in C)\} \\ &= (A \cup B) \cap (A \cup C) \end{aligned}$$

y para la segunda:

$$\begin{aligned} A \cap (B \cup C) &= \{x \mid x \in A \wedge (x \in B \vee x \in C)\} \\ &= \{x \mid (x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)\} \\ &= (A \cap B) \cup (A \cap C) \end{aligned}$$



La unión y la intersección son operaciones duales, usando el complemento.

Proposición 2.2.4

Sean A, B conjuntos. Entonces

$$(A \cup B)^c = A^c \cap B^c$$

Como corolario, $(A \cap B)^c = A^c \cup B^c$.



Demostración. Vamos a probar una igualdad de conjuntos probando que cada uno está incluido en el otro.

- $(A \cup B)^c \subseteq A^c \cap B^c$: Sea $x \in (A \cup B)^c$. Entonces $x \notin (A \cup B)$. Como $A \cup B = \{y \mid y \in A \text{ o } y \in B\}$, entonces tenemos que es falso que $(x \in A \text{ o } x \in B)$, y luego tenemos que $x \notin A$ y $x \notin B$. Pero $A^c = \{y \mid y \notin A\}$, y análogamente para B^c , luego $x \in A^c$ y $x \in B^c$. Por definición de \cap , entonces, $x \in A^c \cap B^c$.
- $A^c \cap B^c \subseteq (A \cup B)^c$. Sea $x \in A^c \cap B^c$. Luego por definición de \cap , $x \in A^c$ y $x \in B^c$. Luego por definición de X^c para conjuntos X , $x \notin A$, y $x \notin B$. Luego $x \notin A \cup B$, dado que $A \cup B = \{y \mid y \in A \text{ o } y \in B\}$. Luego $x \in (A \cup B)^c$.

El corolario es inmediato, pues $(A \cap B)^c = ((A^c \cup B^c))^c = A^c \cup B^c$.



Proposición 2.2.5

Sean A, B conjuntos. Entonces $B \setminus (B \setminus A) = A \cap B$.



Demostración. Hagamos esto enteramente por definición.

$$\begin{aligned} B \setminus (B \setminus A) &= \{x \mid x \in B \wedge x \notin (B \setminus A)\} \\ &= \{x \mid x \in B \wedge x \notin \{y \in B \mid y \notin A\}\} \\ &= \{x \mid x \in B \wedge (x \notin B \vee (x \in B \wedge x \in A))\} \\ &= \{x \mid x \in B \wedge x \in A\} \\ &= A \cap B \end{aligned}$$



Proposición 2.2.6

Sean A, B, C conjuntos. Entonces $(A \setminus B) \setminus C \subseteq A \setminus (B \setminus C)$.



Demostración. Sea $x \in (A \setminus B) \setminus C$. Entonces $x \in A$, $x \notin B$, y $x \notin C$. Como $x \notin B$, entonces con más razon $x \notin (B \setminus C)$, dado que $B \setminus C \subseteq B$. Luego $x \in A \wedge x \notin (B \setminus C)$, y luego $x \in A \setminus (B \setminus C)$.

La igualdad no vale en general. Por ejemplo, podemos tomar $A = \{1, 2, 3\}$, $B = \{2, 3\}$, y $C = \{3, 4, 5\}$. Entonces $(A \setminus B) \setminus C = \{1\}$, pero $A \setminus (B \setminus C) = \{1, 3\}$.



Proposición 2.2.7

Sea $f : X \rightarrow Y$ una función, y denotemos por $f^{-1}(C) = \{x \in X \mid f(x) \in C\}$, para cualquier subconjunto $C \subseteq Y$.

Entonces para todo A, B subconjuntos de Y , tenemos que $f^{-1}(A \Delta B) = f^{-1}(A) \Delta f^{-1}(B)$. ♥

Demostración.

- \subseteq : Sea $x \in f^{-1}(A \Delta B)$. Entonces $f(x) \in A \Delta B$, es decir, $f(x) \in (A \setminus B) \cup (B \setminus A)$. Partimos en casos:
 1. Si $f(x) \in A \setminus B$, entonces $f(x) \notin B$, y $f(x) \in A$. Luego, $x \notin f^{-1}(B)$, y $x \in f^{-1}(A)$, y luego $x \in (f^{-1}(A) \setminus f^{-1}(B)) \subseteq f^{-1}(A) \Delta f^{-1}(B)$.
 2. Si $f(x) \in B \setminus A$ sucede algo análogo, y por ende $x \in f^{-1}(A) \Delta f^{-1}(B)$.
- \supseteq : Sea $x \in f^{-1}(A) \Delta f^{-1}(B) = (f^{-1}(A) \setminus f^{-1}(B)) \cup (f^{-1}(B) \setminus f^{-1}(A))$. Partimos en casos:
 1. Si $x \in f^{-1}(A) \setminus f^{-1}(B)$, entonces $f(x) \in A$, pero $f(x) \notin B$. Luego $f(x) \in (A \setminus B) \subseteq A \Delta B$, y por lo tanto $x \in f^{-1}(A \Delta B)$.
 2. Si $x \in f^{-1}(B) \setminus f^{-1}(A)$, pasa algo análogo, y tenemos que $x \in f^{-1}(A \Delta B)$.

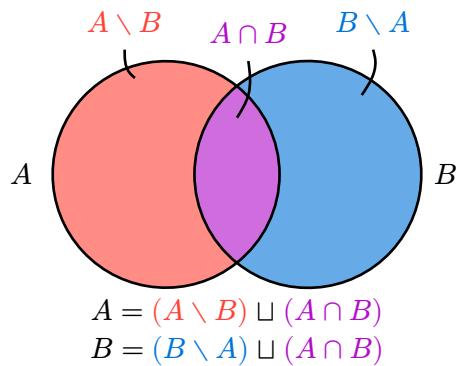
□

Proposición 2.2.8

Sean A y B conjuntos tal que $|A| = |B|$. Entonces:

1. $|A \setminus B| = |B \setminus A|$.
2. $|\Delta(A, B)|$ es par.
3. Si $\Delta(A, B) \neq \emptyset$, entonces $(A \setminus B) \neq \emptyset$ y $(B \setminus A) \neq \emptyset$. ♥

Demostración. Recordemos cómo funcionan las uniones e intersecciones de conjuntos.



Sabemos que $A = (A \setminus B) \sqcup (A \cap B)$, y $B = (B \setminus A) \sqcup (A \cap B)$. Llamando a $|A \setminus B| = \alpha$, $|B \setminus A| = \beta$, y $|A \cap B| = \gamma$, tenemos que $|A| = \alpha + \gamma$, y $|B| = \beta + \gamma$. Como $|A| = |B|$, tenemos que $\alpha + \gamma = \beta + \gamma$. Luego, $\alpha = \beta$, es decir, $|A \setminus B| = |B \setminus A|$.

Para el segundo punto, como $\Delta(A, B) = (A \setminus B) \sqcup (B \setminus A)$, tenemos que $|\Delta(A, B)| = |(A \setminus B)| + |(B \setminus A)| = \alpha + \beta = 2\alpha = 2\beta$, luego es par.

Finalmente, si $\Delta(A, B) \neq \emptyset$, entonces $|\Delta(A, B)| = 2\alpha = 2\beta > 0$, luego $\alpha = \beta > 0$, y luego $(A \setminus B) \neq \emptyset$, como también $(B \setminus A) \neq \emptyset$.

□

2.1 Ejercicios

Ejercicio 2.2.9

Sea A un conjunto. Probar que $(A^c)^c = A$.

♦

Ejercicio 2.2.10

Sean A, B conjuntos. Probar que las siguientes proposiciones son equivalentes:

- $A \subseteq B$
- $A \cap B = A$
- $A \cup B = B$

♦

Ejercicio 2.2.11

Sean A, B conjuntos. Probar que:

$$A \setminus (A \setminus B) = A \cap B$$

♦

Ejercicio 2.2.12

Sean A, B, C conjuntos. Probar que:

$$A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$$

♦

Ejercicio 2.2.13

Sean A, B, C conjuntos. Probar que la siguiente proposición es cierta, o exhibir un contraejemplo.

$$(A \cup B) \setminus C = A \cup (B \setminus C)$$

♦

Ejercicio 2.2.14

Se define el producto cartesiano de dos conjuntos A y B como $A \times B = \{(a, b) \mid a \in A, b \in B\}$. Sean A, B, C conjuntos. Probar que:

$$A \times (B \setminus C) = (A \times B) \setminus (A \times C)$$

♦

Ejercicio 2.2.15

Se define el conjunto de partes de un conjunto A como $\mathcal{P}(A) = \{X \mid X \subseteq A\}$. Probar que:

$$\mathcal{P}(A) \cap \mathcal{P}(B) = \mathcal{P}(A \cap B)$$

Ejercicio 2.2.16

Sean A, B, C conjuntos. Probar que:

$$A \cap (B \Delta C) = (A \cap B) \Delta (A \cap C)$$

Ejercicio 2.2.17

Sean A, B conjuntos. Probar que si $A \Delta B = A \Delta C$, entonces $B = C$.

Pueden ver esto como un análogo a que para variables enteras, $X \oplus (X \oplus Y) = Y$, es decir, el o-exclusivo es cancelativo.

Ejercicio 2.2.18

Sea $f : X \rightarrow Y$ una función, y $A, B \subseteq Y$. Probar que:

$$f^{-1}(A \setminus B) = f^{-1}(A) \setminus f^{-1}(B)$$

Ejercicio 2.2.19

Sea $f : X \rightarrow Y$ una función, y $A, B \subseteq X$. Probar que la siguiente proposición es cierta, o dar un contraejemplo:

$$f(A \cap B) = f(A) \cap f(B)$$

3 Funciones

Imagino que ya tienen en mente una noción del concepto de función. En esta sección quiero mostrarles una definición formal, y algunas propiedades básicas, para que vean cómo se hacen demostraciones sobre funciones.

Definición 7

Sean A, B conjuntos, $A \times B = \{(a, b) \mid a \in A, b \in B\}$ el producto cartesiano entre A y B . Una **función** f es un subconjunto de $A \times B$, tal que para todo $a \in A$, existe un único $b \in B$ tal que $(a, b) \in f$. Escribimos $f : A \rightarrow B$. Llamamos a A el **dominio** de f , y a B el **codominio** de f .

Usualmente vamos a definir funciones usando expresiones de la forma $f(X) = Y$, donde X es un elemento del dominio, e Y un elemento del codominio, cuya expresión depende de X . Por ejemplo, $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Z}$, $f(n, m) = n + m$, $g : \mathbb{R} \rightarrow \mathbb{N}$, $g(x) = 45$, o $h : \mathbb{Q} \rightarrow \mathbb{Q} \times \mathbb{A} \times \mathbb{N}$, $h(y) = (y, k(y + 3), 2)$. Notar que algunas tales expresiones no definen una única función, por ejemplo $f : \mathbb{R} \rightarrow \mathbb{R}$, $f(x) = f(x)$, $g : \mathbb{R} \rightarrow \mathbb{R}$, $g(x) = 2 + g(x - 2)$, o aún $g : \mathbb{Z} \rightarrow \mathbb{Z}$, $h(x) = 2k(x + 1)$ con $k : \mathbb{Z} \rightarrow \mathbb{Z}$, $k(x) = \frac{h(x-1)}{2}$.

Definición 8

Sea $f : A \rightarrow B$ una función. Decimos que f es:

- **inyectiva** si para todo $x, x' \in A$, si $f(x) = f(x')$, entonces $x = x'$.
- **sobreyectiva** si para todo $y \in B$, existe algún $x \in A$ tal que $f(x) = y$.
- **biyectiva** si es inyectiva y sobreyectiva.

Definición 9

Sean $f : A \rightarrow B, g : B \rightarrow C$ funciones. Definimos la **composición** de f y g , notada $g \circ f : A \rightarrow C$, como la función dada por $(g \circ f)(x) = g(f(x))$ para todo $x \in A$.

Definición 10

Sea A un conjunto. Definimos $\text{id}_A : A \rightarrow A$ como la función identidad sobre A , dada por $\text{id}_A(x) = x$ para todo $x \in A$.

Proposición 2.3.1

Sea $f : A \rightarrow B$ una función. Entonces f es biyectiva si y sólo si existe una función $g : B \rightarrow A$ tal que $f \circ g = \text{id}_B$, y $g \circ f = \text{id}_A$.

Demostración.

- $\Rightarrow)$ Asumimos que f es biyectiva. Definimos $g : B \rightarrow A$ como sigue: para cada $y \in B$, como f es sobreyectiva, existe algún $x \in A$ tal que $f(x) = y$. Como f es inyectiva, este x es único. Definimos entonces $g(y) = x$.

Veamos que $f \circ g = \text{id}_B$. Sea $y \in B$. Entonces, por definición de g , $f(g(y)) = f(x) = y$, luego $(f \circ g)(y) = y = \text{id}_B(y)$ para todo y , y luego $f \circ g = \text{id}_B$. Análogamente, sea $x \in A$. Entonces, por definición de f , $g(f(x)) = g(y) = x$, luego $(g \circ f)(x) = x = \text{id}_A(x)$ para todo x , y luego $g \circ f = \text{id}_A$.

- $\Leftarrow)$ Asumimos que existe una función $g : B \rightarrow A$ tal que $f \circ g = \text{id}_B$, y $g \circ f = \text{id}_A$.

Queremos probar que f es biyectiva, es decir, inyectiva y sobreyectiva.

1. Inyectiva: Sean $x, x' \in A$ tal que $f(x) = f(x')$. Podemos aplicarle g a ambos lados de la ecuación, obteniendo $g(f(x)) = g(f(x'))$, luego $(g \circ f)(x) = (g \circ f)(x')$. Como $g \circ f = \text{id}_A$, entonces $\text{id}_A(x) = \text{id}_A(x')$, y luego $x = x'$, probando que f es inyectiva.
2. Sobreyectiva: Sea cualquier $y \in B$, y definamos $x = g(y)$. Entonces podemos aplicar f a ambos lados, obteniendo $f(x) = f(g(y))$, o equivalentemente, $f(x) = (f \circ g)(y)$. Como $f \circ g = \text{id}_B$, tenemos $f(x) = \text{id}_B(y)$, y luego $f(x) = y$. Como esto vale para todo $y \in B$, vemos que f es sobreyectiva.

□

Proposición 2.3.2

Sean $f : A \rightarrow B, g : B \rightarrow C$ funciones. Si f y g son inyectivas, entonces $g \circ f$ es inyectiva.

♡

Demostración. Sea $x, x' \in A$ tal que $(g \circ f)(x) = (g \circ f)(x')$. Entonces, por definición de composición, $g(f(x)) = g(f(x'))$. Como g es inyectiva, entonces $f(x) = f(x')$. Como f es inyectiva, entonces $x = x'$. Luego, $g \circ f$ es inyectiva. \square

Proposición 2.3.3

Sean $f : A \rightarrow B, g : B \rightarrow C$ funciones. Si $g \circ f$ es inyectiva, entonces f es inyectiva.



Demostración. Sea $x, x' \in A$ tal que $f(x) = f(x')$. Entonces, por definición de composición, $(g \circ f)(x) = g(f(x)) = g(f(x')) = (g \circ f)(x')$. Como $g \circ f$ es inyectiva, entonces $x = x'$. Luego, f es inyectiva.

Notemos que no necesariamente g es inyectiva. Por ejemplo, si $A = \{1\}, B = \{2, 3\}$, y $C = \{4\}$, y definimos $f : A \rightarrow B$ como $f(1) = 2$, y $g : B \rightarrow C$ como $g(2) = g(3) = 4$, entonces $g \circ f : A \rightarrow C$ es inyectiva, pero g no lo es. \square

Proposición 2.3.4

Sean $f : A \rightarrow B, g : B \rightarrow C$ funciones. Si f y g son sobreyectivas, entonces $g \circ f$ es sobreyectiva.



Demostración. Sea $z \in C$. Como g es sobreyectiva, existe algún $y \in B$ tal que $g(y) = z$. Como f es sobreyectiva, existe algún $x \in A$ tal que $f(x) = y$. Luego, $(g \circ f)(x) = g(f(x)) = g(y) = z$. Luego, $g \circ f$ es sobreyectiva. \square

Proposición 2.3.5

Sean $f : A \rightarrow B, g : B \rightarrow C$ funciones. Si $g \circ f$ es sobreyectiva, entonces g es sobreyectiva.



Demostración. Sea $z \in C$. Como $g \circ f$ es sobreyectiva, existe algún $x \in A$ tal que $(g \circ f)(x) = z$. Por definición de composición, $g(f(x)) = z$. Llamando $y = f(x)$, tenemos que $y \in B$, y $g(y) = z$. Luego, g es sobreyectiva.

Notemos que no hace falta que f sea sobreyectiva. Por ejemplo, si $A = \{1, 0\}, B = \{1, 0, -1\}, C = \{1, 0\}$, y definimos $f : A \rightarrow B$ como $f(1) = 1, f(0) = 0$, y $g : B \rightarrow C$ como $g(1) = 1, g(0) = 0, g(-1) = 0$, entonces $g \circ f : A \rightarrow C$ es sobreyectiva, pero f no lo es. \square

Proposición 2.3.6

Sea $f : A \rightarrow B$ una función. Si existe una función $g : B \rightarrow A$ tal que $f \circ g = \text{id}_B$, y $g \circ f = \text{id}_A$, entonces f es biyectiva, y $f^{-1} = g$.



Demostración. Para probar que f es biyectiva, tenemos que mostrar que es inyectiva y sobreyectiva.

- Sean x, x' tal que $f(x) = f(x')$. Podemos aplicarle g a ambos lados de la ecuación, obteniendo $g(f(x)) = g(f(x'))$, luego $(g \circ f)(x) = (g \circ f)(x')$. El enunciado nos dice que $g \circ f = \text{id}_A$, luego esto es $\text{id}_A(x) = \text{id}_A(x')$, pero $\text{id}_A(y) = y$ para todo $y \in A$, luego esto nos dice que $x = x'$, probando que f es inyectiva.
- Sea cualquier $y \in B$, y definamos $x = g(y)$. Entonces podemos aplicar f a ambos lados, obteniendo $f(x) = f(g(y))$, o equivalentemente, $f(x) = (f \circ g)(y)$. El enunciado nos dice que $f \circ g = \text{id}_B$, entonces sabemos que $f(x) = \text{id}_B(y)$. Pero $\text{id}_B(y) = y$, y luego $f(x) = y$. Luego, para todo $y \in B$, encontramos un $x \in A$ tal que $f(x) = y$, probando que y es sobreyectiva.

Luego f es biyectiva. Para ver que $f^{-1} = g$, tenemos que probar que $f^{-1}(y) = g(y)$ para todo $y \in B$. Sea $y \in B$. Como f es sobreyectiva, existe un $x \in A$ tal que $f(x) = y$. Luego, $f^{-1}(y) = f^{-1}(f(x)) = x$ por definición de función inversa. Asimismo, $g(y) = g(f(x)) = (g \circ f)(x) = \text{id}_A(x) = x$. Luego ambos $f^{-1}(y)$ y $g(y)$ son iguales a x , y luego $f^{-1}(y) = g(y)$ para todo $y \in B$, que es precisamente la definición de $f^{-1} = g$. \square

Finalmente, esta notación nos va a ser útil:

Definición 11

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, y $k \in \mathbb{R}_{\geq 0}$. Definimos las funciones $f + g$, fg , y kf , tal que para todo $n \in \mathbb{N}$:

- $(f + g)(n) = f(n) + g(n)$
- $(fg)(n) = f(n)g(n)$
- $(kf)(n) = kf(n)$

Asimismo, cuando A y B son dos conjuntos de funciones $\mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, y $k \in \mathbb{R}_{\geq 0}$, definimos:

- $A + B = \{a + b \mid a \in A, b \in B\}$
- $AB = \{ab \mid a \in A, b \in B\}$
- $kA = \{ka \mid a \in A\}$

3.1 Ejercicios

Ejercicio 2.3.7

Sea $f : A \rightarrow B$ una función. Demostrar que $f \circ \text{id}_A = f$ y que $\text{id}_B \circ f = f$.

Ejercicio 2.3.8

Para cada una de las siguientes funciones, demostrar si es inyectiva, sobreyectiva, ambas, o ninguna.

- $f : \mathbb{R} \rightarrow \mathbb{R}, f(x) = x^2$
- $f : \mathbb{N} \rightarrow \mathbb{N}, f(x) = x^2$
- $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}, f(x) = x^2$
- $f : \mathbb{Z} \rightarrow \mathbb{Z}, f(x) = 2x$
- $f : \mathbb{Q} \rightarrow \mathbb{Q}, f(x) = 2x$

Ejercicio 2.3.9

Sean $f : A \rightarrow B, g : B \rightarrow C, h : C \rightarrow D$ funciones. Demostrar que la composición es asociativa, es decir, que $h \circ (g \circ f) = (h \circ g) \circ f$.

Ejercicio 2.3.10

Sean $f : A \rightarrow B$ y $g : B \rightarrow A$ funciones tales que $g \circ f = \text{id}_A$. Demostrar que f es inyectiva y g es sobreyectiva. Dar un ejemplo donde f no sea sobreyectiva.

Ejercicio 2.3.11

Sea $f : A \rightarrow B$ una función, y $X, Y \subseteq A$. Probar o dar un contraejemplo:

- $f(X \cap Y) \subseteq f(X) \cap f(Y)$
- $f(X \cap Y) = f(X) \cap f(Y)$
- $f(X \cup Y) = f(X) \cup f(Y)$

Ejercicio 2.3.12

Sean $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Demostrar que $f(g + h) = fg + fh$.

4 Sucesiones y series

Esta sección contiene varios ejercicios resueltos sobre sumatorias. Asumo que han visto sumatorias antes, y por ende no voy a explicar qué son o qué significa la notación \sum . Para alumnos interesados en aprender más, recomiendo [13].

Ejercicio 2.4.1

Sea $n \in \mathbb{N}$. Demostrar que $\sum_{i=0}^n (2i + 1) = (n + 1)^2$.

Demostración. Vamos a probar esto por inducción, $P(n) : \sum_{i=0}^n (2i + 1) = (n + 1)^2$.

- $P(0)$. El lado izquierdo de la ecuación es $\sum_{i=0}^0 (2i + 1) = 2 \times 0 + 1 = 1$. El lado derecho es $(0 + 1)^2 = 1$. Luego, vale $P(0)$.
- Sea $n \in \mathbb{N}$. Queremos ver que $P(n) \Rightarrow P(n + 1)$. Luego, vamos a asumir $P(n)$, y vamos a ver $P(n + 1)$. Ver $P(n + 1)$ es que $\sum_{i=0}^{n+1} (2i + 1) = (n + 2)^2$. Razonemos entonces:

$$\begin{aligned}
\sum_{i=0}^{n+1} (2i + 1) &= 2(n+1) + 1 + \sum_{i=0}^n (2i + 1) \\
&= 2(n+1) + 1 + (n+1)^2, \text{ usando } P(n) \\
&= 2n + 2 + 1 + (n+1)^2 \\
&= 1^2 + 2 \times (n+1) \times 1 + (n+1)^2 \\
&= ((n+1) + 1)^2 \\
&= (n+2)^2
\end{aligned}$$

que es lo que queríamos demostrar.

Luego, vale $P(n)$ para todo $n \in \mathbb{N}$. □

Ejercicio 2.4.2

Demostrar que para todo $n \in \mathbb{N}, n \geq 5$, tenemos $2^n > n^2$.



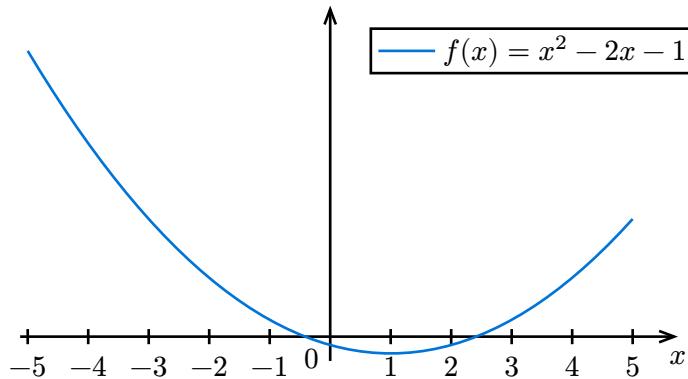
Demostración. Vamos a demostrar la propiedad $P(n) : n \geq 5 \Rightarrow 2^n > n^2$, por inducción. Para $n < 5$, la demostración es trivial, puesto que no hay nada que probar («falso implica todo»).

Luego vamos a empezar nuestra inducción con $n = 5$ como caso base.

- $P(5)$: Vemos que $2^5 = 32$, mientras que $5^2 = 25$, con lo cual efectivamente $2^5 > 5^2$, que es $P(5)$.
- Sea $n \in \mathbb{N}$. $P(n) \Rightarrow P(n+1)$: Si $n \leq 4$, vimos que $P(n+1)$, sea porque la premisa de $P(n+1)$ es falsa cuando $n < 3$, o porque $P(5)$ es cierto porque lo probamos arriba, cuando $n = 4$. Luego asumimos $n \geq 5$. Por un lado, tenemos $2^{n+1} = 2(2^n) > 2n^2$, donde usamos $P(n)$ y $n \geq 5$ para obtener la desigualdad, mediante la conclusión de $P(n)$. Por el otro, $(n+1)^2 = n^2 + 2n + 1$. Si probamos que $2n^2 > n^2 + 2n + 1$, tendremos que $2^{n+1} > (n+1)^2$.

$$\begin{aligned}
2n^2 &> n^2 + 2n + 1 \\
n^2 - 2n - 1 &> 0
\end{aligned}$$

Consideremos ahora el polinomio $f(x) = x^2 - 2x - 1$. Lo podemos factorizar como $f(x) = (x - (1 + \sqrt{2}))(x - (1 - \sqrt{2}))$, es decir que sus raíces son $a = 1 - \sqrt{2}$, y $b = 1 + \sqrt{2}$. Como el coeficiente principal de $f(x)$ es 1, f es positiva en $(-\infty, a)$, es negativa en (a, b) , y es positiva en (b, ∞) .



Como $n \geq 5 > b = 1 + \sqrt{5}$, tendremos que $f(n) > 0$, y luego $n^2 - 2n - 1 > 0$, es decir, $2n^2 > n^2 + 2n + 1$. Como dijimos, esto implica que $2^{n+1} > (n+1)^2$, lo cual prueba $P(n+1)$.

□

Ejercicio 2.4.3

Sea $(a_n)_{n \in \mathbb{N}}$ la sucesión dada por $a_0 = 1$, y para todo $n \in \mathbb{N}$, $n > 0$, $a_n = 2(n-1)a_{n-1} + 2^n(n-1)!$.

Demostrar que para todo $n \in \mathbb{N}$, $n > 0$, $a_n = 2^n n!$.

♦

Demostración. Como toda sucesión definida recursivamente, tenemos una estructura inductiva. Luego, intentemos probar esto por inducción. La proposición que vamos a probar es $P(n) : a_n = 2^n n!$.

1. $P(0)$: Si $n = 0$, entonces queremos ver $P(0)$, es decir que $a_0 = 2^0 1! = 1$. Esto es cierto, porque por definición $a_0 = 1$.
2. $P(n) \Rightarrow P(n+1)$. Queremos probar que $a_{n+1} = 2^{n+1}(n+1)!$. Sabemos que $a_{n+1} = 2na_n + 2^{n+1}n!$. Como vale $P(n)$, podemos reemplazar a_n por $2^n n!$. Luego, sabemos que $a_{n+1} = 2n(2^n n!) + 2^{n+1}n! = 2^{n+1}(nn! + n!) = 2^{n+1}n!(n+1) = 2^{n+1}(n+1)!$, que es lo que queríamos demostrar.

□

Ejercicio 2.4.4 (Series geométricas)

Sean $n \in \mathbb{N}$, y $a \in \mathbb{C}$, $a \neq 1$. Probar que

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

♦

Demostración. Vamos a probar esto mediante una serie de manipulaciones a esa ecuación, teniendo cuidado de que cada una sea un si-y-sólo-si.

$$\begin{aligned}
\sum_{i=0}^n a^i &= \frac{a^{n+1} - 1}{a - 1} \\
(a - 1) \sum_{i=0}^n a^i &= a^{n+1} - 1 \\
a \left(\sum_{i=0}^n a^i \right) - \sum_{i=0}^n a^i &= a^{n+1} - 1 \\
1 + \sum_{i=1}^{n+1} a^i - \sum_{i=0}^n a^i &= a^{n+1} \\
\sum_{i=0}^0 a^i + \sum_{i=1}^{n+1} a^i - \sum_{i=0}^n a^i &= a^{n+1} \\
\sum_{i=0}^{n+1} a^i - \sum_{i=0}^n a^i &= a^{n+1} \\
\sum_{i=n+1}^{n+1} a^i + \sum_{i=0}^n a^i - \sum_{i=0}^n a^i &= a^{n+1} \\
a^{n+1} &= a^{n+1}
\end{aligned}$$

Como llegamos a algo cierto a través de manipulaciones reversibles, cada paso puede ser revertido para empezar con $a^{n+1} = a^{n+1}$ y concluir con $\sum_{i=0}^n a^i = \frac{a^{n+1}-1}{a-1}$.

Notar que para revertir la multiplicación por $a - 1$ que hacemos para ir de la primer ecuación a la segunda, estamos usando que $a \neq 1$. \square

Ejercicio 2.4.5

Sea la sucesión $(a_n)_{n \in \mathbb{N}}$ definida como $a_0 = 1$, y para todo $n \in \mathbb{N}, n > 0$, $a_n = 4a_{n-1} - 2 \frac{(2n-2)!}{n!(n-1)!}$. Probar que para todo $n \in \mathbb{N}$, $a_n = \binom{2n}{n}$.

Demostración. Nuevamente, vamos a usar inducción, porque (a_n) tiene una estructura inductiva (es decir, está definida recursivamente).

Vamos a probar la proposición $P(n) : a_n = \binom{2n}{n}$, para todo $n \in \mathbb{N}$. Recordemos que $\binom{a}{b} = \frac{a!}{b!(a-b)!}$, para todo $a, b \in \mathbb{N}$.

Sea $n \in \mathbb{N}$.

1. Si $n = 0$, entonces queremos ver $P(0)$, que es $a_0 = \binom{0}{0} = \frac{0!}{0!0!} = \frac{1}{1} = 1$, y esto es cierto pues definimos $a_0 = 1$.
2. Si $n > 0$, entonces podemos usar la hipótesis inductiva en $n - 1$, y sabemos $P(n - 1)$. Esto nos dice que $a_{n-1} = \binom{2(n-1)}{n-1}$. Queremos ver $P(n + 1)$, es decir, que $a_n = \binom{2n}{n}$. Como sabemos por definición que $a_n = 4a_{n-1} - 2 \frac{(2n-2)!}{n!(n-1)!}$, podemos reemplazar lo que sabemos es a_{n-1} acá, y sabemos luego que $a_n = 4 \binom{2(n-1)}{n-1} - 2 \frac{(2n-2)!}{n!(n-1)!}$.

$$\begin{aligned}
a_n &= 4 \binom{2n-2}{n-1} - 2 \frac{(2n-2)!}{n!(n-1)!} \\
&= 4 \frac{(2n-2)!}{(n-1)!(n-1)!} - \frac{(2n-2)!}{n!(n-1)!} \\
&= \frac{(2n-2)!}{(n-1)!} \left(\frac{4}{(n-1)!} - \frac{2}{n!} \right) \\
&= \frac{(2n-2)!}{(n-1)!} \left(\frac{4n!}{n!(n-1)!} - \frac{2(n-1)!}{n!(n-1)!} \right) \\
&= \frac{(2n-2)!}{(n-1)!} \frac{4n(n-1)! - 2(n-1)!}{n!(n-1)!} \\
&= \frac{(2n-2)!}{(n-1)!} \frac{4n-2}{n!} \\
&= 2 \frac{(2n-2)!}{(n-1)!} \frac{2n-1}{n!} \\
&= 2 \frac{(2n-1)!}{n!(n-1)!} \\
&= 2 \binom{2n-1}{n} \\
&= \binom{2n-1}{n} + \binom{2n-1}{n} \\
&= \binom{2n-1}{n} + \binom{2n-1}{2n-1-n}, \text{ pues } \binom{a}{b} = \binom{a}{a-b} \\
&= \binom{2n-1}{n} + \binom{2n-1}{n-1} \\
&= \binom{2n}{n}, \text{ pues } \binom{a}{b} + \binom{a}{b-1} = \binom{a+1}{b}
\end{aligned}$$

que es lo que queríamos demostrar. □

4.1 Ejercicios

Ejercicio 2.4.6

Probar que para todo $n \in \mathbb{N}$, $\sum_{i=1}^n i = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$.

Ejercicio 2.4.7

Probar que para todo $n \in \mathbb{N}$, $1^3 + 2^3 + 3^3 + \dots + n^3$ es un cuadrado perfecto.

Ejercicio 2.4.8

Probar que para todo $n \in \mathbb{N}$, $\sum_{i=1}^n \frac{1}{(2i-1)(2i+1)} = \frac{n}{2n+1}$.

5 Combinatoria

Ejercicio 2.5.1

¿Cuántos números naturales hay menores o iguales que 1000 que no son ni múltiplos de 3 ni múltiplos de 5?



Demostración. Para esto vamos a usar el principio de inclusión-exclusión. Simbólicamente, este nos dice que para todo par de conjuntos A, B :

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Para resolver este ejercicio, usamos $A = \{n \in \mathbb{N} \mid n \leq 1000 \wedge n \not\equiv 0 \pmod{3}\}$, y $B = \{n \in \mathbb{N} \mid n \leq 1000 \wedge n \not\equiv 0 \pmod{5}\}$. Lo que nos pide el enunciado es $|A \cap B|$. Luego, esto es $|A| + |B| - |A \cup B|$.

Por definición, $|A \cup B| = \{n \in \mathbb{N} \mid n \leq 1000 \wedge (n \not\equiv 0 \pmod{3}) \vee n \not\equiv 0 \pmod{5}\}$. Pero si prestamos atención, el que algo sea o no múltiplo de tres o no múltiplo de cinco, es lo mismo que no ser múltiplo de quince. Podemos razonarlo o podemos hacer una tabla:

n	$n \pmod{3}$	$n \pmod{5}$	$n \pmod{15}$
0	0	0	0
1	1	1	1
2	2	2	2
3	0	3	3
4	1	4	4
5	2	0	5
6	0	1	6
7	1	2	7
8	2	3	8
9	0	4	9
10	1	0	10
11	2	1	11
12	0	2	12
13	1	3	13
14	2	4	14

Luego, $A \cup B = \{n \in \mathbb{N} \mid n \leq 1000 \wedge n \not\equiv 0 \pmod{15}\}$. Esto, a su vez, es $A \cup B = \{n \in \mathbb{N} \mid n \leq 1000\} \setminus \{n \in \mathbb{N} \mid n \leq 1000 \wedge n \equiv 0 \pmod{15}\}$. Es decir, sacarle los múltiplos de 15, al conjunto de todos los números menores o iguales a 1000. ¿Cuántos tales múltiplos hay? Uno de cada 15 números va a ser múltiplo de 15, luego hay $\lfloor \frac{1000}{15} \rfloor$ tales números, y $|A \cup B| = 1000 - \lfloor \frac{1000}{15} \rfloor = 1000 - 66 = 934$.

Encontrar $|A|$ y $|B|$ es similar. $|A| = 1000 - \lfloor \frac{1000}{3} \rfloor = 667$, y $|B| = 1000 - \lfloor \frac{1000}{5} \rfloor = 800$.

Luego, lo que nos piden es $|A \cap B| = 667 + 800 - 934 = 533$. □

Ejercicio 2.5.2

¿Cuántos palíndromes distintos de longitud n se pueden armar usando un conjunto de k símbolos?

Demostración. Si algo es un palíndrome, entonces se lee igual hacia adelante que hacia atrás. Sea $S = \{x_1, x_2, \dots, x_{n-1}, x_n\}$ un tal palíndrome. Entonces $x_n = x_1$, y $x_{n-1} = x_2$, etcétera. Tenemos que tener cuidado, entonces, con qué pasa en el medio, cuando no hay una igualdad extra. Por ejemplo, si $n = 3$, entonces hay sólo una igualdad, $x_1 = x_3$. La igualdad $x_2 = x_2$ no dice nada, entonces no restringe nuestras posibilidades.

Entonces, partamos en dos casos, dependiendo de si n es par o impar.

- Si $n \equiv 0 \pmod{2}$, entonces $n = 2t$ para algún $t \in \mathbb{N}$. Los primeros t símbolos son totalmente arbitrarios, entonces tenemos k^t posibles cadenas. Como los siguientes t símbolos están totalmente determinados por los primeros, no hay posibilidades restantes, y el número de palíndromos de longitud $n = 2t$ usando un conjunto de k símbolos es $k^{\frac{n}{2}}$.

x_1	x_2	\dots	x_t	$x_{t+1} = x_{n-(t+1)} = x_{t-1}$	$x_{t+2} = x_{n-(t+2)} = x_{t-2}$	\dots	$x_n = x_1$
-------	-------	---------	-------	-----------------------------------	-----------------------------------	---------	-------------

- Si $n \equiv 1 \pmod{2}$, entonces $n = 2t + 1$, para algún $t \in \mathbb{N}$. Los primeros $t + 1$ símbolos son arbitrarios, y tenemos k^{t+1} cadenas. Los últimos t símbolos están totalmente determinados por los primeros t , entonces no hay más posibilidades, y tenemos k^{t+1} palíndromos posibles.

x_1	x_2	\dots	x_t	x_{t+1}	$x_{t+2} = x_{n-(t+1)} = x_t$	\dots	$x_n = x_1$
-------	-------	---------	-------	-----------	-------------------------------	---------	-------------

Vemos entonces que la fórmula general para el número de palíndromos de longitud n usando un conjunto de k símbolos es $k^{\lceil \frac{n}{2} \rceil}$. \square

Ejercicio 2.5.3

Sin calcular los valores explícitamente ni expandir a factoriales, probar que

$$\binom{10}{4} = \binom{9}{3} + \binom{9}{4}$$

Demostración. Esto es un caso particular de la fórmula $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, con $n = 10, k = 4$.

Queremos dar una demostración de este hecho, sin expandir los coeficientes binomiales a sus factoriales. ¿Qué otra cosa sabemos sobre estos coeficientes? Que $\binom{n}{k}$ es el número de subconjuntos de tamaño k , de un conjunto de tamaño n .

Vamos a usar la técnica de «contar lo mismo de dos formas distintas». Supongamos que tenemos un conjunto X de tamaño n . Sea Y el conjunto de subconjuntos de X de tamaño k .

- Por un lado, $|Y| = \binom{n}{k}$, porque esa es precisamente la semántica de $\binom{n}{k}$.
- Por otro lado, sea $x \in X$. Los elementos de Y se dividen en los que contienen a x , y los que no contienen a x . ¿Cuántos hay que no tienen a x ? Eso es lo mismo que elegir subconjuntos de tamaño k de $X \setminus \{x\}$, que tiene tamaño $n - 1$. Luego, hay $\binom{n-1}{k}$ de esos. ¿Cuántos hay

que *sí* tienen a x ? Si esos ya tienen a x , el número de cosas que pueden elegir es $k - 1$ cosas más, pero no pueden volver a elegir a x , entonces tienen $n - 1$ elementos de X para elegir. Luego, hay $\binom{n-1}{k-1}$ de esos. Entonces, en total, $|Y| = \binom{n-1}{k} + \binom{n-1}{k-1}$.

Luego, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ para todo $n, k \in \mathbb{N}$, con $k \leq n$, en particular vale para $n = 10, k = 4$. □

Ejercicio 2.5.4 (Teorema binomial)

Probar que para todo $x, y \in \mathbb{R}$, y para todo $n \in \mathbb{N}$, tenemos que

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$$



Para esta demostración les voy a mostrar el proceso de deducción e ideas que hago antes de formalizarla.

Tengo una suma hasta n , quizás puedo descomponer la suma hasta n en una suma hasta $n - 1$? A ver....

Es más, tengo algo «a la» n , entonces puedo descomponer eso como $(x + y)(x + y)^{n-1}$. Si ahí uso la hipótesis inductiva, veamos qué queda...

$$\begin{aligned} (x + y)^{n+1} &= (x + y)(x + y)^n \\ &= (x + y) \sum_{k=0}^n \binom{n}{k} x^k y^{n-k} \\ &= \sum_{k=0}^n \binom{n}{k} x^{k+1} y^{n-k} + \sum_{k=0}^n \binom{n}{k} x^k y^{n-k+1} \end{aligned}$$

Quiero ver si puedo combinar estas dos ecuaciones, quizás emparejando coeficientes, porque así se sumarían los coeficientes binomiales. Pensando en cosas que valgan para sumas de coeficientes binomiales, recuerdo que $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, puede ser que emparejando los coeficientes que tengan el mismo $x^i y^j$, me queden así los coeficientes? A ver...

Tengo $i, j \in \mathbb{N}$, quiero ver qué coeficiente binomial multiplica a $x^i y^j$ en la primera sumatoria, y luego en la segunda. En la primera, $i = k + 1, j = n - k$. Entonces el coeficiente es $\binom{n-j+(i-1)}{k-i-1}$. En la segunda, tenemos $i = k, j = n - k + 1$, entonces el coeficiente es $\binom{j+i-1}{i}$.

Juntando estos dos, tenemos que la sumatoria es $\sum_{i,j} x^i y^j (\binom{j+i-1}{i-1} + \binom{j+i-1}{i})$, donde estoy sumando i, j sobre algún conjunto que no quiero pensar por ahora. Pero esto es bueno, los términos son precisamente de la forma que pensaba que eran, si los sumo me queda $\binom{j+i}{i}$.

Ahora tengo que pensar sobre dónde sumo los i, j . Todos los términos en las sumatorias tienen el mismo grado, son todos de grado $n + 1$. Los términos que estoy sumando ahora tienen grado $i + j$, entonces $i + j = n + 1$. Como quiero que me quede una sumatoria de sólo un índice, elijo i , y me queda $j = n + 1 - i$. ¿Cuáles son los bordes del índice i ? Hay un término de la sumatoria de la derecha donde tengo x^0 , y acá estoy diciendo x^i , así que seguro tengo que tener un término con $i = 0$. Por otro lado, de la sumatoria izquierda tengo un término x^{n+1} , y nuevamente acá digo x^i , así que i tiene que llegar hasta $n + 1$.

Entonces, usando que $j = n + 1 - i$ y el coeficiente del término $x^i y^j$ es $\binom{j+i}{i}$, la suma de las sumatorias me queda $\sum_{i=0}^{n+1} \binom{n+1-i+i}{i} x^i y^{n+1-i}$, y esto es igual a $\sum_{i=0}^{n+1} \binom{n+1}{i} x^i y^{n+1-i}$, que es precisamente lo que quiero probar.

OK, perfecto. ¡A formalizarlo! Voy a tener que pensar un rato para hacer menos grotesco el reindeindexado de las sumas.

Demostración. Vamos a usar inducción. Sea $P(n) : \forall x, y \in \mathbb{R}. (x+y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$.

- Caso base, $P(0)$. $P(0) : \forall x, y \in \mathbb{R}. (x+y)^0 = \sum_{i=0}^0 \binom{0}{k} x^k y^{0-k} = \binom{0}{0} x^0 y^0 = 1$. Como $(x+y)^0 = 1$ para todo x, y (sí, $0^0 = 1$), esto es cierto.
- Paso inductivo. Sea $n \in \mathbb{N}$. Asumo $P(n)$, quiero ver $P(n+1)$. Sean $x, y \in \mathbb{R}$.

$$\begin{aligned}
(x+y)^{n+1} &= (x+y)(x+y)^n \\
&= (x+y) \left(\sum_{k=0}^n \binom{n}{k} x^k y^{n-k} \right), \text{ por hipótesis inductiva} \\
&= x \left(\sum_{k=0}^n \binom{n}{k} x^k y^{n-k} \right) + y \left(\sum_{k=0}^n \binom{n}{k} x^k y^{n-k} \right) \\
&= \sum_{k=0}^n \binom{n}{k} x^{k+1} y^{n-k} + \sum_{k=0}^n \binom{n}{k} x^k y^{n-k+1} \\
&= \sum_{i=1}^{n+1} \binom{n}{i-1} x^i y^{n-(i-1)} + \sum_{k=0}^n \binom{n}{k} x^k y^{n-k+1}, \text{ llamando } i = k+1, \\
&= \sum_{j=1}^{n+1} \binom{n}{j-1} x^j y^{n+1-j} + \sum_{j=0}^n \binom{n}{j} x^j y^{n+1-j}, \text{ llamando a ambos índices } j
\end{aligned}$$

Acá hay que tener cuidado. Lo que queremos hacer es que los índices sean iguales, que es sacarle el último término ($j = n + 1$) a la primer sumatoria, y el primer término ($j = 0$) a la segunda. Lo único que sé es que $n \in \mathbb{N}$, entonces tengo al menos un término en cada sumatoria, porque $n + 1 \geq 1$ (para la primera) y $n \geq 0$ (para la segunda). No podría, si quisiera, sacar dos términos de cada una, porque no sé si *hay* dos términos en cada una.

$$\begin{aligned}
&= \binom{n}{n} x^{n+1} y^{n+1-(n+1)} + \sum_{j=1}^n \binom{n}{j-1} x^j y^{n+1-j} \\
&\quad + \sum_{j=1}^n \binom{n}{j} x^j y^{n+1-j} + \binom{n}{0} x^0 y^{n+1-0} \\
&= \binom{n}{n} x^{n+1} y^0 + \binom{n}{0} x^0 y^{n+1} + \sum_{j=1}^n \left(\binom{n}{j-1} + \binom{n}{j} \right) x^j y^{n+1-j} \\
&= \binom{n+1}{n+1} x^{n+1} y^0 + \binom{n+1}{0} x^0 y^{n+1} + \sum_{j=1}^n \binom{n+1}{j} x^j y^{n+1-j} \\
&= \sum_{j=0}^{n+1} \binom{n+1}{j} x^j y^{n+1-j}
\end{aligned}$$

que es lo que queríamos demostrar.

□

Ejercicio 2.5.5

Sean $f, g : \mathbb{N} \rightarrow \mathbb{N}$. Se define la convolución de f y g como la función $(f * g)(n) = \sum_{i=0}^n f(i)g(n-i)$.

Probar que la convolución es asociativa. Es decir, que para cuales quiera $f, g, h : \mathbb{N} \rightarrow \mathbb{N}$, tenemos $f * (g * h) = (f * g) * h$.

◆

Para este ejercicio también voy a escribirles en qué pienso a medida que lo hago.

Primero voy a expandir uno de los lados, a ver qué queda. La igualdad de funciones es igualdad en cada valor de entrada, así que sea $n \in \mathbb{N}$.

$$\begin{aligned}(f * (g * h))(n) &= \sum_{i=0}^n f(i)(g * h)(n-i) \\ &= \sum_{i=0}^n f(i) \left(\sum_{j=0}^{n-i} g(j)h(n-i-j) \right)\end{aligned}$$

Puedo que los índices tengan una relación simple, puedo en vez de ir desde $j = 0$ hasta $n - i$, ir desde i hasta n , y lo único que estoy haciendo es sumándole i a j . Luego cuando en el término digo $g(j)$, se convierte en $g(j-i)$, y cuando digo $h(n-i-j)$, se convierte en $h(n-i-(j-i)) = h(n-j)$.

$$\begin{aligned}&= \sum_{i=0}^n f(i) \left(\sum_{j=i}^n g(j-i)h(n-j) \right) \\ &= \sum_{0 \leq i \leq j \leq n} f(i)g(j-i)h(n-j)\end{aligned}$$

Lo de $f(i)g(j-i)$ se parece bastante a una convolución, específicamente $(f * g)(j)$. Tengo entonces que hacer que la suma de i esté afuera de la suma de j . Pero eso me quedaría $\sum_{j=i}^n f(i)g(j-i)$ adentro, que no es lo que quiero tener, porque no es una convolución. Más aún, no podría sacar el $h(n-j)$ afuera de esa convolución, porque es distinto para cada (j) término. OK, entonces la sumatoria interna tiene que ser sobre i .

$$\begin{aligned}&= \sum_{j=0}^n \sum_{i=0}^j f(i)g(j-i)h(n-j) \\ &= \sum_{j=0}^n (f * g)(j)h(n-j) \\ &= ((f * g) * h)(n)\end{aligned}$$

OK, pasémoslo en limpio.

Demostración. Sea $n \in \mathbb{N}$, y $f, g, h : \mathbb{N} \rightarrow \mathbb{N}$.

Entonces:

$$\begin{aligned}(f * (g * h))(n) &= \sum_{i=0}^n f(i)(g * h)(n - i) \\&= \sum_{i=0}^n f(i) \left(\sum_{j=0}^{n-i} g(j)h(n - i - j) \right) \\&= \sum_{i=0}^n f(i) \left(\sum_{j=i}^n g(j - i)h(n - j) \right) \\&= \sum_{0 \leq i \leq j \leq n} f(i)g(j - i)h(n - j) \\&= \sum_{j=0}^n \sum_{i=0}^j f(i)g(j - i)h(n - j) \\&= \sum_{j=0}^n (f * g)(j)h(n - j) \\&= ((f * g) * h)(n)\end{aligned}$$

□

6 Análisis asintótico

Muchas veces vamos a querer analizar el crecimiento de una función, cuando su entrada crece. Queremos por ejemplo tener una noción de que la función $g(n) = 100\log_2(n)$ es «más chica» que la función $f(n) = n^2$, a pesar de que $g(3) > f(3)$, por ejemplo. Lo que queremos captar es cómo estas funciones se comportan «en el límite», es decir, cuando n es muy grande.

ⓘ Nota

Nuestros algoritmos consumen recursos: tiempo, memoria, ancho de banda, energía.

Para analizar un algoritmo, modelamos el consumo de un recurso como una función $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ del tamaño de la entrada.

Pero esta función depende de detalles que queremos ignorar. Si medimos tiempo de ejecución, una computadora más rápida multiplicará todos los valores por alguna constante $c < 1$. Si contamos operaciones, la elección de qué operaciones contar cambia la función por un factor constante. Si hay costos fijos de inicialización o terminación, la función se desplaza por una constante aditiva.

Queremos una noción de equivalencia entre funciones que sea invarianta bajo estas transformaciones: que f y $c \cdot f + d$ sean «esencialmente iguales» para cualquier $c > 0$ y $d \in \mathbb{R}_{\geq 0}$. La notación asintótica nos da exactamente esto.

Para esto vamos a definir ciertos conjuntos de funciones.

Definición 12

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Definimos los conjuntos:

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \exists \alpha > 0 \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, (n \geq n_0 \Rightarrow g(n) \leq \alpha f(n))\}$$

$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \exists \alpha > 0 \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, (n \geq n_0 \Rightarrow g(n) \geq \alpha f(n))\}$$

$$\Theta(f) = O(f) \cap \Omega(f)$$



Podemos leer la expresión $f \in O(g)$ como « f está asintóticamente dominada por g », y la expresión $f \in \Omega(g)$ como « f asintóticamente domina a g », y la expresión $f \in \Theta(g)$ como « f y g son asintóticamente equivalentes».

⚠️ Advertencia

Esta notación no significa «aproximadamente» en el sentido coloquial. Es una relación matemática precisa entre funciones, aplicable a cualquier recurso que podamos modelar como función del tamaño de entrada.

Algo **muy** importante es que $O(f)$ y $\Omega(f)$ y $\Theta(f)$ son **conjuntos de funciones**. No son números, y no son funciones. Es increíblemente común que los alumnos se confundan con esto, especialmente cuando, como veremos más adelante, veamos notación como $5n^2 + O(n)$.

Veamos qué están captando estas definiciones, usando el ejemplo de arriba. Si $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, entonces $O(f)$ es el conjunto de funciones de la forma $g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, tales que existen constantes $\alpha \in \mathbb{R}$, $\alpha > 0$, y $n_0 \in \mathbb{N}$, tales que a partir de n_0 , $g(n) \leq \alpha f(n)$. Con $f(n) = n^2$ y $g(n) = 100\log_2(n)$, podemos tomar $\alpha = 100$, y $n_0 = 1$, y plantear:

$$\begin{aligned} g(n) &\leq \alpha f(n) \\ 100\log_2(n) &\leq 100n^2 \\ \log_2(n) &\leq n^2 \end{aligned}$$

Como sabemos que $\log_2(n) \leq n$ para todo $n \in \mathbb{N}$, $n \geq 1 = n_0$, y a su vez $n \leq n^2$ para todo $n \in \mathbb{N}$, esto es cierto. Luego, tenemos que $g \in O(f)$, g está asintóticamente dominada por f .

Veamos ahora una propiedad que relaciona dos de estos conjuntos.

Proposición 2.6.1

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ funciones. Entonces, $g \in O(f)$ si y sólo si $f \in \Omega(g)$.



Demostración.

$$\begin{aligned}
g \in O(f) &\Leftrightarrow \exists \alpha > 0 \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, (n \geq n_0 \Rightarrow g(n) \leq \alpha f(n)) \\
&\Leftrightarrow \exists \alpha > 0 \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, \left(n \geq n_0 \Rightarrow \frac{1}{\alpha} g(n) \leq f(n) \right) \\
&\Leftrightarrow \exists \alpha > 0 \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, \left(n \geq n_0 \Rightarrow f(n) \geq \frac{1}{\alpha} g(n) \right) \\
&\text{Que existe } \alpha > 0 \text{ es lo mismo que existe } \frac{1}{\alpha} > 0, \text{ luego} \\
&\Leftrightarrow \exists \alpha > 0 \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, (n \geq n_0 \Rightarrow f(n) \geq \alpha g(n)) \\
&f \in \Omega(g)
\end{aligned}$$

□

Vemos entonces que O y Ω son duales. Veamos un par de propiedades básicas sobre estos conjuntos.

Proposición 2.6.2

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, y $k \in \mathbb{R}_{>0}$. Entonces $kf \in \Theta(f)$.



Demostración. Debemos probar que $kf \in O(f)$ y $kf \in \Omega(f)$.

Para $kf \in O(f)$, debemos encontrar $\alpha > 0$ y $n_0 \in \mathbb{N}$ tales que para todo $n \geq n_0$, $(kf)(n) \leq \alpha f(n)$. Como $k > 0$, podemos tomar $\alpha = k$ y $n_0 = 0$. Entonces, para todo $n \geq n_0$, tenemos que $(kf)(n) = kf(n) \leq kf(n) = \alpha f(n)$.

Para $kf \in \Omega(f)$, debemos encontrar $\beta > 0$ y $n_1 \in \mathbb{N}$ tales que para todo $n \geq n_1$, $(kf)(n) \geq \beta f(n)$. Como $k > 0$, podemos tomar $\beta = k$ y $n_1 = 0$. Entonces, para todo $n \geq n_1$, tenemos que $(kf)(n) = kf(n) \geq kf(n) = \beta f(n)$.

Por lo tanto, $kf \in O(f) \cap \Omega(f) = \Theta(f)$.

□

Corolario 2.6.2.1

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Entonces $f \in O(f)$.



Proposición 2.6.3

Sean $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, tales que $f \in O(g)$ y $g \in O(h)$. Entonces $f \in O(h)$.



Demostración. Como $f \in O(g)$ y $g \in O(h)$, existen $\alpha_1, \alpha_2 > 0 \in \mathbb{R}$, y $n_1, n_2 \in \mathbb{N}$, tales que para todo $n \geq n_1$, tenemos que $f(n) \leq \alpha_1 g(n)$, y para todo $n \geq n_2$, tenemos que $g(n) \leq \alpha_2 h(n)$. Luego, para todo $n \geq \max(n_1, n_2)$, tenemos que $f(n) \leq \alpha_1 g(n) \leq \alpha_1 \alpha_2 h(n)$, lo que demuestra que $f \in O(h)$.

□

Nota

Esas dos propiedades nos dicen que tenemos un preorden entre funciones, definiendo $f \leq g \Leftrightarrow f \in O(g)$.

Tenemos una herramienta útil para probar pertenencia a estos conjuntos asintóticos, que es usar límites¹⁴.

Proposición 2.6.4

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, tal que g es positiva a partir de algún número n_0 . Sea $L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$, con $L \in \mathbb{R} \cup \{\infty\}$ (es decir, el límite existe). En el límite podemos asumir que $n \geq n_0$, para que la división tenga sentido. Entonces:

- $L < \infty \Leftrightarrow f \in O(g)$
- $L > 0 \Leftrightarrow f \in \Omega(g)$
- $0 < L < \infty \Leftrightarrow f \in \Theta(g)$



Demostración.

- $L < \infty \Rightarrow f \in O(g)$: Por definición de límite, existe $n_1 \in \mathbb{N}$ tal que para todo $n \geq n_1$, $\left| \frac{f(n)}{g(n)} - L \right| < 1$. Luego $\frac{f(n)}{g(n)} < L + 1$, y por lo tanto $f(n) < (L + 1)g(n)$ para todo $n \geq \max(n_0, n_1)$. Tomando $\alpha = L + 1$, tenemos $f \in O(g)$.
- $f \in O(g) \Rightarrow L < \infty$: Por definición de $O(g)$, existen $\alpha > 0$ y $n_1 \in \mathbb{N}$ tal que $f(n) \leq \alpha g(n)$ para todo $n \geq n_1$. Luego $\frac{f(n)}{g(n)} \leq \alpha$ para todo $n \geq \max(n_0, n_1)$. Como el límite L existe por hipótesis, y la sucesión $\frac{f(n)}{g(n)}$ está eventualmente acotada por α , tenemos $L \leq \alpha < \infty$.
- $L > 0 \Rightarrow f \in \Omega(g)$: Por definición de límite, tomando $\varepsilon = \frac{L}{2} > 0$, existe $n_1 \in \mathbb{N}$ tal que para todo $n \geq n_1$, $\left| \frac{f(n)}{g(n)} - L \right| < \frac{L}{2}$. Luego $\frac{f(n)}{g(n)} > L - \frac{L}{2} = \frac{L}{2}$, y por lo tanto $f(n) > (\frac{L}{2})g(n)$ para todo $n \geq \max(n_0, n_1)$. Tomando $\alpha = \frac{L}{2}$, tenemos $f \in \Omega(g)$.
- $f \in \Omega(g) \Rightarrow L > 0$: Por definición de $\Omega(g)$, existen $\alpha > 0$ y $n_1 \in \mathbb{N}$ tal que $f(n) \geq \alpha g(n)$ para todo $n \geq n_1$. Luego $\frac{f(n)}{g(n)} \geq \alpha$ para todo $n \geq \max(n_0, n_1)$. Como el límite L existe por hipótesis, y la sucesión $\frac{f(n)}{g(n)}$ está eventualmente acotada inferiormente por $\alpha > 0$, tenemos $L \geq \alpha > 0$.
- $0 < L < \infty \Leftrightarrow f \in \Theta(g)$: Se sigue de los dos puntos anteriores, pues $\Theta(g) = O(g) \cap \Omega(g)$.



Veamos cómo usar esta propiedad de límites.

Ejercicio 2.6.5

Sean $f(n) = 7\log_2(n)$, $g = \sqrt{n} + 1$. Entonces $f \in O(g)$.



¹⁴La restricción de que L exista no es realmente necesaria. Las equivalencias valen si uno toma \limsup para O y \liminf para Ω , que existen aún cuando el límite tradicional no existe. Como no quiero asumir que vieron límites superiores e inferiores, uso esta formulación más restringida, que es suficiente en la práctica.

Demostración. Como $\sqrt{n} + 1 > 0$ para todo $n \in \mathbb{N}$, entonces podemos tomar =

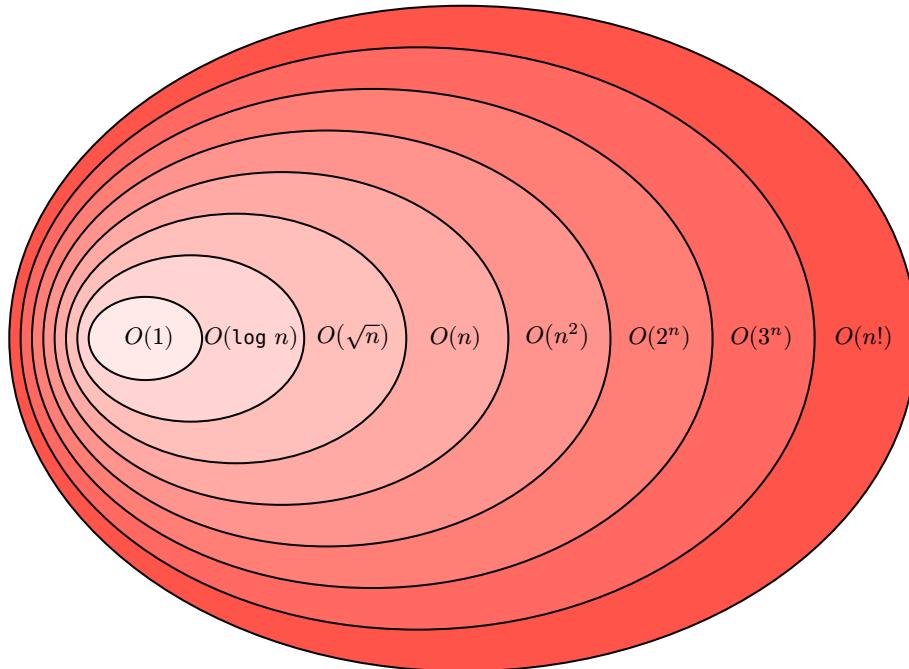
$$\begin{aligned} L &= \lim_{n \rightarrow \infty} \frac{7\log_2(n)}{\sqrt{n} + 1} \\ &= \lim_{n \rightarrow \infty} \frac{7 \frac{\log n}{\log 2}}{\sqrt{n} + 1} \\ &= \lim_{n \rightarrow \infty} \left(\frac{7}{\log 2} \right) \frac{\log n}{\sqrt{n} + 1} \end{aligned}$$

Usando la regla de L'Hôpital, obtenemos

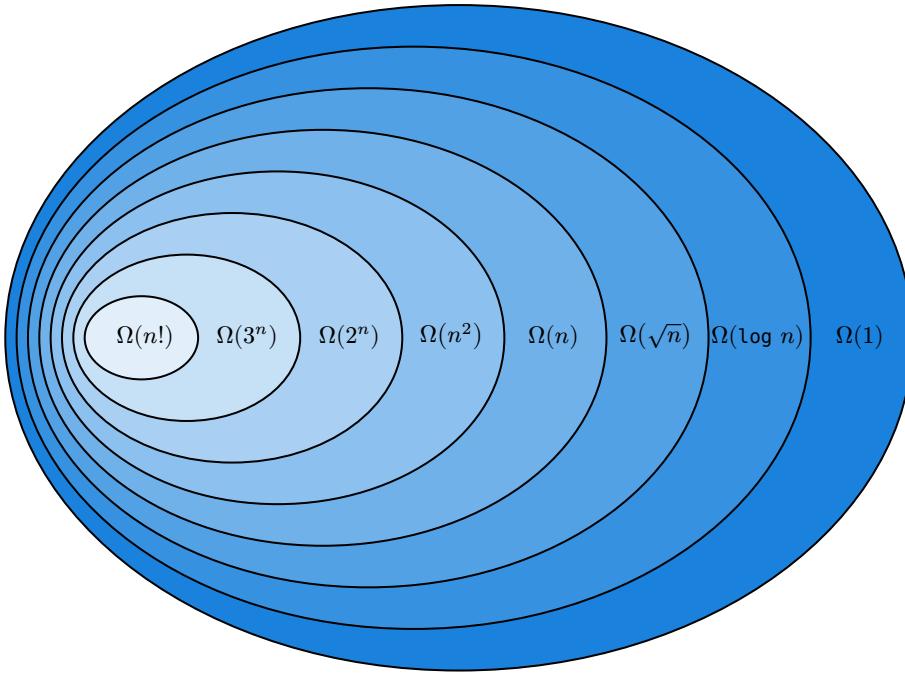
$$\begin{aligned} L &= \left(\frac{7}{\log 2} \right) \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} \\ &= \left(\frac{7}{\log 2} \right) \lim_{n \rightarrow \infty} \frac{2\sqrt{n}}{n} \\ &= \left(\frac{7}{\log 2} \right) \lim_{n \rightarrow \infty} \frac{2}{\sqrt{n}} \\ &= \left(\frac{7}{\log 2} \right) 0 \\ &= 0 \end{aligned}$$

Por lo tanto, $f \in O(g)$. □

De hecho, tendremos las siguientes inclusiones.



Vemos, entonces, que estar en $O(n^2)$ implica estar en $O(n^6)$, y que estar en $O(3^n \log n)$, implica estar en $O(4^n \sqrt{n})$. Tendremos las inclusiones opuestas cuando usamos cotas inferiores, usando Ω :



Todas las funciones están acotadas por debajo por alguna constante (por ejemplo, 0), pero muy pocas están acotadas por debajo, eventualmente, por un múltiplo de $n!$.

En las sección de ejercicios van a tener que demostrar varias de esas inclusiones. La siguiente propiedad nos va a dejar sumar estos conjuntos.

Proposición 2.6.6

Sean $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, con $f \in O(h)$ y $g \in O(h)$. Entonces $f + g \in O(h)$.



Demostración. Por hipótesis, $f \in O(h)$ implica que existen $\alpha > 0$ y $n_1 \in \mathbb{N}$ tal que $f(n) \leq \alpha h(n)$ para todo $n \geq n_1$. Similarmente, $g \in O(h)$ implica que existen $\beta > 0$ y $n_2 \in \mathbb{N}$ tal que $g(n) \leq \beta h(n)$ para todo $n \geq n_2$.

Sea $n_0 = \max(n_1, n_2)$ y sea $\gamma = \alpha + \beta$. Entonces para todo $n \geq n_0$ tenemos que

$$\begin{aligned} f(n) + g(n) &\leq \alpha h(n) + \beta h(n) \\ &= (\alpha + \beta)h(n) \\ &= \gamma h(n) \end{aligned}$$

Luego, $f(n) + g(n) \leq \gamma h(n)$ para todo $n \geq n_0$, lo que demuestra que $f + g \in O(h)$.



La siguiente propiedad nos deja quedarnos con los términos que crecen más rápidamente, en una suma.

Proposición 2.6.7

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, con $f \in O(g)$. Entonces $f + g \in \Theta(g)$.



Demostración. Para probar que $f + g \in \Theta(g)$, debemos probar que $f + g \in O(g)$ y $f + g \in \Omega(g)$.

- $f + g \in O(g)$: Por hipótesis, $f \in O(g)$ implica que existen $\alpha > 0$ y $n_0 \in \mathbb{N}$ tal que $f(n) \leq \alpha g(n)$ para todo $n \geq n_0$. Luego, para todo $n \geq n_0$:

$$\begin{aligned} f(n) + g(n) &\leq \alpha g(n) + g(n) \\ &= (\alpha + 1)g(n) \end{aligned}$$

Por lo tanto, $f + g \in O(g)$.

- $f + g \in \Omega(g)$: Como $f(n) \geq 0$ para todo $n \in \mathbb{N}$, tenemos que $f(n) + g(n) \geq g(n)$ para todo $n \in \mathbb{N}$. Tomando $\beta = 1$, tenemos que $f(n) + g(n) \geq \beta g(n)$ para todo $n \in \mathbb{N}$. Por lo tanto, $f + g \in \Omega(g)$.

Como $f + g \in O(g)$ y $f + g \in \Omega(g)$, concluimos que $f + g \in \Theta(g)$. □

Luego, en una función como $h(n) = n^2 + 3n$, como $3n \in O(n^2)$ pues $\lim_{n \rightarrow \infty} \frac{3n}{n^2} = 0$, tenemos que $h \in \Theta(n^2)$. Las siguientes dos propiedades nos dicen cómo multiplicar dos de estos conjuntos, o multiplicarlos por un escalar.

Proposición 2.6.8

Sean $f, F, g, G : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Si $f \in O(F)$ y $g \in O(G)$, entonces $fg \in O(FG)$.

La demostración es el Ejercicio 2.6.14.

Proposición 2.6.9

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, y $\alpha > 0 \in \mathbb{R}$. Entonces $f \in O(g)$ si y sólo si $\alpha f \in O(g)$.

La demostración es el Ejercicio 2.6.13.

Veamos un ejemplo en la práctica.

Ejercicio 2.6.10

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ la función dada por $f(n) = 3n^2 + 2n + 1$. Probar que $f \in O(n^2)$.

Demostración. Primero hagamos las cuentas «a mano». Tenemos que elegir un $\alpha \in \mathbb{R}$, $\alpha > 0$, y un $n_0 \in \mathbb{N}$, tal que para todo $n \in \mathbb{N}$, $n \geq n_0$, se cumpla que $f(n) \leq \alpha n^2$. Intuitivamente, f crece parecido a $3n^2$. Luego, si elegimos $\alpha = 4$, eventualmente $f(n) \leq \alpha n^2$. Probemos esto formalmente.

Elegimos $\alpha = 4$. Queremos encontrar n_0 tal que si $n \geq n_0$, entonces $3n^2 + 2n + 1 \leq 4n^2$.

$$\begin{aligned} 3n^2 + 2n + 1 &\leq 4n^2 \\ \Leftrightarrow 2n + 1 &\leq n^2 \\ \Leftrightarrow 0 &\leq n^2 - 2n - 1 \end{aligned}$$

Ahora consideremos la función $h(n) = n^2 - 2n - 1$. Para ver dónde es no-negativa, podemos encontrar sus raíces:

$$\begin{aligned} n^2 - 2n - 1 &= 0 \\ n &= \frac{2 \pm \sqrt{4 + 4}}{2} \\ n &= 1 \pm \sqrt{2} \end{aligned}$$

Luego, como h es una parábola que crece en sus extremos, cuando $n \geq 1 + \sqrt{2}$, $h(n) \geq 0$.

Como $\sqrt{2} < 2$, podemos elegir $n_0 = 3$, y garantizamos que si $n \geq n_0$, entonces $h(n) \geq 0$, que vimos es equivalente a que $f(n) \leq 4n^2$. Luego, $f \in O(n^2)$. \square

Demostración. Ahora usemos las herramientas que aprendimos. $f(n) = g(n) + h(n)$, con $g(n) = 3n^2$, y $h(n) = 2n + 1$. Como $h \in O(g)$ pues $\lim_{n \rightarrow \infty} \frac{3n^2}{2n+1} = \infty > 0$, entonces por la Proposición 2.6.7, $f \in \Theta(g)$. Por la Proposición 2.6.2, sabemos que $g \in \Theta(n^2)$.

Finalmente, por la Proposición 2.6.3, tenemos que $f \in \Theta(n^2)$. En particular, $f \in O(n^2)$. \square

Demostración. Finalmente usemos sólo la propiedad de límites. Sea $L = \lim_{n \rightarrow \infty} \frac{3n^2 + 2n + 1}{n^2} = \lim_{n \rightarrow \infty} 3 + \frac{2}{n} + \frac{1}{n^2} = 3$. Entonces por la Proposición 2.6.4, tenemos que $3n^2 + 2n + 1 \in \Theta(n^2) \subseteq O(n^2)$. \square

Podemos generalizar este hecho.

Proposición 2.6.11

Sea $n \in \mathbb{N}$, y $f \in \mathbb{N}[x]$ un polinomio de grado n en la variable x , con coeficientes naturales. Entonces $f \in \Theta(x^n)$.

La demostración es el Ejercicio 2.6.15.

Una propiedad que vamos a usar mucho es que $\Theta(\log_a n) = \Theta(\log_b n)$ para todos $a, b \in \mathbb{N}_{>1}$. Esto nos va a dejar escribir $\Theta(\log n)$, sin especificar la base del logaritmo, pero sin perder precisión.

Proposición 2.6.12

Sean $a, b \in \mathbb{N}_{>1}$. Entonces $\log_a x \in \Theta(\log_b x)$.

Demostración. Sea $L = \lim_{x \rightarrow \infty} \frac{\log_a x}{\log_b x}$. Entonces:

$$\begin{aligned}
L &= \lim_{x \rightarrow \infty} \frac{\log_a x}{\log_b x} \\
&= \lim_{x \rightarrow \infty} \frac{\frac{\ln x}{\ln a}}{\frac{\ln x}{\ln b}} \\
&= \lim_{x \rightarrow \infty} \frac{\ln x}{\ln a} \cdot \frac{\ln b}{\ln x} \\
&= \lim_{x \rightarrow \infty} \frac{\ln b}{\ln a} \\
&= \frac{\ln b}{\ln a} \in \mathbb{R}_{>0}
\end{aligned}$$

Por lo tanto, $\log_a x \in \Theta(\log_b x)$. □

6.1 Ejercicios

Ejercicio 2.6.13

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, y $\alpha > 0 \in \mathbb{R}$. Entonces $f \in O(g)$ si y sólo si $\alpha f \in O(g)$.

Ejercicio 2.6.14

Sean $f, F, g, G : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Si $f \in O(F)$ y $g \in O(G)$, entonces $fg \in O(FG)$.

Ejercicio 2.6.15

Sea $n \in \mathbb{N}$, y $f \in \mathbb{N}[x]$ un polinomio de grado n en la variable x , con coeficientes naturales. Entonces $f \in \Theta(x^n)$.

Ejercicio 2.6.16

Probar que si $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ es tal que $f(n) = 4^n$, entonces $f \notin O(2^n)$.

Ejercicio 2.6.17

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ funciones. Recordando que $f \cdot O(g) = \{f \cdot h \mid h \in O(g)\}$, probar que $f \cdot O(g) = O(f \cdot g)$.

Ejercicio 2.6.18

Sean $f, g : \mathbb{N} \rightarrow \mathbb{N}$ funciones. Probar que si $f \in \Theta(g)$, entonces $g \in \Theta(f)$.

Ejercicio 2.6.19

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ funciones, y $n_0 \in \mathbb{N}$. Probar que si $f(n) \leq g(n)$ para todo $n \geq n_0$, entonces $f \in O(g)$.

Mostrar que la vuelta no vale, exhibiendo un ejemplo explícito de f y g , tal que $f \in O(g)$, pero no existe ningún n_0 tal que para todo $n \geq n_0$ se tenga $f(n) \leq g(n)$.



Ejercicio 2.6.20

Sea $n \in \mathbb{N}$, y $f \in \mathbb{Z}[x]$ un polinomio de grado n . Probar que $f \in O(x^n)$. Mostrar que no es cierto que $f \in \Omega(x^n)$, dando un ejemplo explícito de n y f .

Este ejercicio es distinto a Ejercicio 2.6.15, pues los coeficientes son enteros, no sólo naturales.



Ejercicio 2.6.21

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ tales que $f \in \Theta(g)$. Probar que $\log_2 f \in \Theta(\log g)$.

Probar que **no** es cierto, en general, que si $\log_2 f \in \Theta(\log g)$, entonces $f \in \Theta(g)$.



Ejercicio 2.6.22

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Es cierto que o bien $f \in O(g)$, o bien $g \in O(f)$? De ser cierto, demostrarlo. De caso contrario, exhibir un contraejemplo, y demostrar que efectivamente es un contraejemplo.



7 Álgebra asintótica

Vimos como notaciones como $O(\dots)$ y $\Omega(\dots)$ se pueden usar para aproximar el crecimiento de funciones. Vamos a querer expresar que no sólo una función se comporta de esa manera, sino que «una parte» de una función se comporta de esa manera. Por ejemplo, al decir que $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, $f(n) = n^2 + O(n)$, queremos decir que existe una función $g \in O(n)$ tal que $f(n) = n^2 + g(n)$ para todo $n \in \mathbb{N}$. Esto nos dice algo más que sólo saber que $f \in O(n^2)$, nos dice algo sobre la estructura de f . Al mismo tiempo, no nos dice exactamente *quién* es f , lo cual es útil pues a veces vamos a querer obviar exactamente qué elemento de $O(n)$ es g .

Definición 13

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, y $k \in \mathbb{R}_{\geq 0}$. Definimos:

- $f + O(g) = \{f + h \mid h \in O(g)\}$
- $fO(g) = \{fh \mid h \in O(g)\}$
- $kO(f) = \{kh \mid h \in O(f)\}$

Definiciones equivalentes valen para Ω y Θ .



⚠️ Advertencia

Tengan cuidado, ¡decir que $f(n) = n^2 + O(n)$ **no** define f ! Sólo nos dice que f está en el conjunto $\{n^2 + g \mid g \in O(n)\}$. Más propiamente escrito, esto sería $f \in n^2 + O(n)$. Les presento

esta notación porque es usual verla, pero deben tener cuidado, pues cosas «obvias» como dar vuelta esta «igualdad» dejan de funcionar. No tiene sentido decir que $n^2 + O(n) = f$, como tampoco va a tener sentido razonamientos del estilo «Como $f = n^2 + O(n)$ y $g = n^2 + O(n)$, entonces $f = g$.» .

Esta notación es tan poderosa como peligrosa. ¡Están advertidos!



Veamos algunos ejemplos.

Ejercicio 2.7.1

Sea $f(n) = 2n^2 + O(n)$, y $g(n) = 4n + \Theta(\log n)$. Demostrar que $fg \in O(n^3)$.

Demostración. Por definición, existen funciones $h_1 \in O(n)$ y $h_2 \in \Theta(\log n)$ tales que $f(n) = 2n^2 + h_1(n)$ y $g(n) = 4n + h_2(n)$.

Luego:

$$\begin{aligned} f(n)g(n) &= (2n^2 + h_1(n))(4n + h_2(n)) \\ &= 8n^3 + 2n^2h_2(n) + 4nh_1(n) + h_1(n)h_2(n) \end{aligned}$$

Ahora debemos probar que cada término está en $O(n^3)$:

- $8n^3 \in O(n^3)$ por la Proposición 2.6.9.
- Como $h_1 \in O(n)$, entonces $4nh_1 \in O(n \cdot n) = O(n^2) \subseteq O(n^3)$ por el Ejercicio 2.6.14.
- Como $h_2 \in \Theta(\log n) \subseteq O(\log n)$, entonces $2n^2h_2 \in O(n^2 \log n) \subseteq O(n^3)$ por el Ejercicio 2.6.14, pues $\log n \in O(n)$.

- Como $h_1 \in O(n)$ y $h_2 \in O(\log n)$, entonces $h_1 h_2 \in O(n \log n) \subseteq O(n^2) \subseteq O(n^3)$ por el Ejercicio 2.6.14.

Por lo tanto, $fg \in O(n^3)$. □

7.1 Errores frecuentes

Es muy común que se confundan con esta notación. A continuación les voy a dar algunas demostraciones incorrectas. Presten mucha atención, intenten ver dónde exactamente les estoy mintiendo.

Proposición falsa 2.7.2

Sea $g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, con $g(n) = 2^n$. Entonces $g \in O(1)$.



Demostración incorrecta. Es fácil ver que $O(1) \cdot O(1) = O(1)$. Luego, por inducción, tendremos $\prod_{i=1}^n O(1) = O(1)$.

Sea $f : \mathbb{N} \rightarrow \mathbb{N}$, $f(m) = 2$ para todo $m \in \mathbb{N}$. Claramente $f \in O(1)$. Luego, $(\prod_{i=1}^n f) \in \prod_{i=1}^n O(1) = O(1)$. Asimismo, si llamamos $g(n) = \prod_{i=1}^n f(n)$, tendremos que $g(n) = \prod_{i=1}^n 2 = 2^n$. Por lo tanto, $2^n \in O(1)$. □

Esto es obviamente incorrecto. Es cierto que $O(1) \cdot O(1) = O(1)$. Y también es cierto que para todo $k \in \mathbb{N}$ fijo, tenemos que $\prod_{i=1}^k O(1) = O(1)$. Recordemos qué significa que $h \in O(1)$: Existen constantes $\alpha \in \mathbb{R}_{>0}$, $n_0 \in \mathbb{N}$, tal que $h(n) \leq \alpha$ para todo $n \geq n_0$. Para cada $k \in \mathbb{N}$, vamos a tener dos tales constantes. Lo que *no* vamos a tener, son dos constantes que valgan para *todo* k al mismo tiempo.

Lo que la inducción nos da es, para cada $k \in \mathbb{N}$, una constante α_k tal que el producto de k funciones acotadas por 2, está acotado por $\alpha_k 2^k$. Si k es una constante, esto es realmente una inclusión en el conjunto $O(1)$. Pero si tomamos a n como variable, el que exista una constante α tal que podemos acotar al producto de las n funciones por $\alpha 2^n$ no implica que este producto está en $O(1)$, pues la cota resultante *es una función de n*, no una constante. Veamos otra demostración que tiene el mismo error, y esta parecería ser «obvia».

Proposición 2.7.3

Sea $T(n) = \sum_{i=1}^n (i + O(1))$. Entonces $T \in O(n^2)$.



Demostración incorrecta. Por definición, para cada i en $[1, \dots, n]$, existe una función $h_i \in O(1)$ tal que el i -ésimo término de la suma es $i + h_i(n)$.

Luego:

$$\begin{aligned} T(n) &= \sum_{i=1}^n i + h_i(n) \\ &= \sum_{i=1}^n i + \sum_{i=1}^n h_i(n) \end{aligned}$$

Para el primer término, sabemos que $\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{n^2+n}{2} \in \Theta(n^2) \subseteq O(n^2)$.

Para el segundo término, como cada $h_i \in O(1)$, existen $\alpha_i > 0$ y $n_i \in \mathbb{N}$ tales que para todo i , para todo $n \geq n_i$, $h_i(n) \leq \alpha_i$. Sea $\alpha = \max\{\alpha_1, \alpha_2, \dots, \alpha_n\}$ y $m_0 = \max\{n_1, n_2, \dots, n_n\}$.

Entonces, para todo $n \geq m_0$:

$$\sum_{i=1}^n h_i(n) \leq \sum_{i=1}^n \alpha = n\alpha \in O(n) \subseteq O(n^2)$$

Por lo tanto, $T \in O(n^2)$. □

Este razonamiento es incorrecto. Para verlo, basta tomar $h_i(n) = i^3$ para cada $i \in [1, \dots, n]$, una función constante (no depende de su argumento), y por lo tanto $h_i \in O(1)$. Entonces $T(n) = \sum_{i=1}^n i + \sum_{i=1}^n h_i(n) = \frac{n(n+1)}{2} + \sum_{i=1}^n i^3 = \frac{n(n+1)}{2} + \left(\frac{n(n+1)}{2}\right)^2 \in \Omega(n^4)$, que tiene intersección nula con $O(n^2)$. El error en la demostración es que m_0 y α dependen de n (son un máximo de n cosas). La definición de $O(\dots)$ requiere que sean *constantes*.

Esto nos muestra una sutileza sobre la notación asintótica. Al usar la expresión « $O(\dots)$ » en un contexto como el de esa sumatoria, donde hay dos variables libres (i y n) en vez de sólo una, no está claro con respecto a cuál variable estamos diciendo que varía nuestra función. A priori, en un término así podemos usar ambas, teniendo una función de varias variables. Más adelante veremos notación asintótica con múltiples variables. Por ahora, para probar lo que queremos que valga, vamos a pedir que para todo n , y vamos a querer que $O(1)$ en esa sumatoria signifique una función $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, donde la sumatoria es de la forma $T(n) = \sum_{i=1}^n i + g(i, n)$, y existe una constante $\alpha \in \mathbb{R}$ tal que para todo $n \in \mathbb{N}$, y para todo $1 \leq i \leq n$, $g(i, n) \leq \alpha$. Esto nos va a permitir la siguiente demostración.

Demostración. Sea $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, tal que existe $\alpha \in \mathbb{R}$ tales que para todo $n \in \mathbb{N}$ y para todo $1 \leq i \leq n$, $g(i, n) \leq \alpha$. Luego:

$$\begin{aligned} T(n) &= \sum_{i=1}^n i + g(i, n) \\ &= \sum_{i=1}^n i + \sum_{i=1}^n g(i, n) \\ &\leq \sum_{i=1}^n i + \sum_{i=1}^n \alpha \\ &= \frac{n(n+1)}{2} + \alpha n \end{aligned}$$

Por lo tanto, $T \in O(n^2)$. □

Proposición falsa 2.7.4

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida por:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ T(n - 1) + 1 & \text{si } n > 0 \end{cases}$$

Entonces $T \in O(1)$.



Demostración incorrecta. Probemos esto por inducción en n .

- Caso base. $T(0)$ es una constante, luego está en $O(1)$.
- Paso inductivo, $n > 0$. Asumimos que $T(k) \in O(1)$ para todo $k < n$, queremos ver que $T(n) \in O(1)$. Como $n > 0$, $T(n) = T(n - 1) + 1$. Como $n - 1 < n$, por hipótesis inductiva $T(n - 1) \in O(1)$. Luego, como $O(1) + 1 = O(1)$, tenemos que $T(n) \in O(1)$.

Asimismo, vemos que $T(n) = n$, y por lo tanto, $n \in O(1)$.



Esto es un sinsentido. Empezamos diciendo sinsentidos al decir « $T(0)$ es una constante, luego está en $O(1)$ ». Recordemos, **$O(1)$ es un conjunto de funciones**. Mientras tanto, $T(0)$ es un número, es cero. De ninguna manera vamos a tener que $T(0)$ está en $O(1)$, es como decir que un elefante está en un conjunto de jirafas: no tipa.



Luego seguimos con la confusión al decir $T(k) \in O(1)$. Nada de esto tiene sentido, y no tiene sentido hablar de « $T(n) \in O(1)$ ». He visto alumnos que se confunden entre f , una función, y $f(x)$, el resultado de evaluar una función en un punto, x . Hasta algunos libros fomentan esa confusión. En este caso, esa confusión nos dejó decir sinsentidos.

Veamos cómo podríamos demostrar algo sobre esta función.

Proposición 2.7.5

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida por:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ T(n - 1) + 1 & \text{si } n > 0 \end{cases}$$

Entonces $T \in O(n)$.



Demostración. Vamos a mostrar que existen constantes $\alpha \in \mathbb{R}_{>0}$, $n_0 \in \mathbb{N}$, tal que para todo $n \in \mathbb{N}$, $(n \geq n_0) \Rightarrow T(n) \leq \alpha n$. En particular, vamos a probar por inducción que $n_0 = 0$, $\alpha = 1$ funcionan.

Formalmente, sea $P(k) : (k \geq n_0) \Rightarrow T(k) \leq k$.

- Caso base, $P(0)$. Tenemos que probar que $(0 \geq n_0) \Rightarrow T(0) \leq 0$. Esto es cierto porque la conclusión es cierta, $0 = T(0) \leq 0 = 0$.
- Paso inductivo. Tenemos $k \in \mathbb{N}$. Asumimos $P(k)$, queremos probar $P(k+1)$. Luego, queremos probar que $(k+1 \geq n_0) \Rightarrow T(k+1) \leq (k+1)$. Para probar una implicación, podemos asumir la premisa, y luego $k+1 \geq n_0$. Como $k+1 \geq n_0$, con más razón $k \geq n_0$. Ahora expandimos $T(k+1) = T(k) + 1$. Por hipótesis inductiva, como $k \geq n_0$, tenemos que $T(k) \leq k$. Juntando, tenemos que $T(k+1) \leq k+1$, que es lo que queríamos demostrar.

Luego, $T(k) \leq k$ para todo $k \in \mathbb{N}$. Esto es precisamente la definición de que $T \in O(n)$, usando $n_0 = 0$, $\alpha = 1$. □

Proposición falsa 2.7.6

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ una función. Entonces $f \in O(1)$. ♥

Demostración incorrecta. Sea $g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, $g(n) = f(n) + n$. Como $f(n) \geq 0$ para todo $n \in \mathbb{N}$, tenemos que $g(n) \geq n$ para todo $n \in \mathbb{N}$. Por lo tanto, $g \in \Omega(n)$, con lo cual $n \in O(g)$. Ahora bien, $f(n) = g(n) - n$, y luego $f \in O(g) - O(g) = O(g - g) = O(0) \subseteq O(1)$. Por lo tanto, $f \in O(1)$. □

La resta de conjuntos asintóticos no funciona así. El que algo esté en $O(g)$ no significa que sea g . El que $n \in O(g)$ no significa que al restar $f - n$ estamos restando $f - g$, y poner $O(\dots)$ al rededor no lo hace cierto. Notemos acá cómo usamos $n \in O(g)$ para querer decir que si $h(n) = n$, entonces $h \in O(g)$. Recordemos que $O(\dots)$ es un conjunto de funciones, no de números, ni variables.

Por último, veamos una propiedad que a veces quieren que valga.

Proposición falsa 2.7.7

Sean $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Si $f \in \Theta(g)$, entonces $h \circ f \in \Theta(h \circ g)$. ♥

Esto no es cierto, y basta tomar $f(n) = 2n$, $g(n) = n$, y $h(n) = 2^n$. f y g son asintóticamente equivalentes. Sin embargo $(h \circ f)(n) = 2^{2n} = (2^n)^2$ no es asintóticamente equivalente a $(h \circ g)(n) = 2^n$.

7.2 Ejercicios

Ejercicio 2.7.8

Sea $k \in \mathbb{N}$ una constante fija. Probar que si $f_1, f_2, \dots, f_k \in O(1)$, entonces $\prod_{i=1}^k f_i \in O(1)$.

Explicar por qué este resultado **no** implica que $2^n \in O(1)$. ♣

Ejercicio 2.7.9

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ definida por $T(n) = \sum_{i=1}^n i^2$. Probar que $T \in \Theta(n^3)$. ♣

Ejercicio 2.7.10

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ tal que $f(n) = n + (-1)^n n$. Determinar si $f \in O(n)$, $f \in \Omega(n)$, y/o $f \in \Theta(n)$. Demostrar las tres cosas.

Ejercicio 2.7.11

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ tales que $f \in \Theta(g)$. Probar que $2^f \notin \Theta(2^g)$ en general, dando un contraejemplo explícito.

Ejercicio 2.7.12

Sean $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ con $g(n) \geq 1$ para todo n . Probar que si $f \in O(g)$ y $h \in O(g)$, entonces $f - h \in O(g)$.

Mostrar que **no** es cierto que $f - h \in O(g - g) = O(0)$.

Ejercicio 2.7.13

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ definida por $f(n) = \lfloor \log_2(n+1) \rfloor$. Probar que $f \in \Theta(\log n)$.

Ejercicio 2.7.14

Sea $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ tal que $f(n) = 3n^2 + O(n)$. Probar que $f^2 \in 9n^4 + O(n^3)$.

8 Recurrencias

Muchas veces vamos a tener funciones que están definidas de forma recursiva, también llamadas recurrencias. Es decir, son funciones de la forma $f(n) = \dots f(k) \dots$, con $k < n$. Por ejemplo, los números de Fibonacci están definidos como:

$$f(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ f(n-1) + f(n-2) & \text{si } n > 2 \end{cases}$$

También vamos a encontrarnos con recurrencias definidas sólo hasta conjuntos asintóticos, como por ejemplo:

$$T(n) = \begin{cases} 5 & \text{si } n = 1 \\ T(n-1) + O(n^2) & \text{si } n > 1 \end{cases}$$

Vamos a querer hacer análisis asintótico de estas funciones, y frecuentemente esto comienza encontrando una forma cerrada para la recurrencia. En este capítulo veremos tres técnicas para hacer esto: expansión directa, árboles de recursión, y finalmente el teorema maestro.

8.1 Expansión

La técnica de **expansión** consiste en aplicar repetidamente la definición recursiva de una función T , sustituyendo $T(n-1)$, $T(n-2)$, etc., hasta llegar a un caso base. Esto nos permite ver el patrón resultante y conjeturar una forma cerrada.

El procedimiento es:

1. Expandir $T(n)$ usando la definición recursiva, hasta reconocer un patrón o llegar al caso base.

2. Expresar la suma o producto resultante en forma cerrada.
3. Verificar la solución por inducción.

Esta técnica funciona particularmente bien para **recurrencias lineales**, donde $T(n)$ depende de $T(n - 1)$, $T(n - 2)$, etc. Para recurrencias que dividen el argumento (como $T(\frac{n}{2})$), la expansión se vuelve más complicada, y usaremos otras técnicas.

Comencemos con un ejemplo sencillo, una suma aritmética.

Ejercicio 2.8.1

Sea $T : \mathbb{N} \rightarrow \mathbb{N}$ definida como:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n - 1) + n & \text{si } n > 1 \end{cases}$$

Encontrar su forma cerrada.



Solución. **Paso 1: Expansión.** Expandimos $T(n)$ usando la definición recursiva:

$$\begin{aligned} T(n) &= T(n - 1) + n \\ &= T(n - 2) + (n - 1) + n \\ &= T(n - 3) + (n - 2) + (n - 1) + n \\ &\vdots \\ &= T(1) + 2 + 3 + \dots + n \\ &= 1 + \sum_{i=2}^n i \end{aligned}$$

Paso 2: Forma cerrada. Reconocemos el patrón como una suma aritmética. Conjeturamos que $T(n) = \frac{n(n+1)}{2}$.

Paso 3: Verificación por inducción. Probamos que $T(n) = \frac{n(n+1)}{2}$ para todo $n \geq 1$.

- *Caso base:* $T(1) = 1 = \frac{1 \cdot 2}{2}$. Cumple.
- *Paso inductivo:* Sea $n > 1$. Asumimos $T(n - 1) = \frac{(n-1)n}{2}$. Entonces:

$$T(n) = T(n - 1) + n = \frac{(n-1)n}{2} + n = \frac{n(n-1) + 2n}{2} = \frac{n(n+1)}{2}$$

Luego $T(n) = \frac{n(n+1)}{2}$, y por lo tanto $T \in \Theta(n^2)$.

Y un poco más complejo, una suma geométrica.

Ejercicio 2.8.2

Sea $T : \mathbb{N} \rightarrow \mathbb{N}$ definida como:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 2T(n - 1) + 1 & \text{si } n > 0 \end{cases}$$

Encontrar su forma cerrada.



Solución. **Paso 1: Expansión.** Expandimos $T(n)$ usando la definición recursiva:

$$\begin{aligned}
 T(n) &= 2T(n-1) + 1 \\
 &= 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 \\
 &= 4(2T(n-3) + 1) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 \\
 &\vdots \\
 &= 2^n T(0) + 2^{n-1} + \dots + 2 + 1 \\
 &= 2^n + \sum_{i=0}^{n-1} 2^i
 \end{aligned}$$

Paso 2: Forma cerrada. Reconocemos el patrón como una suma geométrica. Conjeturamos que $T(n) = 2^{n+1} - 1$.

Paso 3: Verificación por inducción. Probamos que $T(n) = 2^{n+1} - 1$ para todo $n \geq 0$.

- *Caso base:* $T(0) = 1 = 2^1 - 1$. Cumple.
- *Paso inductivo:* Sea $n > 0$. Asumimos $T(n-1) = 2^n - 1$. Entonces:

$$T(n) = 2T(n-1) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$$

Luego $T(n) = 2^{n+1} - 1$, y por lo tanto $T \in \Theta(2^n)$.

Algunas recurrencias lineales tienen formas cerradas conocidas:

Recurrencia	Forma cerrada	Orden
$T(n) = T(n-1) + c$	suma aritmética	$\Theta(n)$
$T(n) = T(n-1) + n$	suma triangular	$\Theta(n^2)$
$T(n) = T(n-1) + n^k$	suma de potencias	$\Theta(n^{k+1})$
$T(n) = a \cdot T(n-1) + c$	suma geométrica	$\Theta(a^n)$

Cuando la recurrencia usa notación asintótica, la expansión nos da una cota asintótica.

Ejercicio 2.8.3

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida como:

$$T(n) = \begin{cases} 5 & \text{si } n = 1 \\ T(n-1) + O(n^2) & \text{si } n > 1 \end{cases}$$

Demostrar que $T \in O(n^3)$.

Demuestra. Por definición de la notación algebraica asintótica, existe una función $h \in O(n^2)$ tal que para todo $n > 1$, $T(n) = T(n-1) + h(n)$.

Expandiendo la recurrencia:

$$\begin{aligned}
T(n) &= T(n-1) + h(n) \\
&= T(n-2) + h(n-1) + h(n) \\
&= T(n-3) + h(n-2) + h(n-1) + h(n) \\
&\vdots \\
&= T(1) + \sum_{i=2}^n h(i) \\
&= 5 + \sum_{i=2}^n h(i)
\end{aligned}$$

Como $h \in O(n^2)$, existe $\alpha > 0$ y $n_0 \in \mathbb{N}$ tales que para todo $i \geq n_0$, $h(i) \leq \alpha i^2$. Definamos, entonces, $\beta = \max(\alpha, \max_{i=2}^{n_0} \frac{h(i)}{i^2})$. Tenemos, entonces, que $h(i) \leq \beta i^2$ para todo $i \in \mathbb{N}, i \geq 2$. Luego:

$$\sum_{i=2}^n h(i) \leq \sum_{i=2}^n \beta i^2 = \beta \sum_{i=2}^n i^2 = \beta \left(\frac{n(n+1)(2n+1)}{6} - 1 \right) \in O(n^3)$$

Por lo tanto, $T(n) = 5 + O(n^3)$, y luego $T \in O(n^3)$. \square

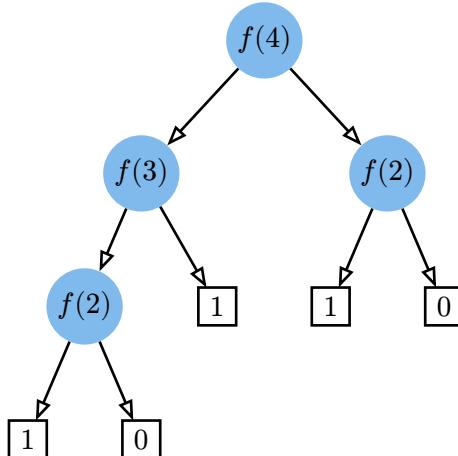
No siempre va a ser fácil encontrar un patrón mediante expansiones. En particular, cuando la recurrencia divide el argumento (por ejemplo, $T(n) = 2T(\frac{n}{2}) + n$), la expansión directa se complica. Para esos casos, usaremos árboles de recursión.

8.2 Árboles de recursión

En el contexto de ciencias de la computación, nos vamos a enfocar más en la forma de computar Y , dado X . Al ver una definición como esta para $f : \mathbb{N} \rightarrow \mathbb{N}$:

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n \geq 2 \end{cases}$$

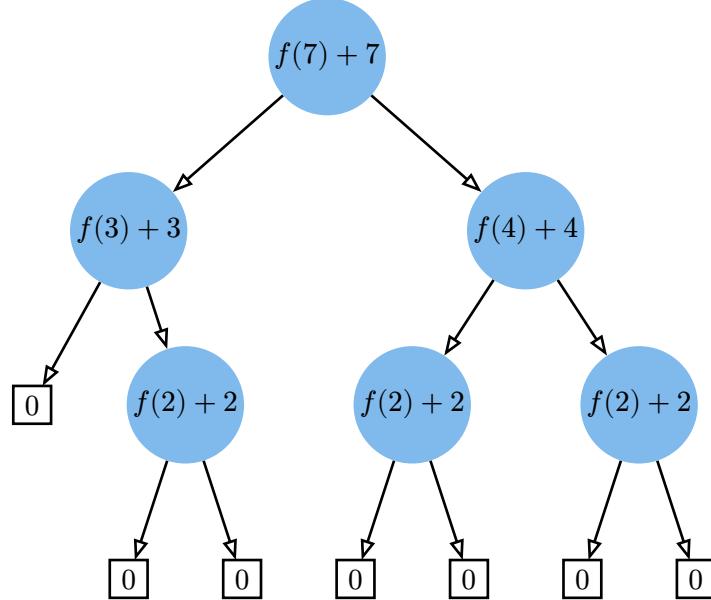
vamos a querer considerar cómo computar valores de f . Una herramienta usual para funciones recursivas va a ser el **árbol de recursión**, que nos muestra qué llamadas de f dependen de qué otras llamadas de f . Para f definida como arriba, el árbol para $n = 4$ se ve así:



Esta herramienta nos va a ser útil para encontrar formas cerradas de funciones, o al menos darnos una idea de quién puede ser. Por ejemplo, consideremos la siguiente función, $f : \mathbb{N}_{>0} \rightarrow \mathbb{N}$:

$$f(n) = \begin{cases} 0 & \text{si } n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + f(\lceil \frac{n}{2} \rceil) + n & \text{si } n > 1 \end{cases}$$

Veamos cómo usar árboles de recursión para entender esta función. Intentemos encontrar una forma cerrada para f , viendo el árbol de recursión para $n = 7$:



Esto es un árbol de 4 niveles. Podemos ver que en cada nivel donde no hay hojas, la suma los términos en cada vértice es 7. En el primer nivel, hay un sólo vértice, y suma 7 al valor total («+7»). En el segundo nivel, se suman 3 y 4, que otra vez suman 7 al valor total, en conjunto. Esto es porque cada vez que caemos en el caso recursivo, $f(n)$ tiene un $+n$ al final.

En el tercer y cuarto nivel hay hojas, y todas cuestan 0. Para saber el costo total, podemos calcular el costo de cada nivel. Para los niveles que no tienen hojas, el costo es exactamente n . Para los niveles que tienen sólo hojas, el costo es 0. Luego, queda saber cuántos niveles hay con hojas, y en esos niveles, cuántas hojas hay.

Queremos entender la estructura del árbol de recursión. Sea $k = \lfloor \log_2(n) \rfloor$. Vamos a probar formalmente que:

1. Los niveles $0, 1, \dots, k - 1$ están completos (todos sus vértices son internos).
2. El nivel k es el primer nivel que contiene hojas.
3. Si n es potencia de 2, el nivel k contiene sólo hojas y el árbol tiene altura exactamente k .
4. Si n no es potencia de 2, el nivel k contiene tanto hojas como vértices internos, y el nivel $k + 1$ contiene sólo hojas.

Para esto, primero caracterizamos los valores en cada nivel del árbol. Definimos el *valor* de un vértice como el argumento de f en ese vértice. Por ejemplo, en el árbol de $f(7)$, la raíz tiene valor 7, sus hijos tienen valores 3 y 4, etc. Un vértice es una *hoja* si su valor es 1 (caso base de f), y es un *vértice interno* si su valor es mayor a 1 (caso recursivo).

Proposición 2.8.4

Sea v el valor de un vértice en el nivel i del árbol de recursión. Entonces:

$$\left\lfloor \frac{n}{2^i} \right\rfloor \leq v \leq \left\lceil \frac{n}{2^i} \right\rceil$$

♥

Demostración. Por inducción en i .

- Caso base, $i = 0$. El único vértice en el nivel 0 es la raíz, con valor n . Como $\left\lfloor \frac{n}{2^0} \right\rfloor = n = \left\lceil \frac{n}{2^0} \right\rceil$, el predicado vale.
- Paso inductivo, $i > 0$. Sea v un vértice en el nivel i . Su padre, en el nivel $i - 1$, tiene valor p tal que por hipótesis inductiva, $\left\lfloor \frac{n}{2^{i-1}} \right\rfloor \leq p \leq \left\lceil \frac{n}{2^{i-1}} \right\rceil$. El valor de v es $\left\lfloor \frac{p}{2} \right\rfloor$ o $\left\lceil \frac{p}{2} \right\rceil$. Usando que $\left\lfloor \frac{\lfloor x \rfloor}{m} \right\rfloor = \left\lfloor \frac{x}{m} \right\rfloor$ y $\left\lceil \frac{\lceil x \rceil}{m} \right\rceil = \left\lceil \frac{x}{m} \right\rceil$ para $m \in \mathbb{N}_{>0}$:

$$\lfloor v \rfloor \geq \left\lfloor \frac{\lfloor \frac{n}{2^{i-1}} \rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2^i} \right\rfloor$$

$$\lceil v \rceil \leq \left\lceil \frac{\lceil \frac{n}{2^{i-1}} \rceil}{2} \right\rceil = \left\lceil \frac{n}{2^i} \right\rceil$$

Luego, $\left\lfloor \frac{n}{2^i} \right\rfloor \leq v \leq \left\lceil \frac{n}{2^i} \right\rceil$.

□

Ahora podemos determinar qué niveles contienen hojas (vértices con valor 1) y cuáles contienen sólo vértices internos (valor ≥ 2).

Como $k = \lfloor \log_2(n) \rfloor$, tenemos que $2^k \leq n < 2^{k+1}$.

1. Niveles $i < k$ están completos: Para $i < k$, tenemos $2^i \leq 2^{k-1} < \frac{2^k}{2} \leq \frac{n}{2}$. Luego:

$$\left\lfloor \frac{n}{2^i} \right\rfloor \geq \left\lfloor \frac{n}{2^{k-1}} \right\rfloor \geq \left\lfloor \frac{2^k}{2^{k-1}} \right\rfloor = 2$$

Por la proposición anterior, todo vértice v en el nivel i tiene $v \geq 2$, así que es un vértice interno.

1. El nivel k contiene hojas: Tenemos $2^k \leq n < 2^{k+1}$, lo que implica $1 \leq \frac{n}{2^k} < 2$. Luego $\left\lfloor \frac{n}{2^k} \right\rfloor = 1$. El vértice en la rama más a la izquierda (que siempre toma $\lfloor \cdot \rfloor$) tiene valor exactamente $\left\lfloor \frac{n}{2^k} \right\rfloor = 1$, que es una hoja.
2. Si n es potencia de 2, entonces $n = 2^k$. Luego, $\frac{n}{2^k} = 1$, y $\lceil \frac{n}{2^k} \rceil = 1$. Por la proposición, todo vértice en el nivel k tiene valor v con $1 \leq v \leq 1$, así que $v = 1$. Todos son hojas, y el árbol tiene altura exactamente k .
3. Si n no es potencia de 2, entonces $n \neq 2^k$. Como además $n \geq 2^k$, tenemos $n > 2^k$. Combinando con $n < 2^{k+1}$ y dividiendo por 2^k , obtenemos $1 < \frac{n}{2^k} < 2$. Luego, $\lceil \frac{n}{2^k} \rceil = 2$. El vértice más a la derecha tiene valor $\lceil \frac{n}{2^k} \rceil = 2$, que es un vértice interno. Luego el nivel k tiene tanto hojas (valor 1) como vértices internos (valor 2). Para el nivel $k + 1$, como $n < 2^{k+1}$:

$$\left\lceil \frac{n}{2^{k+1}} \right\rceil \leq \left\lceil \frac{2^{k+1}-1}{2^{k+1}} \right\rceil = 1$$

Así, todo vértice en el nivel $k + 1$ tiene valor 1, es decir, es hoja. El árbol tiene altura $k + 1$.

Resumiendo: el árbol tiene k niveles completos (0 a $k - 1$), un nivel k que es el primero con hojas, y posiblemente un nivel $k + 1$ con sólo hojas (si n no es potencia de 2).

Sabiendo esto, calculemos $f(n)$. Sean L el número de hojas en el nivel k , y m el número de vértices internos en el nivel k . Como los niveles $0, \dots, k - 1$ están completos, el nivel k tiene exactamente 2^k vértices:

$$m + L = 2^k$$

Cada vértice interno en el nivel k tiene valor 2 (sus dos hijos serán hojas con valor 1). Cada hoja en el nivel k tiene valor 1. Como la suma de valores en cualquier nivel completo es n , y el nivel k tiene hojas de valor 1 y vértices internos de valor 2:

$$1 \cdot L + 2 \cdot m = n$$

Combinando ambas ecuaciones, obtenemos:

$$\begin{aligned} m + L &= 2^k \\ 2m + L &= n \end{aligned}$$

Restando, $m = n - 2^k$, y luego $L = 2^{k+1} - n$.

Los niveles $0, \dots, k - 1$ son completos y cada uno suma exactamente n , aportando $k \cdot n$ al total. El nivel k tiene m vértices internos de valor 2, aportando $2m$. Las hojas aportan 0. Entonces:

$$f(n) = k \cdot n + 2m = k \cdot n + 2(n - 2^k) = n \lfloor \log_2(n) \rfloor + 2n - 2^{\lfloor \log_2(n) \rfloor + 1}$$

Notemos que, en particular, cuando n es potencia de 2, $f(n) = n \log_2 n$.

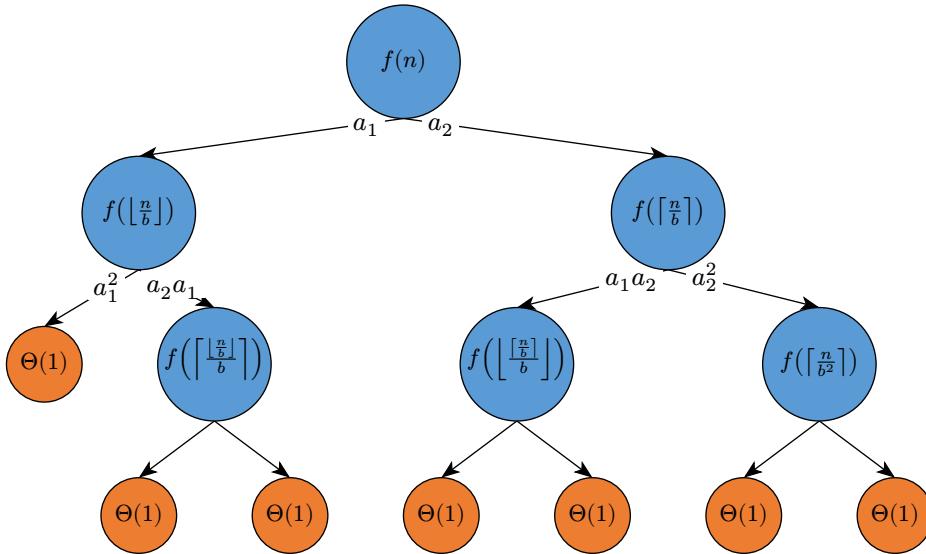
8.3 Teorema maestro

Acotar esa función nos costó bastante trabajo. Afortunadamente hay un teorema que nos ayuda bastante, pues lo podemos aplicar mecánicamente, y generaliza lo que hicimos.

La idea del teorema es mirar el árbol de recurrencia de nuestra función T . Cada vértice va a contribuir algo a la suma total. Hay aproximadamente $a^{\log_b n} = n^{\log_b a}$ hojas.

Pueden pasar tres cosas con el valor de T :

- El valor de las hojas domina a f . Si f crece menos rápido que $n^{\log_b a}$, entonces la mayor contribución al valor de T viene, asintóticamente, dada por las hojas. Luego $T \in \Theta(n^{\log_b a})$.
- El valor de las hojas está balanceado con el valor de f . Luego el costo de cada nivel es $f(n)$ o $n^{\log_b a}$, cualquiera de las dos, y habiendo $\log_b n$ niveles, tenemos que $T \in \Theta(n^{\log_b a} \log_b^{k+1} n)$.
- El valor de f domina a los valores de las hojas. Obtenemos $T \in \Theta(f)$.



Suma de valores de vértices internos:

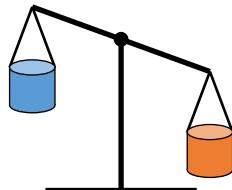
$$\sum_{j < d} a^j f\left(\frac{n}{b^j}\right)$$

con $d = \left\lfloor \log_b \left(\frac{n}{n_0}\right) \right\rfloor + 1$

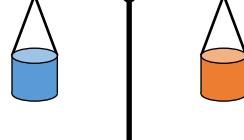
Suma de valores de hojas:

$$\Theta(a^d) = \Theta(n^c)$$

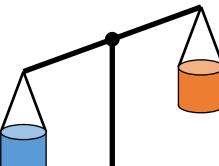
con $c = \log_b a$, y
 $d = \left\lfloor \log_b \left(\frac{n}{n_0}\right) \right\rfloor + 1$



Caso 1: Las hojas dominan.



Caso 2: Las hojas y los vértices internos están balanceados.



Caso 3: Los vértices internos dominan.

Teorema 9 (Teorema maestro)

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ una función tal que:

$$T(n) = \begin{cases} b_n & \text{si } 0 \leq n < n_0 \\ a_1 T\left(\lfloor \frac{n}{b} \rfloor\right) + a_2 T\left(\lceil \frac{n}{b} \rceil\right) + f(n) & \text{si } n \geq n_0 \end{cases}$$

para alguna función $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, algunas constantes $b_0, \dots, b_{n_0-1} \in \mathbb{R}_{\geq 0}$, $a_1, a_2 \in \mathbb{N}$, $b \in \mathbb{N} \geq 2$, $n_0 \in \mathbb{N}$, y asumiendo $n_0 \geq 2$ si $a_2 > 0$ para estar bien definida. Llameemos $a = a_1 + a_2 \geq 1$, y $c = \log_b a$. Entonces:

- Si $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$, entonces $T \in \Theta(n^c)$.
- Si $f \in \Theta(n^c \log^k n)$ para algún $k \in \mathbb{N}$, entonces $T \in \Theta(n^c \log^{k+1} n)$.
- Si $f \in \Omega(n^{c+\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$, y además existen $n_1 \in \mathbb{N}$ y $r < 1 \in \mathbb{R}_{\geq 0}$ tal que $a_1 f\left(\lfloor \frac{n}{b} \rfloor\right) + a_2 f\left(\lceil \frac{n}{b} \rceil\right) \leq r f(n)$ para todo $n \geq n_1$, entonces $T \in \Theta(f)$.



La demostración está en la Sección 1 del apéndice. Veamos cómo usar este teorema.

Proposición 2.8.5

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida como:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + 35 & \text{si } n \geq 1 \end{cases}$$

Demostrar que $T \in \Theta(\log n)$.



Demostración. Veamos en qué caso del teorema caemos. Esta recurrencia es de la forma $a_1 = 1$, $a_2 = 0$, $b = 2$. Luego, $c = \log_b a = \log_2 1 = 0$. Tenemos $f(n) = 35$. Es cierto que $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? No, pues $n^c = n^0 = 1$, con lo cual $O(n^{c-\varepsilon}) = O(\frac{1}{n^\varepsilon})$. Una constante (35 en este caso) no puede siempre ser menor que $\frac{1}{n^\varepsilon}$ para algún ningún $\varepsilon > 0$, pues eventualmente $\frac{1}{n^\varepsilon}$ decrece hasta ser arbitrariamente cercana a cero. Luego no caemos en el primer caso.

Es cierto que $f \in \Theta(n^c \log^k n)$? Sí, con $k = 0$. Este conjunto es $\Theta(n^c \log^0 n) = \Theta(n^0 (\log n)^0) = \Theta(1)$, y efectivamente $f \in \Theta(1)$. Luego, el teorema maestro nos dice que $T \in \Theta(n^0 \log^1 n) = \Theta(\log n)$. \square

Proposición 2.8.6

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida como:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + 7n^2 & \text{si } n \geq 1 \end{cases}$$

Demostrar que $T \in \Theta(n^2)$.



Demostración. Veamos en qué caso del teorema caemos. Esta recurrencia es de la forma $a_1 = 1$, $a_2 = 0$, $b = 2$. Luego, $c = \log_b a = \log_2 1 = 0$. Tenemos $f(n) = 7n^2$. Es cierto que $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? No, pues $f \in \Omega(n^2)$, cuya intersección con $O(\frac{1}{n^\varepsilon})$ es vacía.

Es cierto que $f \in \Theta(n^c \log^k n)$ para algún $k \in \mathbb{N}$? No, pues $n^c = n^0 = 1$, con lo cual $\Theta(n^c \log^k n) = \Theta(\log^k n)$. Luego no caemos en este caso, pues no hay ningún k tal que $f \in \Theta(\log^k n)$, al ser f un polinomio de grado mayor a 0.

Es cierto que $f \in \Omega(n^{c+\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? Sí, con $\varepsilon = 2$. Este conjunto es $\Omega(n^{c+\varepsilon}) = \Omega(n^2)$, y efectivamente $f \in \Omega(n^2)$ (y de hecho, $f \in \Theta(n^2)$). Ahora verifiquemos la condición de regularidad. Queremos encontrar $0 \leq r < 1$ tal que

$$\begin{aligned} a_1 f\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + a_2 f\left(\left\lceil \frac{n}{b} \right\rceil\right) &= f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\leq r f(n) \end{aligned}$$

Como f es monotónicamente creciente, tenemos:

$$f\left(\left\lfloor \frac{n}{b} \right\rfloor\right) \leq f\left(\frac{n}{2}\right) = \left(\frac{7}{4}\right)n^2 \leq \left(\frac{1}{4}\right)7n^2 = \frac{1}{4}f(n)$$

Luego, tomando $r = \frac{1}{4}$, tenemos que $f\left(\left\lfloor \frac{n}{b} \right\rfloor\right) \leq rf(n)$, para todo $n \in \mathbb{N}$. Luego $T \in \Theta(n^2)$. \square

Proposición 2.8.7

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida como:

$$T(n) = \begin{cases} 3 & \text{si } n = 0 \\ 2 & \text{si } n = 1 \\ 34 & \text{si } n = 2 \\ 9 & \text{si } n = 3 \\ 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + T\left(\lceil \frac{n}{4} \rceil\right) + 7n + 9 & \text{si } n \geq 4 \end{cases}$$

Demostrar que $T \in \Theta(n \log n)$.



Demostración. Tenemos $a = a_1 + a_2 = 3 + 1 = 4$, y $b = 4$, con lo cual $c = \log_b a = \log_4 4 = 1$. Tenemos $f(n) = 7n + 9$. Es cierto que $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? No, pues $n^c = n^1 = n$, con lo cual $O(n^{c-\varepsilon}) = O\left(\frac{n^1}{n^\varepsilon}\right)$. Un término lineal ($7n + 9$ en este caso) no puede siempre ser menor que $\frac{n}{n^\varepsilon}$ para algún ningún $\varepsilon > 0$, pues eventualmente $\frac{n}{n^\varepsilon}$ crece hasta ser arbitrariamente grande. Luego no caemos en el primer caso.

Es cierto que $f \in \Theta(n^c \log^k n)$? No, pues $n^c = n^1 = n$, con lo cual $\Theta(n^c \log^k n) = \Theta(n \log^k n)$. Con $k = 0$, tenemos precisamente que $f \in \Theta(n \log^0 n) = \Theta(n)$. Luego, el teorema maestro nos dice que $T \in \Theta(n \log^1 n) = \Theta(n \log n)$. \square

Proposición 2.8.8

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida como:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 4T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n & \text{si } n \geq 2 \end{cases}$$

Demostrar que $T \in \Theta(n^2)$.



Demostración. Tenemos $a = 4$, $b = 2$, con lo cual $c = \log_b a = \log_2 4 = 2$. Tenemos $f(n) = n$.

Es cierto que $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? Tomando $\varepsilon = 1$, tenemos $n^{c-\varepsilon} = n^{2-1} = n$. Efectivamente $f(n) = n \in O(n)$. Luego caemos en el primer caso.

El teorema maestro nos dice que $T \in \Theta(n^c) = \Theta(n^2)$. \square

Proposición 2.8.9

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida como:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ 8T(\lfloor \frac{n}{3} \rfloor) + n^2 & \text{si } n \geq 3 \end{cases}$$

Demostrar que $T \in \Theta(n^2)$.



Demostración. Tenemos $a = 8$, $b = 3$, con lo cual $c = \log_b a = \log_3 8 \approx 1.89$. Tenemos $f(n) = n^2$.

Es cierto que $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? Tendríamos que $n^2 \in O(n^{c-\varepsilon})$, lo cual requiere $2 \leq c - \varepsilon$, es decir $\varepsilon \leq c - 2 \approx -0.11 < 0$. Como necesitamos $\varepsilon > 0$, no caemos en el primer caso.

Es cierto que $f \in \Theta(n^c \log^k n)$ para algún $k \in \mathbb{N}$? Tendríamos $n^2 \in \Theta(n^c \log^k n)$ con $c \approx 1.89 < 2$. Pero n^2 crece estrictamente más rápido que $n^c \log^k n$ para cualquier k fijo, pues $\lim_{n \rightarrow \infty} \frac{n^2}{n^c \log^k n} = \lim_{n \rightarrow \infty} \frac{n^{2-c}}{\log^k n} = \infty$. Luego no caemos en el segundo caso.

Es cierto que $f \in \Omega(n^{c+\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? Tomando $\varepsilon = 0.1$, tenemos $c + \varepsilon \approx 1.99 < 2$, así que $n^2 \in \Omega(n^{1.99})$. Esto es cierto.

Debemos verificar la condición de regularidad: que exista $k < 1$ tal que $a \cdot f(\frac{n}{b}) \leq k \cdot f(n)$. Tenemos:

$$8 \cdot f\left(\frac{n}{3}\right) = 8 \cdot \left(\frac{n}{3}\right)^2 = 8 \frac{n^2}{9}$$

Queremos $8 \frac{n^2}{9} \leq k \cdot n^2$, es decir $k \geq \frac{8}{9}$. Tomando $k = \frac{8}{9} < 1$, la condición se cumple.

Luego caemos en el tercer caso, y el teorema maestro nos dice que $T \in \Theta(f(n)) = \Theta(n^2)$. \square

Proposición 2.8.10

El número de multiplicaciones que realiza el algoritmo de Strassen para multiplicación de matrices, dadas dos matrices de $n \times n$ donde n es una potencia de 2, tiene la recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{si } n > 1 \end{cases}$$

Demostrar que $T \in \Theta(n^{\log_2 7})$.



Demostración. Tenemos $a = 7$, $b = 2$, con lo cual $c = \log_b a = \log_2 7 \approx 2.81$. Tenemos $f(n) \in \Theta(n^2)$.

Es cierto que $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$? Tomando $\varepsilon = 0.8$, tenemos $c - \varepsilon \approx 2.01$. Debemos verificar que $n^2 \in O(n^{2.01})$, lo cual es cierto pues $2 < 2.01$. Luego caemos en el primer caso.

El teorema maestro nos dice que $T \in \Theta(n^c) = \Theta(n^{\log_2 7})$. □

Proposición 2.8.11

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, definida como:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 2T(\lfloor \frac{n}{2} \rfloor) + \frac{n}{\log n} & \text{si } n \geq 1 \end{cases}$$

Mostrar que el teorema maestro no aplica directamente a esta recurrencia.



Demostración. Tenemos $a = 2$, $b = 2$, con lo cual $c = \log_b a = \log_2 2 = 1$. Tenemos $f(n) = \frac{n}{\log n}$.

Veamos que no caemos en ninguno de los tres casos.

- Caso 1. Requiere $f \in O(n^{c-\varepsilon}) = O(n^{1-\varepsilon})$ para algún $\varepsilon > 0$. Pero $\lim_{n \rightarrow \infty} \frac{n}{n^{1-\varepsilon}} = \lim_{n \rightarrow \infty} \frac{n^\varepsilon}{\log n} = \infty$ para todo $\varepsilon > 0$. Luego $\frac{n}{\log n} \notin O(n^{1-\varepsilon})$ para ningún $\varepsilon > 0$.
- Caso 2. Requiere $f \in \Theta(n^c \log^k n) = \Theta(n \log^k n)$ para algún $k \geq 0$. Tenemos $f(n) = \frac{n}{\log n} = n \log^{-1} n$. Esto correspondería a $k = -1$, pero el teorema maestro requiere $k \geq 0$. Luego no caemos en el segundo caso.
- Caso 3. Requiere $f \in \Omega(n^{c+\varepsilon}) = \Omega(n^{1+\varepsilon})$ para algún $\varepsilon > 0$. Pero $\lim_{n \rightarrow \infty} \frac{n}{n^{1+\varepsilon}} = \lim_{n \rightarrow \infty} \frac{1}{n^\varepsilon \log n} = 0$ para todo $\varepsilon > 0$. Luego $\frac{n}{\log n} \notin \Omega(n^{1+\varepsilon})$ para ningún $\varepsilon > 0$.

Como no caemos en ninguno de los tres casos, el teorema maestro no aplica a esta recurrencia. Habría que usar otros métodos (como el árbol de recursión o el método de sustitución) para determinar su comportamiento asintótico. □

Proposición 2.8.12

Sea la $T : \mathbb{N} \rightarrow \mathbb{N}$ una función dada por $T(0) = 1$, y para todo $n \in \mathbb{N}$ tal que $n > 0$, definimos $T(n) = 2T(\frac{n}{2}) + \Theta(n \log n)$. Es decir, existe una función $h : \mathbb{N} \rightarrow \mathbb{N}$ tal que $h \in \Theta(n \log n)$, y $T(n) = 2T(\frac{n}{2}) + h(n)$ para todo $n \in \mathbb{N}, n > 0$.

Probar que $T \in \Theta(n \log^2 n)$.



Demostración. Podemos usar el teorema maestro, que nos dice que si tenemos una función T de la forma $T(n) = aT(\frac{n}{b}) + f(n)$ para todo $n \in \mathbb{N}, n > 0$, y $f \in \Theta(n^{\log_b(a)} \log^k n)$ para algún $k \in \mathbb{N}$, entonces $T \in \Theta(n^{\log_b(a)} \log^{k+1}(n))$. Basta elegir $k = 1$, $a = 2$, $b = 2$, para ver que estamos dentro de las condiciones de este caso del teorema, y como $\log_2(2) = 1$, tenemos que $T \in \Theta(n \log^2 n)$. □

8.4 Ejercicios

Ejercicio 2.8.13

Sea $T : \mathbb{N} \rightarrow \mathbb{N}$ una función que cumple que para todo $n > 0$, $T(n) = 4T(\lfloor \frac{n}{3} \rfloor) + O(n \log n)$, y $T(0) = 0$. Probar que $T \in \Theta(n^{\log_3 4})$.



Ejercicio 2.8.14

Sea $T : \mathbb{N} \rightarrow \mathbb{N}$ una función que cumple que para todo $n > 0$, $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n \log n$, y $T(0) = 7$. Probar que $T \in \Theta(n \log^2 n)$.



Ejercicio 2.8.15

Sea $T : \mathbb{N} \rightarrow \mathbb{N}$ tal que $T(n) = 2T(\lfloor \frac{n}{4} \rfloor) + \sqrt{n}$, y $T(0) = 1$. Probar que $T \in \Theta(\sqrt{n} \log n)$.



Ejercicio 2.8.16

Sea $T : \mathbb{N} \rightarrow \mathbb{N}$ una función que cumple que para todo $n > 4$, $T(n) = 16T(\lfloor \frac{n}{4} \rfloor) + n!$, y $T(k) = k^2$ para $0 \leq k \leq 4$. Probar que $T \in \Theta(n!)$.



Ejercicio 2.8.17

Sea $T : \mathbb{N} \rightarrow \mathbb{N}$ una función que cumple que para todo $n > 0$, $T(n) = 3T(\lfloor \frac{n}{3} \rfloor) + \sqrt{n}$, y $T(0) = 0$. Probar que $T \in \Theta(n)$.



Parte III: Diseño y correctitud de algoritmos

1 Correctitud de algoritmos

TODO: This.

2 Análisis de complejidad de algoritmos

Vamos a poner en práctica lo que aprendimos sobre análisis asintótico de funciones, para entender el comportamiento de algoritmos. Para eso, necesitamos establecer un puente entre el mundo abstracto de las funciones $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ y el mundo concreto de los algoritmos que corren en computadoras.

2.1 Modelo de cómputo

Cuando analizamos un algoritmo, estamos haciendo una abstracción. No analizamos código en un lenguaje particular corriendo en una computadora específica, con su procesador, su memoria, y su sistema operativo. En cambio, asumimos un *modelo de cómputo*: una descripción simplificada de cómo funciona una computadora, que define qué operaciones básicas podemos realizar.

Definición 14

El **Random Access Machine**[14] (RAM) es un modelo de cómputo con las siguientes características:

- La memoria es una secuencia infinita de celdas, cada una capaz de almacenar un número entero de tamaño arbitrario.
- Leer o escribir cualquier celda de memoria toma tiempo constante (de ahí «random access»).
- Las operaciones aritméticas básicas (suma, resta, multiplicación, división, comparación) toman tiempo constante.
- Las operaciones de control de flujo (saltos condicionales, llamadas a funciones) toman tiempo constante.



Este es el modelo que usamos implícitamente cuando analizamos algoritmos en esta materia. Es una simplificación útil: nos permite ignorar detalles como el tamaño del caché, la velocidad del disco, o las optimizaciones del compilador, y enfocarnos en la estructura del algoritmo: operaciones aritméticas/lógicas, control de flujo, y acceso a memoria.

ⓘ Nota

Existen otros modelos de cómputo. Las *máquinas de Turing*[15] son el modelo teórico más fundamental, usado para definir qué es computable. El modelo *word RAM*[16] asume que los enteros tienen un tamaño máximo de w bits. El modelo de *memoria externa*[17] cuenta accesos a disco en vez de operaciones. El modelo de *random-access stored-program machine*[18] (RASP) es igual a RAM, excepto que la lista de instrucciones está en sí almacenada en la memoria, en vez de en un lugar separado. Cada modelo es útil para distintos propósitos, pero el modelo RAM es el estándar para análisis de algoritmos.

Notemos que el modelo RAM asume que sumar o multiplicar cualquier par de enteros es una única operación. Si queremos que este modelo modele nuestras computadoras, esto es demasiado para asumir: Nuestras computadoras representan enteros grandes usando muchos bits, y el tiempo real que toma sumarlos depende del número de bits que tengan. El modelo word RAM es más realista cuando trabajamos con enteros que caben en una palabra de w bits, y además nos permite analizar operaciones bit a bit. Sin embargo, si nuestro problema trata de enteros de precisión arbitraria (mucho más grandes que w bits), como en criptografía o teoría de números, ni RAM ni word RAM son adecuados: deberíamos contar operaciones sobre bits o dígitos.

El modelo determina qué operaciones contamos y cómo las pesamos. Cuando decimos que una ejecución de un algoritmo con cierta entrada hace « n^2 operaciones», estamos diciendo que hace n^2 operaciones básicas bajo el modelo RAM. Si usáramos otro modelo, el conteo podría ser diferente.

2.2 Tamaño de entrada

Para analizar un algoritmo, necesitamos definir qué significa el «tamaño» de una entrada. Esta elección no es única, y es parte del modelado del problema.

Definición 15

Dado un algoritmo A que recibe entradas de un conjunto \mathcal{I} , una **función de tamaño** es una función $|\cdot| : \mathcal{I} \rightarrow \mathbb{N}$ que asigna a cada entrada un número natural que llamamos su *tamaño*.



Algunas elecciones comunes de tamaño:

Tipo de entrada	Tamaño típico
Lista o arreglo	$n = \text{número de elementos}$
Cadena de caracteres	$n = \text{longitud de la cadena}$
Número entero k	$n = k$ (valor), o $n = \log_2 k$ (bits)
Matriz M de $m \times n$ enteros	$m \times n$

⚠️ Advertencia

La elección del tamaño afecta el análisis. Consideremos un algoritmo que determina si un número k es primo, iterando desde 2 hasta \sqrt{k} .

```
def es_primo(k: int) -> bool:
    for i in range(2, int(sqrt(k)) + 1):
        if k % i == 0:
            return False
    return True
```

- Si definimos tamaño como $n = k$, el algoritmo hace $O(\sqrt{n})$ operaciones.
- Si definimos tamaño como $n = \log_2 k$ (el número de bits para representar k), el algoritmo hace $O(\sqrt{2^n}) = O(2^{\frac{n}{2}})$ operaciones.

Ambos análisis son correctos, pero dicen cosas distintas. El segundo es más honesto, pues tenemos que usar $\log_2 k$ bits para representar k . También es fácil ver por qué el algoritmo trivial es demasiado lento en la práctica: toma tiempo *exponencial* en el tamaño de la entrada en el peor caso.

Sugiero que sean precisos a la hora de definir el tamaño de entrada de su problema. Cuando veamos algoritmos sobre grafos, no va a ser lo mismo que nuestra entrada sea una lista de m pares de enteros, que una matriz binaria de $n \times n$ celdas. El tamaño de entrada, así como el algoritmo entero, va a tener que corresponderse con cómo representamos la entrada.

2.3 Funciones de costo

Una vez fijados el modelo de cómputo y la función de tamaño, podemos definir funciones que miden el costo de ejecutar un algoritmo. Algunas funciones comunes son las que miden el número de operaciones necesarias y el espacio necesario.

Definición 16

Sea A un algoritmo que recibe entradas del conjunto \mathcal{I} . Definimos:

- $T_A : \mathcal{I} \rightarrow \mathbb{R}_{\geq 0}$, donde $T_A(x)$ es el número de operaciones que realiza A al recibir la entrada x . Esto también se conoce como el **tiempo** que toma ejecutar $A(x)$.
- $S_A : \mathcal{I} \rightarrow \mathbb{R}_{\geq 0}$, donde $S_A(x)$ es la cantidad de memoria que usa A al recibir la entrada x , por encima de la memoria usada por la entrada.



Estas funciones dependen del algoritmo y de la entrada específica, no sólo de su tamaño. Por ejemplo, un algoritmo de ordenamiento puede hacer distinto número de comparaciones para dos listas de la misma longitud.

(i) Nota

Podemos usar funciones de costo para medir otros recursos además de tiempo y espacio:

- Número de comparaciones (para algoritmos de ordenamiento o búsqueda).
- Número de accesos a disco (para algoritmos de memoria externa).
- Número de mensajes enviados (para algoritmos distribuidos).
- Número de multiplicaciones (para algoritmos numéricos donde son costosas).

La elección depende de qué recurso es el cuello de botella en nuestra aplicación.

Veamos algunos ejemplos de funciones de costo que miden distintos recursos.

Ejemplo 3.2.1 (Tiempo)

Consideremos el siguiente algoritmo que calcula la suma de los elementos de una lista:

```
def sumar(lista: list[int], n: int) -> int:
    total = 0
    i = 0
    while i < n:
        total += lista[i]
        i += 1
    return total
```

Sea n la longitud de la lista `lista`. Definimos el tamaño de entrada como n . En cada iteración del ciclo se realizan una comparación, dos sumas, y un acceso a memoria (constantes bajo el modelo RAM), y hay n iteraciones. Antes de entrar al ciclo hay una asignación, y luego de la última iteración del ciclo realizamos una comparación que da falso, que nos dice que podemos salir del ciclo. Luego, si estamos midiendo el número de operaciones, $T_A(x) = 4n + 2$ para toda lista x de longitud n . En este caso, la función de costo no depende de los valores concretos de la lista, sino sólo de su longitud.



Notemos cómo analizando los pasos que realiza el algoritmo, y contando lo que nos interesa, conseguimos la función de costo.

Ejemplo 3.2.2 (Comparaciones)

Consideremos un algoritmo que encuentra el máximo de una lista:

```
def maximo(lista: list[int], n: int) -> int:
    m = lista[0]
    for i in range(1, n):
        if lista[i] > m:
            m = lista[i]
    return m
```

Si definimos nuestro costo C_A como el número de comparaciones que realiza nuestro algoritmo A , para cualquier lista de longitud n se realizan exactamente $n - 1$ comparaciones: una por cada iteración del ciclo. La función de costo es, luego, $C_A(x) = n - 1$, independiente de los valores de la entrada.

En cambio, si midiéramos el número de *asignaciones* a m , ese valor sí depende de la entrada. Si la lista está ordenada de menor a mayor, se hacen n asignaciones (la inicial más una por cada iteración). Si está ordenada de mayor a menor, se hace una sola asignación (la inicial). Esto muestra que la elección del recurso a medir puede hacer que la función de costo dependa o no de la entrada concreta.

Ejemplo 3.2.3 (Mensajes enviados)

En sistemas distribuidos, a menudo medimos el número de mensajes que se envían entre los nodos de una red. Consideremos una red de n nodos conectados en anillo (cada nodo puede comunicarse con sus dos vecinos), y un algoritmo simple de *broadcast*: el nodo 0 quiere enviar un dato a todos los demás.

```
def broadcast(x, id: int, n: int):
    if id == 0:
        send(x, id + 1)
    else:
        y = receive(id - 1)
        if id != n - 1:
            send(y, id + 1)
```

Este algoritmo envía exactamente $n - 1$ mensajes: el nodo 0 inicia, y luego cada nodo intermedio reenvía, hasta que el dato llega al nodo $n - 1$. La función de costo es $M_A(x) = n - 1$, donde M mide mensajes enviados.

En cambio el siguiente algoritmo, que envía mensajes en paralelo, usa el mismo número de mensajes, pero tiempo logarítmico:

```
def broadcast_paralelo(x, id: int, n: int):
    if id != 0:
        x = receive(id//2)
    if id < n / 2:
        parallel(
            send(x, 2 * id + 1),
            send(x, 2 * id + 2),
        )
```

Ejemplo 3.2.4 (Espacio)

Consideremos dos algoritmos que invierten una lista.

El primero usa un arreglo auxiliar:

```
def invertir_con_copia(lista: list[int]) -> list[int]:
    n = len(lista)
    resultado = [0] * n
    for i in range(n):
        resultado[n - 1 - i] = lista[i]
    return resultado
```

El segundo lo hace *in-place*, intercambiando elementos:

```
def invertir_sin_copia(lista: list[int]) -> None:
    n = len(lista)
    for i in range(n // 2):
        lista[i], lista[n - 1 - i] = (
            lista[n - 1 - i], lista[i]
        )
```

Si medimos el espacio *auxiliar* (la memoria adicional que usa el algoritmo más allá de la entrada), el primero tiene $S_A(x) = n$ (necesita un arreglo de n celdas), mientras que el segundo tiene $S_A(x) = 1$ (sólo usa una variable temporal para el intercambio). Ambos realizan $\Theta(n)$ operaciones en tiempo, pero difieren significativamente en el uso de espacio.



Ejemplo 3.2.5 (Tiempo amortizado)

Consideremos un ejemplo más interesante sobre uso de tiempo, donde debemos amortizar el costo de una operación. El siguiente código es esencialmente el mismo que usa C++ para insertar en `std::vector`, o en general, cualquier vector redimensionable en cualquier lenguaje.

```
void* malloc(int size); // Tiempo constante.
void free(void* ptr); // Tiempo constante.

struct vector {
    int* datos;
    int capacidad;
    int tamaño;
};

void vector_agregar(struct vector* v, int x) {
    if (v->tamaño == v->capacidad) {
        // Duplicamos la capacidad.
        int nueva_capacidad = max(1, v->capacidad * 2);
        int* nuevos_datos = malloc(nueva_capacidad * sizeof(int));

        // Copiamos los elementos existentes.
        for (int i = 0; i < v->tamaño; i++) {
            nuevos_datos[i] = v->datos[i];
        }

        // Liberamos la memoria anterior.
        free(v->datos);
        v->datos = nuevos_datos;
        v->capacidad = nueva_capacidad;
    }

    v->datos[v->tamaño] = x;
    v->tamaño++;
}
```

```

    // Actualizamos el vector.
    v->datos = nuevos_datos;
    v->capacidad = nueva_capacidad;
}

// Agregamos el nuevo elemento.
v->datos[v->tamaño] = x;
v->tamaño++;
}

void f(int n) {
    struct vector v = {
        datos: NULL,
        capacidad: 0,
        tamaño: 0
    };

    for (int i = 0; i < n; i++) {
        vector_agregar(&v, i);
    }
}

```

Para entender cuánto tiempo toma $f(n)$, tenemos que entender cuánto tiempo va a tomar cada llamada a `vector_agregar`. Cada llamada a `vector_agregar` toma tiempo $\Theta(1)$ si $v->tamaño < v->capacidad$, y tiempo $\Theta(t)$ (donde t es $v->tamaño$) si hay que redimensionar, porque se copian todos los elementos. En el peor caso una sola llamada toma $\Theta(n)$, pero las redimensiones son infrecuentes: ocurren cuando el tamaño es $0, 1, 2, 4, 8, \dots$, es decir en potencias de 2.

Contemos el costo total de las n llamadas a `vector_agregar`. La j -ésima redimensión (para $j \geq 1$) copia 2^{j-1} elementos. La última redimensión antes de la inserción n -ésima ocurre cuando el tamaño es $2^{\lfloor \log_2(n-1) \rfloor}$. Luego, el costo total de todas las copias es:

$$\sum_{j=0}^{\lfloor \log_2(n-1) \rfloor} 2^j = 2^{\lfloor \log_2(n-1) \rfloor + 1} - 1 \leq 2(n-1) - 1 < 2n \in O(n)$$

Sumando esto con las n asignaciones de costo $\Theta(1)$ cada una, el costo total de las n llamadas es $\Theta(n)$. El costo *amortizado* por llamada es entonces $\frac{\Theta(n)}{n} = \Theta(1)$: si bien una llamada individual a `vector_agregar` puede tomar $\Theta(n)$ en el peor caso, el costo promediado sobre la secuencia de n llamadas es constante.

La función `f`, entonces, toma tiempo $\Theta(n)$.

Este último ejemplo fue un caso de análisis amortizado. Hay varias otras formas de hacer este tipo de análisis, como el método de banquero y el método de potencial. Para los interesados en estos métodos, recomiendo el libro [19], la sección de análisis amortizado. La fuente original del análisis amortizado es [20].

Cuando un algoritmo es recursivo, su función de costo se expresa naturalmente como una *recurrencia*: una ecuación que define $T(n)$ en términos de T evaluada en entradas más pequeñas. Para derivar la recurrencia, leemos el código e identificamos:

1. El caso base: qué hace el algoritmo cuando no hace recursión. Esto nos dice cuánto vale T para los tamaños más chicos.

2. Las llamadas recursivas: cuántas hay, y con qué tamaño de entrada. Cada llamada contribuye un término $T(\dots)$.
3. El trabajo no recursivo: qué hace el algoritmo además de las llamadas recursivas. Esto nos da los términos aditivos a T .

Veamos un ejemplo.

Ejemplo 3.2.6 (Algoritmos recursivos)

En el problema de las Torres de Hanoi[13], tenemos n discos de tamaños distintos apilados en una varilla (el más grande abajo), y queremos moverlos a otra varilla, usando una varilla auxiliar, moviendo un disco a la vez y sin nunca colocar un disco más grande sobre uno más pequeño.



```
def hanoi(n: int, src: int, dst: int, aux: int):
    if n == 0:
        return
    hanoi(n - 1, src, aux, dst)
    mover(src, dst) # Mover el disco de arriba de la pila src a la pila dst.
    hanoi(n - 1, aux, dst, src)

hanoi(n, 0, 2, 1)
```

Midamos el número de movimientos que realiza `hanoi`. Siguiendo los pasos de arriba:

1. Caso base. Cuando $n = 0$, no hacemos ningún movimiento. Luego $T(0) = 0$.
2. Llamadas recursivas. Hay dos llamadas a `hanoi` con argumento $n - 1$. Cada una contribuye $T(n - 1)$ movimientos.
3. Trabajo no recursivo. Una llamada a `mover`, que cuesta 1 movimiento.

Ensamblando, obtenemos la recurrencia $T(0) = 0$ y $T(n) = 2T(n - 1) + 1$ para $n \geq 1$.

Expandiendo los primeros valores, $T(1) = 1$, $T(2) = 3$, $T(3) = 7$, $T(4) = 15$, sospechamos que $T(n) = 2^n - 1$. Probémoslo por inducción.

- $T(0) = 0 = 2^0 - 1$. ✓
- Si $T(n - 1) = 2^{n-1} - 1$, entonces $T(n) = 2T(n - 1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$. ✓

Luego, $T(n) = 2^n - 1$ para todo $n \in \mathbb{N}$, y luego $T \in \Theta(2^n)$.

2.4 Peor caso, mejor caso, y caso promedio

Como la función de costo depende de la entrada específica, y puede haber muchas entradas del mismo tamaño, definimos funciones que agregan el costo sobre todas las entradas de un tamaño dado. Esto nos deja razonar sobre el comportamiento de nuestro algoritmo a medida que el tamaño de la entrada crece, en vez de preocuparnos por la entrada específica.

Definición 17

Sea A un algoritmo, $|\cdot|$ una función de tamaño, y T_A una función de costo. Definimos:

- **Peor caso:** $T_{\max}(n) = \max\{T_A(x) : |x| = n\}$
- **Mejor caso:** $T_{\min}(n) = \min\{T_A(x) : |x| = n\}$
- **Caso promedio:** $T_{\text{avg}}(n) = \mathbb{E}_{x \sim D_n}[T_A(x)]$

donde D_n es una distribución de probabilidad sobre las entradas de tamaño n .



El peor caso es el más usado, porque nos da una **garantía**: sin importar qué entrada x recibamos, el algoritmo no tardará más que $T_{\max}(|x|)$. Esto es útil para sistemas donde necesitamos predecir tiempos de respuesta.

El mejor caso rara vez se usa solo, porque es demasiado optimista. Sin embargo, es útil para establecer *cotas inferiores* para problemas. Por ejemplo, uno puede demostrar que todo algoritmo que sume n enteros es tal que $T_{\min} \in \Omega(n)$.

El caso promedio es útil cuando las entradas vienen de una distribución conocida, o cuando queremos entender el comportamiento «típico». Sin embargo, tiene una sutileza importante:

⚠️ Advertencia

El caso promedio requiere especificar una distribución D_n sobre las entradas. Distintas distribuciones dan distintos promedios. Cuando no se especifica, se suele asumir la distribución uniforme sobre todas las entradas de tamaño n , pero esto no siempre refleja la realidad.

Por ejemplo, para Quicksort con pivote fijo, el caso promedio bajo distribución uniforme es $\Theta(n \log n)$, pero si las entradas reales tienden a estar casi ordenadas, el tiempo va a estar cerca de n^2 .

Veamos un ejemplo concreto para ilustrar estos conceptos.

Ejercicio 3.2.7

Consideremos el siguiente algoritmo de búsqueda lineal:

```
def buscar(lista: list[int], x: int) -> int:
    for i in range(len(lista)):
        if lista[i] == x:
            return i
    return -1
```

Definimos el costo del algoritmo como el número de comparaciones que hace el algoritmo al buscar x en la lista L . Definimos el tamaño de la entrada como la longitud de la lista.

Analizar el peor caso, mejor caso, y caso promedio del número de comparaciones.



Solución.

- Mejor caso. Si x está en la primera posición, hacemos una sola comparación. Luego $T_{\min}(n) = 1$ y luego $T_{\min} \in \Theta(1)$.
- Peor caso. Si x no está en la lista, o está en la última posición, hacemos n comparaciones. Luego $T_{\max}(n) = n$, y luego $T_{\max} \in \Theta(n)$.
- Caso promedio. Asumamos que x está en la lista, y que está en cada posición con igual probabilidad $\frac{1}{n}$. Si x está en la posición i (indexando desde 1), hacemos i comparaciones. Entonces:

$$T_{\text{avg}}(n) = \sum_{i=1}^n i \cdot \frac{1}{n} = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Luego $T_{\text{avg}} \in \Theta(n)$. Si en cambio asumimos que x puede no estar en la lista con probabilidad fija $p \in [0, 1]$, el análisis cambia:

$$T_{\text{avg}}(n) = p \cdot n + (1-p) \cdot \frac{n+1}{2} = n \left(p + \frac{1-p}{2} \right) + \frac{1-p}{2} = n \left(\frac{1+p}{2} \right) + \frac{1-p}{2}$$

Luego $T_{\text{avg}} \in \Theta(n)$. En ambos casos, el promedio es $\Theta(n)$, pero las constantes son distintas.

2.5 Del algoritmo a la función asintótica

Juntando lo que aprendimos sobre análisis asintótico con lo que aprendimos sobre algoritmos, el proceso completo de análisis de un algoritmo A tiene los siguientes pasos:

1. **Fijar el modelo de cómputo.** Usualmente el modelo RAM, implícitamente.
2. **Definir el tamaño de entrada.** ¿Qué significa n para este problema? ¿Es el número de elementos de una lista, el máximo elemento de un conjunto, el número de dígitos de un entero dado, el número de aristas en un árbol, etc? Si hay dos variables, ¿el tamaño de entrada es $n+m$? ¿Es $n \times m$? ¿Es n^{3+m} ?
3. **Definir la función de costo** $T_A(x)$ para cada entrada x . Elegir qué recurso medir (tiempo, espacio, comparaciones, etc.) y definir $T_A(x)$ como la cantidad de ese recurso que consume A al recibir la entrada x . La función de costo va a ser inducida por el código del algoritmo.
4. **Agregar por tamaño.** Elegir peor caso, mejor caso, o promedio de T_A para obtener una función $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Notemos como estas son precisamente las funciones para las cuales aprendimos a hacer análisis asintótico.
5. **Analizar asintóticamente.** Usar las herramientas de la sección anterior (O , Ω , Θ , límites, árboles de recursión, teorema maestro) para caracterizar T .

Nota

Es importante ser preciso al reportar. No es lo mismo decir:

- « A es $O(n^2)$ » (ambiguo: ¿qué estamos midiendo? ¿peor caso?)
- « A corre en tiempo $\Omega(n^2)$ en el peor caso» (claro)
- «El número de comparaciones de A es $\Theta(n \log n)$ en el peor caso» (muy claro)

Ejemplo 3.2.8

Consideremos el siguiente algoritmo de búsqueda binaria, que busca un elemento x en una lista ordenada:

```
def busqueda_binaria(lista: list[int], x: int) -> int:
    lo = 0
    hi = len(lista)
    while lo < hi:
        mid = (lo + hi) // 2
        if lista[mid] == x:
            return mid
        elif lista[mid] < x:
            lo = mid + 1
        else:
            hi = mid
    return -1
```

El invariante del ciclo es que si x está en la lista, entonces está en $\text{lista}[lo:hi]$. Definimos el tamaño de la entrada como n , la longitud de la lista. Sea $C(x)$ el número de comparaciones entre elementos que realiza `busqueda_binaria` sobre la entrada x . Analizar el peor caso y el mejor caso de C .

Solución. Siguiendo los pasos del proceso de análisis:

1. **Modelo de cómputo:** RAM.
2. **Tamaño de entrada:** n , la longitud de la lista.
3. **Función de costo:** $C(x)$ es el número de comparaciones entre elementos que realiza `busqueda_binaria` sobre la entrada x .
4. **Agregación por tamaño:** peor caso, $T_{\max}(n)$, y mejor caso, $T_{\min}(n)$.
5. **Análisis asintótico:**
 - Mejor caso. Si `lista[mid] == x` en la primera iteración, hacemos una sola comparación. Luego $T_{\min}(n) = 1$, y luego $T_{\min} \in \Theta(1)$.
 - Peor caso. El intervalo $[lo, hi]$ comienza con n elementos. En cada iteración donde no encontramos x , se reemplaza `lo` por `mid + 1` o `hi` por `mid`, reduciendo el intervalo de $hi - lo$ elementos a lo sumo $\lfloor \frac{hi - lo}{2} \rfloor$ elementos. El ciclo termina cuando el intervalo se vacía ($lo \geq hi$) o cuando encontramos x . En el peor caso (x no está en la lista), el ciclo ejecuta $\lfloor \log_2 n \rfloor + 1$ iteraciones, cada una con a lo sumo 2 comparaciones.

Tras k iteraciones, el intervalo tiene a lo sumo $\lfloor \frac{n}{2^k} \rfloor$ elementos. El ciclo termina cuando el intervalo se vacía, lo cual requiere $\lfloor \frac{n}{2^k} \rfloor = 0$, es decir $2^k > n$, por lo que el número de iteraciones es $\lfloor \log_2 n \rfloor + 1$. Como cada iteración realiza a lo sumo 2 comparaciones, $T_{\max}(n) \leq 2(\lfloor \log_2 n \rfloor + 1)$. Recíprocamente, cuando x no está en la lista, el ciclo ejecuta las $\lfloor \log_2 n \rfloor + 1$ iteraciones, cada una con al menos 1 comparación, con lo cual $T_{\max}(n) \geq \lfloor \log_2 n \rfloor + 1$. Luego, $T_{\max} \in \Theta(\log n)$.

También podemos pensar en `busqueda_binaria` como un algoritmo recursivo implícito: cada iteración resuelve el problema en un subintervalo de tamaño $\lfloor \frac{n}{2} \rfloor$, sumando $O(1)$ costo adicional (una comparación). Esto nos da la recurrencia $T_{\max}(n) = T_{\max}(\lfloor \frac{n}{2} \rfloor) + O(1)$. Aplicando el teorema maestro con $a_1 = 1$, $a_2 = 0$, $b = 2$, y $f \in O(1)$, tenemos $c =$

$\log_2 1 = 0$, y $f \in \Theta(n^0 \log^0 n) = \Theta(1)$. Caemos en el segundo caso con $k = 0$, y concluimos que $T_{\max} \in \Theta(\log n)$.

Ejemplo 3.2.9

Mergesort es un algoritmo de ordenamiento a base de comparaciones.

```
def merge(left: list[int], right: list[int]) -> list[int]:
    result = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

def mergesort(arr: list[int]) -> list[int]:
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])
    return merge(left, right)
```

Dada una lista de enteros x , sea $C(x)$ el número de comparaciones que hace `mergesort(x)`. Se define $T : \mathbb{N} \rightarrow \mathbb{N}$ como $T(n) = \max_x \{C(x) \mid \text{len}(x) = n\}$. Es decir, el máximo número de comparaciones que realiza `mergesort(x)`, entre todas las listas x tales que `len(x) = n`.

Probar que $T \in O(n \log n)$.

Solución. Sigamos los pasos del proceso de análisis para analizar el comportamiento de `mergesort` en el peor caso.

1. **Modelo de cómputo:** RAM.
2. **Tamaño de entrada:** n , la longitud de la lista.
3. **Función de costo:** sea $C(x)$ el número de comparaciones que realiza `mergesort(x)`.
4. **Agregación por tamaño:** peor caso. Definimos $T(n) = \max\{C(x) : \text{len}(x) = n\}$.
5. **Análisis asintótico:** `mergesort` divide la lista en dos mitades de tamaños $\lfloor \frac{n}{2} \rfloor$ y $\lceil \frac{n}{2} \rceil$, las ordena recursivamente, y luego las combina con `merge`. La función `merge` recorre ambas listas de izquierda a derecha, haciendo una comparación por paso, y en cada paso avanza en una de las dos listas. Como las listas tienen n elementos en total, `merge` hace a lo sumo $n - 1 \in O(n)$ comparaciones (el último elemento no necesita ser comparado). Esto nos da la recurrencia:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

Aplicamos el teorema maestro con $a_1 = 1$, $a_2 = 1$ (luego $a = 2$), $b = 2$, y $f \in O(n)$. Tenemos $c = \log_2 2 = 1$, y $f \in \Theta(n^1 \log^0 n) = \Theta(n)$. Caemos en el segundo caso del teorema maestro con $k = 0$, y concluimos que $T \in \Theta(n \log n)$.

Ejemplo 3.2.10

Consideremos el siguiente algoritmo que determina si una lista está ordenada:

```
def is_sorted(lista: list[int], n: int) -> bool:
    for i in range(n - 1):
        if lista[i] > lista[i + 1]:
            return False
    return True
```

Sea $C(x)$ el número de comparaciones que realiza `is_sorted` sobre la entrada x , y n la longitud de la lista. Analizar el peor caso, mejor caso, y caso promedio del número de comparaciones, asumiendo para el caso promedio que la entrada es una permutación uniformemente al azar de $\{1, \dots, n\}$.

Solución.

- Mejor caso. Si $\text{lista}[0] > \text{lista}[1]$, el algoritmo retorna en la primera iteración, habiendo hecho una sola comparación. Luego $T_{\min}(n) = 1$, y luego $T_{\min} \in \Theta(1)$.
- Peor caso. Si la lista está ordenada, el algoritmo recorre todo el ciclo sin retornar temprano, realizando $n - 1$ comparaciones. Luego $T_{\max}(n) = n - 1$, y luego $T_{\max} \in \Theta(n)$.
- Caso promedio. Para cada $i \in \{0, \dots, n - 2\}$, definimos la variable indicadora X_i , que vale 1 si el algoritmo realiza la comparación en la iteración i , y 0 si ya retornó antes. El número total de comparaciones es $C = \sum_{i=0}^{n-2} X_i$.

El algoritmo llega a la iteración i si y sólo si no encontró ningún descenso en las iteraciones anteriores, es decir, si $\text{lista}[0] < \text{lista}[1] < \dots < \text{lista}[i]$. Como la entrada es una permutación uniformemente al azar, los primeros $i + 1$ elementos están en alguno de los $(i + 1)!$ órdenes posibles, todos equiprobables. Exactamente uno de esos órdenes es el creciente, luego $\mathbb{E}[X_i] = \frac{1}{(i+1)!}$.

Por linealidad de la esperanza:

$$T_{\text{avg}}(n) = \mathbb{E}[C] = \sum_{i=0}^{n-2} \mathbb{E}[X_i] = \sum_{i=0}^{n-2} \frac{1}{(i+1)!} = \sum_{k=1}^{n-1} \frac{1}{k!}$$

Como $\sum_{k=0}^{\infty} \frac{1}{k!} = e$, tenemos que $\sum_{k=1}^{n-1} \frac{1}{k!} \leq e - 1 < 2$. A su vez, $T_{\text{avg}}(n) \geq 1$ para todo $n \geq 2$, porque siempre hacemos al menos una comparación. Luego $T_{\text{avg}} \in \Theta(1)$.

El caso promedio está en $\Theta(1)$, dramáticamente distinto del peor caso, que está en $\Theta(n)$. Intuitivamente, una permutación al azar tiene un descenso muy temprano con alta

probabilidad: sólo una fracción $\frac{1}{k!}$ de las permutaciones tiene sus primeros k elementos en orden creciente, y esa fracción decrece extremadamente rápido.

Ejemplo 3.2.11

Se tienen tres algoritmos que resuelven el mismo problema. Todos realizan algún número positivo de operaciones cuando su entrada tiene tamaño cero.

1. El primero divide un problema de tamaño n en 5 subproblemas de tamaño $\frac{n}{2}$ cada uno, y combina sus soluciones en $O(n)$ operaciones.
2. El segundo divide un problema de tamaño n en 2 subproblemas de tamaño $n - 1$ cada uno, y combina sus soluciones en $O(1)$ operaciones.
3. El tercero divide un problema de tamaño n en 9 subproblemas de tamaño $\frac{n}{3}$ cada uno, y combina sus soluciones en $\Theta(n^2)$ operaciones.

Si nuestro n es enorme, y queremos minimizar el número de operaciones que requiere encontrar una solución, podemos determinar cuál algoritmo nos conviene usar sólo sabiendo esto?



Para esta demostración les voy a escribir el razonamiento que hago mientras escribo la demostración.

El primer caso es fácil, $\log_2(5) > 1$, y el costo de combinar soluciones es pequeño porque $1 < \log_2(5)$, entonces el costo está dominado por las hojas del árbol de recursión, y uso el teorema maestro para saber que esto es $\Theta(n^{\log_2(5)})$.

Para el segundo esto me recuerda a la función $2^n = 2 \times 2^{n-1}$, o también a la sucesión de Fibonacci, $F_n = F_{n-1} + F_{n-2}$. Esas tienen solución exponencial, entonces supongamos que $T(n) \leq \gamma c^n$ para algún $\gamma, c \in \mathbb{R}_{>0}$. $T(0)$ es alguna constante, pongámosle $T(0) = \alpha$. Entonces si quiero que $T(0) \leq \gamma c^0 = \gamma$, voy a tener $\alpha \leq \gamma$, y luego mi γ tiene que cumplir $\gamma \geq \alpha$. Como hay un $O(1)$ en la definición de T , sea $h : \mathbb{N} \rightarrow \mathbb{N}$ tal que $T(n) = 2T(n - 1) + h(n)$, existen $n_0 \in \mathbb{N}, \beta \in \mathbb{R}_{>0}$ tal que para todo $n \geq n_0$, $h(n) \leq \beta$. Veamos qué puedo deducir sobre γ usando β ...

$$\begin{aligned} T(n) &= 2T(n - 1) + h(n) \\ &\leq 2T(n - 1) + \beta \\ &\leq 2\gamma c^{n-1} + \beta \end{aligned}$$

Entonces, si quiero probar que $T(n) \leq \gamma c^n$, tengo que probar que $2\gamma c^{n-1} + \beta \leq \gamma c^n$, y por transitividad de \leq está. Voy a encontrar γ y c que hagan cierto esto. Moralmente espero que $c \approx 2$ porque esa es la solución cuando $\alpha = 1, \beta = 0$ (y $T(n) = 2^n$ para todo n ahí).

$$2\gamma c^{n-1} + \beta \leq \gamma c^n$$

A ver qué pasa si pongo $\gamma = \beta$...

$$\begin{aligned} 2\beta c^{n-1} + \beta &\leq \beta c^n \\ 2c^{n-1} + 1 &\leq c^n \\ 1 &\leq c^{n-1}(c - 2) \end{aligned}$$

pero eso no va a ser cierto si $c = 2$. Todavía no sé que $c = 2$, pero puede ser.

Expandamos un poco, asumiendo n grande (bastante más grande que n_0 para poder usar β)...

$$\begin{aligned} T(n) &\leq 2T(n-1) + \beta \\ &\leq 2(2T(n-2) + \beta) + \beta \\ &= 4T(n-2) + 2\beta + \beta \\ &\leq 4(2T(n-3) + \beta) + 2\beta + \beta \\ &= 8T(n-3) + 4\beta + 2\beta + \beta \\ &\dots, \end{aligned}$$

lo siguiente es falso porque la cota de $h(n) \leq \beta$ sólo vale para $n > n_0$, pero estoy siendo informal...

$$\begin{aligned} &\leq 2^n T(0) + \sum_{i=0}^{n-1} 2^i \beta \\ &= 2^n \alpha + \beta(2^n - 1) \\ &= 2^n(\alpha + \beta) - \beta \end{aligned}$$

OK, eso me facilita las cosas, puedo usar $\gamma = \alpha + \beta$, y la forma que va a tener la cota es $\gamma 2^n - \beta$, no sólo $\gamma 2^n$. Después lo formalizo.

Eso igual sólo me da una cota superior. Tendría que también argumentar una cota inferior para saber bien quién es ese T y poder compararlo.

Para el tercero, tengo $T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n^2)$. Como $\log_3(9) = 2$, no tenemos que $2 < 2$, entonces tenemos que usar el segundo caso del teorema maestro, que con $a = 9, b = 3, k = 0$, nos dice que $T \in \Theta(n^2 \log n)$.

Solución. Calculemos el comportamiento asintótico de cada uno. Vamos a llamar $T_1, T_2, T_3 : \mathbb{N} \rightarrow \mathbb{N}$ a las funciones que, dado un $n \in \mathbb{N}$, nos dicen cuántas operaciones toma cada algoritmo para resolver un problema de tamaño n . Como cada algoritmo está descrito usando $O(\dots)$, no vamos a poder encontrar quiénes son exactamente, sino que vamos a conocer su comportamiento asintótico. Como nos dicen que n es enorme, esto es realmente lo único que importa, porque los factores constantes no van a importar si n es enorme.

1. El primer algoritmo cumple que $T_1(n) = 5T_1\left(\frac{n}{2}\right) + O(n)$. Usando el teorema maestro con $a = 5, b = 2, f \in O(n)$, tenemos que $c = \log_2(5) > \log_2(4) = 2 > 1$, donde 1 es el exponente polinomial de f , por estar en $O(n^1)$. Como $c > 1$, entonces caemos en el primer caso del teorema maestro, el cual nos dice que $T_1 \in \Theta(n^{\log_2(5)})$.
2. Como $T_2(n)$ está definido en términos de $T_2(n-1)$, vamos a usar inducción para ver quién es T_2 . La recurrencia que cumple T_2 es $T_2(n) = 2T_2(n-1) + O(1)$. Sea $\alpha = T_2(0)$. Como $T_2(n) = 2T_2(n-1) + O(1)$, ese $O(1)$ nos dice que existe una función $h : \mathbb{N} \rightarrow \mathbb{N}, h \in O(1)$, tal que $T_2(n) = 2T_2(n-1) + h(n)$ para todo $n \geq 0$. Como $h \in O(1)$, sean $n_0 \in \mathbb{N}, \beta > 0 \in \mathbb{R}$, tales que para todo $n \geq n_0, h(n) \leq \beta$. Sea $\delta = \max\left(\beta, \max_{0 \leq i \leq n_0} h(i)\right)$, y por lo tanto $h(k) \leq \delta$ para todo $k \in \mathbb{N}$. Sea $\gamma = \alpha + \delta$.

Vamos a probar $P(n) : \alpha 2^n \leq T_2(n) \leq \gamma 2^n - \delta$.

1. Caso base, $P(0)$. Por definición de α , $T_2(0) = \alpha = \alpha + \delta - \delta = \gamma - \delta = \gamma 2^0 - \delta$, que junto con $\alpha 2^0 = \alpha$, prueba $P(0)$.

2. Paso inductivo. Asumimos $P(n)$, queremos probar $P(n + 1)$. Sabemos que $T_2(n + 1) = 2T_2(n) + h(n)$.

$$\begin{aligned} T_2(n + 1) &= 2T_2(n) + h(n) \\ &\leq 2T_2(n) + \delta \\ &\leq 2(\gamma 2^n - \delta) + \delta \\ &= \gamma 2^{n+1} - 2\delta + \delta \\ &= \gamma 2^{n+1} - \delta \end{aligned}$$

Asimismo,

$$\begin{aligned} T_2(n + 1) &= 2T_2(n) + h(n) \\ &\geq 2T_2(n) \\ &\geq 2(\alpha 2^n) \\ &\geq \alpha 2^{n+1} \end{aligned}$$

y concluimos que $\alpha 2^{n+1} \leq T_2(n + 1) \leq \gamma 2^{n+1} - \delta$, que es lo que queríamos demostrar, $P(n + 1)$.

Luego, como $\alpha > 0$ porque nos lo dice el enunciado, y sabemos que $T_2(n) \leq \gamma 2^n$ (descartando el término $-\delta$ por transitividad), tenemos que $T_2 \in O(2^n)$ y $T_2 \in \Omega(2^n)$, con lo cual $T_2 \in \Theta(2^n)$.

3. Para el tercer caso, usamos el segundo caso del teorema maestro, con $a = 9$, $b = 3$, $k = 0$, y concluimos que $T_3 \in \Theta(n^2 \log n)$.

Sabemos entonces que el número de operaciones que estos tres algoritmos hacen, ante una entrada de tamaño n , está respectivamente en $\Theta(n^{\log_2(5)})$, $\Theta(2^n)$, y $\Theta(n^2 \log n)$.

Veamos cuál nos conviene usar. Recordando que $g \in O(f) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, y usando las reglas usuales de orden de conjuntos asintóticos:

- Como $T_2 \in \Omega(2^n)$, entonces llamando $f(n) = 2^n$, tenemos que $f \in O(T_2)$. Luego, como $T_1 \in O(n^{\log_2(5)})$, y $O(n^{\log_2(5)}) \subset O(f)$, tenemos que $T_1 \in O(f)$. Como $T_1 \in O(f)$ y $f \in O(T_2)$, tenemos que $T_1 \in O(T_2)$.
- Usando la regla de L'Hopital, tenemos que

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n^{\log_2(5)}}{n \log n} &= \lim_{n \rightarrow \infty} \frac{n^{\log_2(5)-1}}{\log n} \\ &= \lim_{n \rightarrow \infty} \frac{n^\varepsilon}{\log n} \\ &= \lim_{n \rightarrow \infty} n^\varepsilon n^{\varepsilon-1} \\ &= \lim_{n \rightarrow \infty} \varepsilon n^\varepsilon \\ &= \infty \end{aligned}$$

Pues $\log_2(5) - 1 = \varepsilon > 0$. Luego tenemos que $T_3 \in O(T_1)$.

- Como $T_3 \in O(T_1)$ y $T_1 \in O(T_2)$, tenemos que $T_3 \in O(T_2)$.

Luego, como T_3 está asintóticamente dominada por las otras dos, y n es enorme, nos conviene usar el tercer algoritmo.

Ejemplo 3.2.12

Un entero de precisión arbitraria se puede representar como una lista de n palabras de w bits cada una, donde w es el tamaño de palabra de la máquina. Por ejemplo, el entero $2^{65} + 3$ se puede representar, en una máquina con $w = 64$, como la lista $[3, 2]$, donde $\text{words}[0] = 3$ contiene los 64 bits menos significativos y $\text{words}[1] = 2$ contiene los siguientes 64 bits.

El *popcount* (population count) de un entero es la cantidad de bits en 1 en su representación binaria. Consideremos el siguiente algoritmo en el modelo word RAM con palabras de w bits, donde las operaciones bit a bit ($\&$, $>>$) sobre una palabra toman $O(1)$ cada una:

```
def popcount(words: list[int]) -> int:
    count = 0
    for word in words:
        while word > 0:
            count += word & 1
            word >>= 1
    return count
```

Definimos el tamaño de la entrada como n , el número de palabras. Sea $C(\text{words})$ el número de operaciones que realiza *popcount*(words). Analizar el peor caso y el mejor caso de C .

Solución. Siguiendo los pasos del proceso de análisis:

1. **Modelo de cómputo:** word RAM con palabras de w bits. Las operaciones aritméticas y bit a bit sobre palabras de w bits toman $O(1)$.
2. **Tamaño de entrada:** n , el número de palabras.
3. **Función de costo:** $C(\text{words})$, el número de operaciones que realiza *popcount*(words).
4. **Agregación por tamaño:** peor caso, $T_{\max}(n)$, y mejor caso, $T_{\min}(n)$.
5. **Análisis asintótico:**
 - Mejor caso. Si todas las palabras son 0, el ciclo interno no se ejecuta nunca, y el ciclo externo hace una comparación por palabra. Luego $T_{\min} \in \Theta(n)$.
 - Peor caso. En cada iteración del ciclo interno, se realizan una comparación ($\text{word} > 0$), una operación bit a bit ($\text{word} \& 1$), una suma, y un shift ($\text{word} >>= 1$). En el modelo word RAM, cada una toma $O(1)$, por lo que cada iteración del ciclo interno toma $O(1)$. Como cada palabra tiene a lo sumo w bits, el ciclo interno ejecuta a lo sumo w iteraciones por palabra. En el peor caso (todas las palabras tienen todos sus bits en 1), el ciclo interno ejecuta exactamente w iteraciones por cada una de las n palabras. Luego, $T_{\max} \in \Theta(nw)$.

Ejemplo 3.2.13

Consideremos la siguiente implementación recursiva de la sucesión de Fibonacci:

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Definimos el tamaño de la entrada como n . Sea $T(n)$ el número de operaciones que realiza `fib(n)`, y sea $S(n)$ la máxima profundidad de la pila de llamadas durante la ejecución de `fib(n)` (es decir, la máxima cantidad de llamadas a `fib` simultáneamente activas). Analizar el comportamiento asintótico de T y de S .

Solución.

1. **Modelo de cómputo:** RAM.
2. **Tamaño de entrada:** n .
3. **Funciones de costo:** $T(n)$, el número de operaciones, y $S(n)$, la máxima profundidad de pila.
4. **Análisis asintótico:**

- **Número de operaciones.** Cada llamada a `fib` realiza, sin contar sus llamadas recursivas, un número de operaciones acotado: al menos 1 (la comparación $n \leq 1$) y a lo sumo alguna constante $c > 0$ (la comparación, la resta, y la suma). Tenemos la recurrencia $T(n) = T(n - 1) + T(n - 2) + h(n)$ para $n \geq 2$, con $T(0)$ y $T(1)$ constantes positivas, donde $1 \leq h(n) \leq c$ para todo n .

Sea $\varphi = \frac{1+\sqrt{5}}{2}$. Recordemos que φ satisface $\varphi^2 = \varphi + 1$.

Cota inferior. Probemos por inducción que $T(n) \geq \varphi^{n-1}$ para todo $n \geq 0$.

1. $T(0) \geq 1 > \frac{1}{\varphi} = \varphi^{-1}$. ✓
2. $T(1) \geq 1 = \varphi^0$. ✓
3. Para $n \geq 2$, $T(n) = T(n - 1) + T(n - 2) + h(n) \geq T(n - 1) + T(n - 2) \geq \varphi^{n-2} + \varphi^{n-3} = \varphi^{n-3}(\varphi + 1) = \varphi^{n-3}\varphi^2 = \varphi^{n-1}$. ✓

Luego $T \in \Omega(\varphi^n)$.

Cota superior. Sea $\beta = \max(T(0) + c, \frac{T(1)+c}{\varphi})$. Probemos por inducción que $T(n) \leq \beta\varphi^n - c$ para todo $n \geq 0$.

1. $T(0) \leq \beta - c$, ya que $\beta \geq T(0) + c$. ✓
2. $T(1) \leq \beta\varphi - c$, ya que $\beta \geq \frac{T(1)+c}{\varphi}$. ✓
3. Para $n \geq 2$:

$$\begin{aligned} T(n) &= T(n - 1) + T(n - 2) + h(n) \\ &\leq (\beta\varphi^{n-1} - c) + (\beta\varphi^{n-2} - c) + c \\ &= \beta(\varphi^{n-1} + \varphi^{n-2}) - c \\ &= \beta\varphi^{n-2}(\varphi + 1) - c \\ &= \beta\varphi^n - c \end{aligned}$$

Luego $T(n) \leq \beta\varphi^n$ y $T \in O(\varphi^n)$. Combinando, $T \in \Theta(\varphi^n)$.

- **Profundidad de pila.** En la expresión `fib(n - 1) + fib(n - 2)`, la llamada `fib(n - 1)` se ejecuta completamente (y su frame se elimina de la pila) antes de que comience `fib(n - 2)`. Luego, la pila en cualquier momento contiene los frames a lo largo de un único camino desde la raíz hasta el nodo actual en el árbol de recursión.

Formalmente, $S(0) = S(1) = 1$ y $S(n) = 1 + \max(S(n - 1), S(n - 2))$ para $n \geq 2$. Notemos que S es creciente: como $\max(S(n - 1), S(n - 2)) \geq S(n - 1)$, tenemos $S(n) \geq 1 + S(n - 1) > S(n - 1)$ para todo $n \geq 2$, y $S(1) \geq S(0)$. Luego, $\max(S(n - 1), S(n - 2)) = S(n - 1)$ y $S(n) = 1 + S(n - 1)$ para todo $n \geq 2$, de donde $S(n) = n$ para $n \geq 1$. Luego, $S \in \Theta(n)$.

Notar el contraste: el número de operaciones es exponencial ($\Theta(\varphi^n)$), pero la profundidad de pila es lineal ($\Theta(n)$).

Ejemplo 3.2.14

Consideremos la criba de Eratóstenes, que calcula todos los primos hasta n :

```
def sieve(n: int) -> list[bool]:
    is_prime = [True] * (n + 1)
    is_prime[0] = False
    is_prime[1] = False
    for i in range(2, n + 1):
        if is_prime[i]:
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
    return is_prime
```

Definimos el tamaño de la entrada como n . Sea $T(n)$ el número de operaciones que realiza `sieve(n)`. Analizar el comportamiento asintótico de T .

Ayuda: se sabe por [21] que si definimos $g(x) = \sum_{\substack{p \leq x, \\ p \text{ primo}}} \frac{1}{p}$, entonces $g \in \Theta(\log \log x)$.

Solución.

1. **Modelo de cómputo:** RAM.
2. **Tamaño de entrada:** n .
3. **Función de costo:** $T(n)$, el número de operaciones que realiza `sieve(n)`.
4. **Análisis asintótico:**

La inicialización del arreglo toma $O(n)$. El ciclo externo itera $n - 1$ veces, haciendo $O(1)$ trabajo por iteración (verificar `is_prime[i]`). El ciclo interno solo se ejecuta cuando i es primo, y como comienza en i^2 , solo tiene iteraciones cuando $i \leq \sqrt{n}$. Para cada primo $p \leq \sqrt{n}$, el ciclo interno recorre los múltiplos de p desde p^2 hasta n en pasos de p , haciendo $\left\lfloor \frac{n-p^2}{p} \right\rfloor + 1 \leq \frac{n}{p}$ iteraciones.

El número total de operaciones del ciclo interno es:

$$\sum_{\substack{p \leq \sqrt{n}, \\ p \text{ primo}}} \frac{n}{p} = n \sum_{\substack{p \leq \sqrt{n}, \\ p \text{ primo}}} \frac{1}{p} \in \Theta(n \log \log n)$$

pues $\sum_{p \leq \sqrt{n}} \frac{1}{p} \in \Theta(\log \log \sqrt{n}) = \Theta(\log \log n)$, ya que $\log \log \sqrt{n} = \log \left(\frac{1}{2} \log n \right) = \log \log n - \log 2 \in \Theta(\log \log n)$.

Sumando el $O(n)$ de la inicialización y el ciclo externo, que está dominado por $\Theta(n \log \log n)$ para n suficientemente grande, concluimos que $T \in \Theta(n \log \log n)$.

Ejemplo 3.2.15

Se tiene el siguiente algoritmo recursivo:

```
def f(cuts: list[int], i: int, j: int) -> int:
    if j == i + 1:
        return 0
    r = inf
    for k in range(i + 1, j):
        r = min(r, f(cuts, i, k) + f(cuts, k, j))
    return r + cuts[j] - cuts[i]
```

Dar una cota asintótica superior ajustada para el número de operaciones que realice este algoritmo, en función del tamaño de entrada $j - i$.



Solución. Definimos el tamaño de una entrada $(cuts, i, j)$ como $j - i$. La función que describe el tiempo que toma f en correr, en términos del tamaño de entrada, es $T(n) = O(1) + \sum_{k=1}^{n-1} T(k) + T(n - k)$.

Obviamente no podemos usar el teorema maestro para esto, porque T no tiene la forma correcta, así que vamos a tener que analizar cuidadosamente qué está pasando. Jugando un poco, vemos que:

$$\begin{aligned} T(n) &= O(1) + T(1) + T(n - 1) + T(2) + T(n - 2) + \dots + T(n - 1) + T(1) \\ &= O(1) + 2 \sum_{i=1}^{n-1} T(i) \end{aligned}$$

Esto nos puede recordar a la fórmula para las potencias de un número. Por ejemplo, $2^n = 1 + \sum_{i=0}^{n-1} 2^i$, o $3^n = 1 + 2 \sum_{i=0}^{n-1} 3^i$. Como esto se parece bastante a la serie de potencias de tres, vamos a intentar adivinar que $T(n) \leq \alpha 3^n$ para todo $n \in \mathbb{N}$, y algún $\alpha > 0 \in \mathbb{R}$. Esto nos diría que $T \in O(3^n)$.

Probemos esto por inducción. Por definición de « $O(1)$ », sabemos que existe una función $h : \mathbb{N} \rightarrow \mathbb{N}$, tal que $T(n) = h(n) + 2 \sum_{i=1}^{n-1} T(i)$, y existen $n_0 \in \mathbb{N}$, $\beta > 0$ tales que para todo $n \geq n_0$, $h(n) \leq \beta$. Sea $r = \max(\beta, \max_{i=0}^{n_0} h(i))$. Entonces tenemos que para todo n , $h(n) \leq r$. La cota vale para los primeros n_0 valores de n por la segunda rama del max, y vale para todos los valores después de n_0 por la primera rama, que a su vez vale por la definición de $h \in O(1)$.

Luego, para todo n , $T(n) \leq r + 2 \sum_{i=1}^{n-1} T(i)$. Si esto tiene que ser menor o igual a $\alpha 3^n$, veamos quién tiene que ser α .

$$\begin{aligned} T(n) &\leq r + 2 \sum_{i=1}^{n-1} T(i) \\ &\leq r + 2 \sum_{i=1}^{n-1} \alpha 3^i \\ &\leq r + 2\alpha \left(\frac{1}{2}\right)(3^n - 3) \\ &\leq r + \alpha 3^n - 3\alpha \end{aligned}$$

Vamos a querer concluir que $T(n) \leq \alpha 3^n$. Luego, queremos que $r - 3\alpha = 0$, y luego $\alpha = \frac{r}{3}$. Verifiquemos que con ese α se cumple lo que queremos:

$$T(n) \leq r + \alpha 3^n - 3\left(\frac{r}{3}\right) = \alpha 3^n$$

Esto parece funcionar. Probémoslo por inducción, entonces. Sea $P(n) : n \geq 1 \Rightarrow T(n) \leq \alpha 3^n$

- $P(1) = h(1) \leq r = 3\alpha = 3^1\alpha$.
- Asumimos que $P(j)$ vale para todo $j < n$, queremos probar $P(n)$. $T(n) \leq r + 2 \sum_{i=1}^{n-1} T(i) \leq \alpha 3^n$ por el argumento de arriba, donde usamos la hipótesis inductiva para acotar cada $T(i)$ por $\alpha 3^i$.

Luego vemos que tomando $n_0 = 1$, $\alpha = \max\left(\beta, \frac{\max_{i=0}^{n_0} h(i)}{3}\right)$, tenemos que para todo $n \geq n_0$, $T(n) \leq \alpha 3^n$, y por lo tanto $T \in O(3^n)$.

2.6 Ejercicios

Ejercicio 3.2.16

El siguiente es el algoritmo de Strassen para multiplicar dos matrices de $n \times n$.

```
1: procedure STRASSEN( $A \in \mathbb{Z}^{n \times n}$ ,  $B \in \mathbb{Z}^{n \times n}$ )
2:   if  $n \leq n_0$  then
3:     return  $A \times B$ 
4:   end
5:    $m \leftarrow \frac{n}{2}$ 
6:    $\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \leftarrow \text{Split}(A, m)$ 
7:    $\begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \leftarrow \text{Split}(B, m)$ 
8:    $M_1 \leftarrow \text{Strassen}(A_{1,1} + A_{2,2}, B_{1,1} + B_{2,2})$ 
9:    $M_2 \leftarrow \text{Strassen}(A_{2,1} + A_{2,2}, B_{1,1})$ 
10:   $M_3 \leftarrow \text{Strassen}(A_{1,1}, B_{1,2} - B_{2,2})$ 
11:   $M_4 \leftarrow \text{Strassen}(A_{2,2}, B_{2,1} - B_{1,1})$ 
12:   $M_5 \leftarrow \text{Strassen}(A_{1,1} + A_{1,2}, B_{1,1})$ 
13:   $M_6 \leftarrow \text{Strassen}(A_{2,1} - A_{1,1}, B_{1,1} + B_{1,2})$ 
14:   $M_7 \leftarrow \text{Strassen}(A_{1,2} - A_{2,2}, B_{2,1} + B_{2,2})$ 
15:   $C_{1,1} \leftarrow M_1 + M_4 - M_5 + M_7$ 
16:   $C_{1,2} \leftarrow M_3 + M_5$ 
17:   $C_{2,1} \leftarrow M_2 + M_4$ 
18:   $C_{2,2} \leftarrow M_1 - M_2 + M_3 + M_6$ 
19:  return  $\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$ 
20: end
```

Sabiendo que el caso base usa un algoritmo cúbico para multiplicar matrices, y despreciando las operaciones necesarias para las sumas y restas que hace el algoritmo, cuántas operaciones realiza este algoritmo, al ser llamado con dos matrices de tamaño $n \times n$, con $n = 2^k$? Probar formalmente que este algoritmo necesita, en el peor caso, $O(n^{\log_2 7})$ operaciones, con $\log_2 7 < 3$.

3 Divide and conquer y programación dinámica

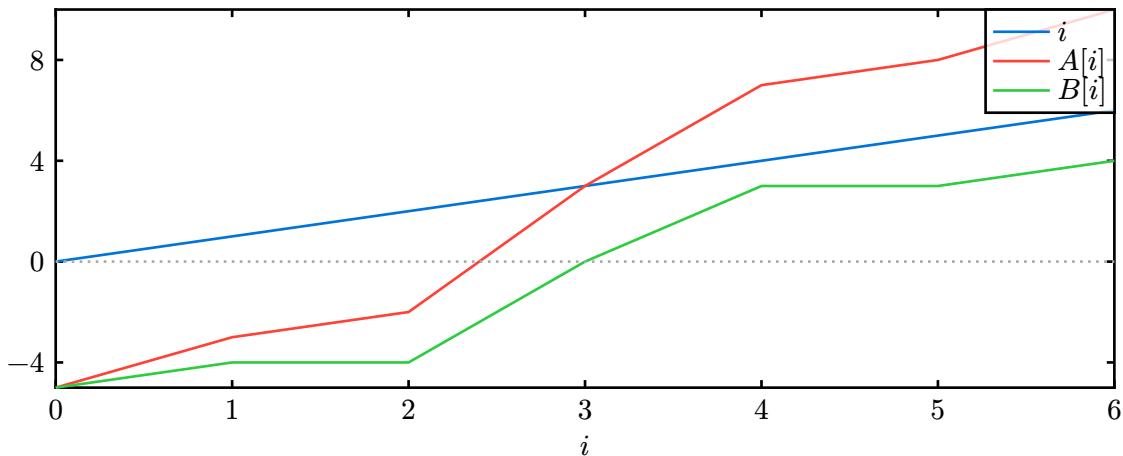
La computación está llena de algoritmos que se basan en dividir un problema en subproblemas más pequeños, resolver esos, y luego combinar los resultados. Estos en general van a tener demostraciones por inducción, donde hacemos inducción en el tamaño de los subproblemas que estamos resolviendo. Nuestra tarea es darle semántica al resultado del algoritmo, definir la noción de tamaño, y probar por inducción en el tamaño de una sub-solución, que el algoritmo es correcto con respecto a su semántica, para todos los subproblemas.



Ejercicio 3.3.1

Se tiene un array A de n enteros, ordenado de manera estrictamente creciente. Dar un algoritmo basado en divide and conquer que determine si existe una posición i tal que $A[i] = i$. Probar su correctitud, y determinar su complejidad asintótica temporal y espacial en el peor caso.

Solución. Podemos considerar la lista $B[i] = A[i] - i$, y vemos que $A[i] = i$ exactamente cuando $B[i] = 0$. Por ejemplo, para $A = (-5, -3, -2, 3, 7, 8, 10)$, tenemos $B = (-5, -4, -4, 0, 3, 3, 4)$, que se ve así:



Como A es una lista creciente de enteros, entonces si $i \in \mathbb{N}, i < n, A[i + 1] \geq A[i] + 1$. Restando i a ambos lados, obtenemos $A[i + 1] - i \geq A[i] + 1 - i$, o equivalentemente, $A[i + 1] - (i + 1) \geq A[i] - i$, que es $B[i + 1] \geq B[i]$, luego B es creciente.

Luego, nuestro algoritmo tiene que encontrar un i tal que $B[i] = 0$, con B creciente. Podemos usar búsqueda binaria para esto. Notemos que no hace falta *crear* B , dado que preguntar si $B[i] > 0$ es lo mismo que preguntar si $A[i] - i > 0 \Leftrightarrow A[i] > i$.

```
def f(A: list[int], i: int, j: int) -> bool:
    if j <= i: return False
    k = i + (j - i) // 2
    if A[k] > k:
        return f(A, i, k)
    elif A[k] < k:
        return f(A, k + 1, j)
    return True
```

La función se llama con $f(A, 0, \text{len}(A))$. El invariante, como en todas las búsquedas binarias, es que tenemos una propiedad $P(i, j)$: Si existe un k tal que $A[k] = k$, entonces $i \leq k < j$.

Demostración. La semántica que le vamos a dar a $f(A, i, j)$ es que es True si y sólo si existe un k en $[i, j)$ tal que $A[k] = k$. El algoritmo devuelve $f(A, 0, \text{len}(A))$. Si logramos probar que f es correcta, como todos los tales índices k están en $[0, |A|)$, el algoritmo va a ser correcto.

Definimos el tamaño de un argumento (A, i, j) como $t(A, i, j) = j - i$, y definimos $P(n)$: Nuestro programa es correcto para todas las entradas de tamaño menor o igual a n . Vamos a probar P por inducción.

1. Caso base, $P(0)$. Si $t(A, i, j) = 0$, entonces $i = j$, y el algoritmo devuelve `False`. La semántica que queríamos que f cumpliera es que devuelve `True` si y sólo si existe un k en $[i, j)$ tal que $A[k] = k$, pero claramente no puede haber ningún tal k si $i = j$, pues $[i, j) = []$. Luego nuestra función es correcta para entradas con tamaño $T(a, i, j) = 0$.
2. Paso inductivo. Asumo que vale $P(n)$, pruebo $P(n + 1)$. Si $t(A, i, j) = n + 1 > 0$, entonces $j > i$. Definimos k como el promedio entre i y j , $k = i + \frac{j-i}{2} = 2\frac{i}{2} + \frac{j-i}{2} = \frac{j+i}{2}$. Como vimos arriba, B es creciente, estricta. Luego:
 1. Si $B[k] > 0$, si existe un k' tal que $B[k'] = 0$, tenemos que tener $k' < k$. Como tenemos que devolver `True` exactamente si existe un tal k' en $[i, j)$, y sabemos que si tal k' existe está antes que k , entonces el k' no va a estar en $[k, j)$, y tiene que estar, si existe, en $[i, k)$. Por ende, nuestro algoritmo es correcto al devolver $f(A, i, k)$, que por hipótesis inductiva es `True` exactamente cuando hay un k' en $[i, k)$ tal que $B[k'] = 0$.
 2. Si $B[k] < 0$, si existe un k' tal que $B[k'] = 0$, tiene que estar después de k . Como tenemos que devolver `True` exactamente si existe un k' en $[i, j)$ tal que $B[k'] = 0$, y de existir tal k' , tiene que ser mayor a k , sabemos que debe estar en $(k, j) = [k + 1, j)$. Luego, nuestro algoritmo es correcto al devolver $f(A, k + 1, j)$, que por hipótesis inductiva es `True` exactamente cuando hay un k' en $[k + 1, j)$ tal que $B[k'] = 0$.

Luego, nuestro algoritmo es correcto para todas las entradas.

Si denotamos por $T(n)$ al número de operaciones que hace nuestro algoritmo al recibir una entrada de tamaño $t(A, i, j) = n$, vemos que $T(0) = c$ para alguna constante c (no podría ser de otra forma, $T(0)$ no puede depender de nada). Mientras tanto, que si $n > 0$, $T(n)$ llama a uno de dos problemas cuyo tamaño es $k - i$, y $j - (k + 1)$ respectivamente, con $k = \lfloor i + \frac{j-i}{2} \rfloor$. Por definición de la función $\lfloor \cdot \rfloor$, sabemos que $i + \frac{j-i}{2} - 1 < k \leq i + \frac{j-i}{2}$.

El primer problema, entonces, tiene tamaño $k - i \leq i + \frac{j-i}{2} - i = \frac{j-i}{2} = \frac{n}{2}$. El segundo problema tiene tamaño $j - (k + 1) \leq j - (i + \frac{j-i}{2} - 1 + 1) = 2\frac{j-i}{2} - \frac{j-i}{2} = \frac{j-i}{2} = \frac{n}{2}$. Además de las llamadas recursivas, hacemos algún número constante de operaciones.

Lo que suelen hacer en la materia es decir que esto es $T(n) = T(\frac{n}{2}) + O(1)$, pero como vemos acá esto no es obviamente correcto (*¿qué es $T(\frac{5}{2})$, si dijimos que $T : \mathbb{N} \rightarrow \mathbb{N}$?*). Nuestro algoritmo a veces va a llamar a un problema de tamaño $\lfloor \frac{n}{2} \rfloor$, y otras veces $n - \lfloor \frac{n}{2} \rfloor - 1$. Como nos piden hacer un análisis de peor caso, esto es $\lfloor \frac{n}{2} \rfloor \geq n - \lfloor \frac{n}{2} \rfloor - 1$, luego en el peor caso siempre caemos en la rama $\lfloor \frac{n}{2} \rfloor$, y para hacer esto lo más grande posible¹⁵, queremos que $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$, con lo cual para caer siempre en esta rama, n tiene que ser una potencia de 2. Esto nos dice que nuestro peor caso son entradas de tamaño

¹⁵Esto asume que T es creciente, lo cual es cierto en este algoritmo, pero no es cierto para todas las funciones.

potencia de 2. Usando esto, podemos concluir que la recurrencia que determina el número de operaciones necesarias *en el peor caso* es $T(n) = T(\frac{n}{2}) + O(1)$, pues en el peor caso, n es una potencia de 2.

Para un tratamiento formal sobre cómo resolver recurrencias que tengan $\lfloor \cdot \rfloor$ y $\lceil \cdot \rceil$, pueden ver [22]. Noten el cuidado que hay que tener al argumentar, y cómo tuvimos que valernos de que estamos buscando el comportamiento asintótico *en el peor caso* para definir T . No existe una función T que nos da el número de pasos necesarios para *toda* entrada de un dado tamaño, porque el número de tales pasos va a variar. Si queremos definir el comportamiento asintótico en el peor caso, tenemos que primero encontrar una tal familia de casos, argumentar por qué son «el peor caso», y recién ahí podemos valernos de que nuestra entrada tiene alguna forma particular (en este caso, tener longitud potencia de 2).

Luego, usando el teorema maestro vemos que $T \in O(\log n)$. La complejidad asintótica espacial es también $O(\log n)$, porque usamos memoria para cada llamada recursiva, en particular para guardar k . Como hay a lo sumo $\lceil \log_2 n \rceil$ llamadas recursivas, y usamos $O(1)$ espacio en cada una, usamos $O(\log n)$ espacio adicional a la entrada en total. \square

Ejercicio 3.3.2

Demostrar que el algoritmo «Mergesort» es correcto.

```
def merge(xs: [int], ys: [int]) -> [int]:
    i, j, n, m = 0, 0, len(xs), len(ys)
    ans = []
    while i < n or j < m:
        if i < n and (j >= m or xs[i] <= ys[j]):
            ans.append(xs[i])
            i += 1
        else:
            ans.append(ys[j])
            j += 1
    return ans

def mergesort(xs: [int]) -> [int]:
    n = len(xs)
    if n <= 1: return xs
    lefts = mergesort(xs[:n//2])
    rights = mergesort(xs[n//2:])
    return merge(lefts, rights)
```

Solución. Este algoritmo consiste de dos sub-algoritmos, `merge` y `mergesort`. Vamos a especificar cada uno, y demostrar que cumplen la especificación.

1. `merge` recibe dos listas de enteros, `xs` e `ys`, ambas ordenadas de forma no-decreciente. Devuelve una lista de enteros `ans`, donde `ans` contiene los mismos elementos que `xs + ys`, y `ans` está ordenada de forma no-decreciente.

2. mergesort recibe una lista de enteros xs , y devuelve una lista de enteros zs , donde zs contiene los mismos elementos que xs , y zs está ordenada de forma no-decreciente.

Lema 3.3.3

`merge` es correcto.



Demostración. Ya vieron cómo demostrar usando el teorema del invariante. También vieron que los algoritmos con estados son, en general, más difíciles de analizar que los que no mutan estado. En esta demostración quiero mostrarles cómo usar las herramientas que tienen para algoritmos recursivos, para algoritmos con estado.

Todo ciclo se puede transformar a un algoritmo con el mismo comportamiento, pero que usa recursión.

```

1: Estado ← EstadoInicial
2: while CONDICIÓN(Estado) do
3:   Estado ← Modificar(Estado)
4: end
5: return POSTPROCESAMIENTO(Estado)
```

Esto es equivalente al siguiente algoritmo recursivo:

```

1: procedure REC(Estado)
2:   if ¬CONDICIÓN(Estado) then
3:     return POSTPROCESAMIENTO(Estado)
4:   end
5:   return REC(MODIFICAR(Estado))
6: end
7: REC(EstadoInicial)
```

Estos algoritmos tienen la misma semántica, y costo computacional¹⁶. Para la función `merge`, la versión iterativa queda así:

```

def merge(xs: [int], ys: [int]) -> [int]:
    n, m = len(xs), len(ys)
    def g(i, j, ans):
        if not (i < n or j < m): return ans
        if i < n and (j >= m or xs[i] <= ys[j]):
            return g(i + 1, j, ans + [xs[i]])
        return g(i, j + 1, ans + [ys[j]])
    return g(0, 0, [])
```

La semántica que le vamos a dar a g es:

¹⁶Para justificar que tienen el mismo costo computacional, hay que definir un modelo computacional. Por ejemplo, hay que definir si una llamada recursiva de este estilo usa espacio para el «stack». Es razonable asumir que las funciones de esta forma no requieren más espacio en memoria que el código iterativo. Alumnos interesados pueden leer sobre el concepto de «tail-call recursion» para entender por qué es razonable asumir esto. También los invito a leer cómo su lenguaje favorito implementa esta optimización.

$$g : [0, \dots, n] \times [0, \dots, m] \times \text{List[int]} \rightarrow \text{List[int]}$$

$g(i, j, a) = a + t$, donde t está ordenada de forma no-decreciente,

y tiene exactamente los elementos de $x[i, \dots, n - 1] + y[j, \dots, m - 1]$

Queremos ver en qué vamos a hacer inducción para probar que g es correcta. Vemos que en cada llamada recursiva, aumenta i o aumenta j , y el otro queda igual. Entonces, lo que decrece es $n - i$, o $m - j$. Si decrece al menos uno, y el otro queda igual, dos opciones para algo en lo que hacer inducción son la suma, $n - i + m - j$, y el producto, $(n - i)(m - j)$. Si elegimos la suma, entonces vamos a tener problemas para probar el caso base, porque $n - i + m - j$ no nos dice que $i = n, m = j$, que es lo que usa g para no-hacer recursión. Luego miremos el producto, $(n - i)(m - j)$, que sí nos deja concluir eso. El problema acá va a ser que cuando $i = n$ o $j = m$, las llamadas recursivas no bajan el valor de este producto (sigue siendo cero). Luego queremos modificar esto a $(n - i + 1)(m - j + 1)$, que no tiene ese problema. Entonces definimos la proposición

Definición 3.3.4

$P(k) : g(i, j, a)$ es correcta para todo $i, j \in \mathbb{N}, 0 \leq i \leq n, 0 \leq j \leq m$ tal que $(n - i + 1)(m - j + 1) = k$.



Vamos a definir por comodidad la notación $a \lesssim b$ como que $a \leq t \forall t \in b$. Por ejemplo, $5 \lesssim [6, 5, 9]$, pero $5 \not\lesssim [8, 3, 10]$.

Para ver que `merge` es correcta, tenemos que probar que vale $g(0, 0, []])$ es correcta. Si probamos que $P((n + 1)(m + 1))$, entonces como $(n - 0 + 1)(m - 0 + 1) = (n + 1)(m + 1)$, vemos que $g(0, 0, [])$ es correcta, y luego que `merge` devuelve una lista que contiene los mismos elementos que $\text{xs}[0 : n] + \text{ys}[0 : m] = \text{xs} + \text{ys}$, pero ordenada de forma no-decreciente, que es precisamente la semántica que queríamos darle a `merge`.

1. Caso base, $P(1)$. Tenemos que probar que para todo $i, j \in \mathbb{N}, 0 \leq i \leq n, 0 \leq j \leq m$ tal que $(n - i + 1)(m - j + 1) = 1$, g es correcta. Como $i \leq n$ y $j \leq m$, entonces ambos factores de este producto son números naturales. Si tenemos un producto de naturales que es 1, entonces ambos naturales son 1. Luego, $n - i + 1 = 1$, y $m - j + 1 = 1$. Esto nos dice que $i = n$, y $j = m$. En este caso, tenemos que `not (i < n or j < m)`, y entonces $g(n, m, a)$ devuelve a . Ahora bien, a es lo mismo que $a + []$ y $[]$ es precisamente la unión de todos los elementos de $x[i, \dots, n - 1] + y[j, \dots, m - 1] = x[n, \dots, n - 1] + y[m, \dots, m - 1] = [] + [] = []$. Luego, g es correcta para este caso.
2. Paso inductivo. Sabemos que vale $P(r)$ para todo $r < k$, queremos ver que vale $P(k)$. Sean entonces $i, j \in \mathbb{N}, 0 \leq i \leq n, 0 \leq j \leq m$, tal que $(n - i + 1)(m - j + 1) = k$. Partimos en casos, si $i = n$, si $j = m$, o si ninguna es cierta.
 - a. Si $i = n$ y $j \neq m$, entonces $k = m - j + 1$. Como $j \leq m$, entonces $j < m$, no salimos en la primer condición (`return ans`). Como $i < n$ es falso, $g(i, j, a)$ evalúa a $g(i, j + 1, a + [y[j]])$. Como $m - (j + 1) + 1 < m - j + 1 = k$, podemos usar la hipótesis inductiva $P(m - (j + 1) + 1)$, para concluir que si llamamos $X = g(i, j + 1, a + [y[j]]) = a + [y[j]] + b$, entonces b es una lista que contiene los elementos de $x[i, \dots, n - 1] + y[j + 1, \dots, m - 1]$, ordenados de forma no-decreciente. Entonces $X \lesssim x[i, \dots, n - 1] + y[j + 1, \dots, m - 1]$, y $X \lesssim y[j + 1, \dots, m - 1] + x[i, \dots, n - 1]$, que es lo que queríamos.

- decreciente. Como $i = n$, entonces $x[i, \dots, n-1] + y[j+1, \dots, m-1] = y[j+1, \dots, m-1]$. Como y está ordenada de forma no-decreciente, entonces $y[j] \lesssim b$. Luego $t = [y[j]] + b$ está ordenada de forma no-decreciente, y tiene los mismos elementos que $y[j, \dots, m-1] = x[i, \dots, n-1] + y[j, \dots, m-1]$. Luego, $X = a + t$, con t teniendo los mismos elementos que $x[i, \dots, n-1] + y[j, \dots, m-1]$, ordenados de forma no-decreciente, que es lo que queríamos demostrar para $P(k)$.
- Pasa algo análogo si $j = m$ y $i < n$.
 - Si $i < n$ y $j < m$, entonces partimos en dos casos, dependiendo de si $x[i] \leq y[j]$ o no.
 - Si $x[i] \leq y[j]$, g devuelve $g(i+1, j, a + [x[i]])$. Como $n - (i+1) + 1 < n - i + 1$, entonces $(n - (i+1) + 1)(m - j + 1) < (n - i + 1)(m - j + 1) = k$, y podemos usar la hipótesis inductiva $P((n - (i+1) + 1)(m - j + 1))$ para ver que $g(i+1, j, a + [x[i]]) = a + [x[i]] + b$, con b una permutación no-decreciente de $x[i+1, \dots, n-1] + y[j, \dots, m-1]$. Como x e y son no-decrecientes, $x[i] \lesssim x[i+1, \dots, n-1]$, y $x[i] \leq y[j] \lesssim y[j, \dots, m-1]$. Luego, $x[i] \lesssim b$, y luego llamando $t = [x[i]] + b$, vemos que $g(i, j, a)$ está devolviendo $a + t$, con t una lista no-decreciente, que contiene los mismos elementos que $x[i, \dots, n-1] + y[j, \dots, m-1]$. Esto es precisamente lo que hay que probar para $P(k)$.
 - Si $x[i] > y[j]$, pasa algo análogo con $g(i, j+1, a + [y[j]])$.

Luego, demostramos $P(k)$ para todo $k \geq 1$. □

Habiendo probado que `merge` es correcta para toda entrada, probamos ahora fácilmente que `mergesort` es correcta.

Lema 3.3.5

`mergesort` es correcta. ♡

Demostración. Al ser `mergesort` una función recursiva, la primer herramienta que vamos a intentar es usar inducción.

Veamos primero, ¿qué es lo que decrece en cada llamada recursiva? Nos dan una lista, x , y la dividimos en dos partes, aproximadamente de la mitad del tamaño cada vez (lo de aproximado es porque no todas las entradas tienen un número par de elementos). Luego, lo que está decreciendo cada vez es el tamaño de la lista que nos pasan.

Definición 3.3.6

$P(k)$: $\text{merge}(x)$ tiene los mismos elementos que x , pero ordenados de forma no-decreciente, para toda lista x con a lo sumo k elementos.

1. Caso base, $P(0)$. Si x tiene 0 elementos, entonces $x = []$, y $\text{merge}([]) = []$ por su primer `if`, que es la respuesta correcta. Luego vale $P(0)$.
2. Caso base, $P(1)$. Si x tiene 1 elemento, entonces $x = [\alpha]$ para algún α , y $\text{merge}([\alpha]) = [\alpha]$ por su primer `if`, que es la respuesta correcta. Luego vale $P(1)$.
3. Paso inductivo. Sea $k \in \mathbb{N}$, $k > 1$. Asumo que vale $P(r)$ para todo $r < k$, quiero ver que vale $P(k)$. Sea $a = \lfloor \frac{k}{2} \rfloor$. Como $k > 1$, entonces $a > 0$. Sea $b = k - a$. Luego $a < k$, y $b < k$. Luego podemos usar las hipótesis inductivas $P(a)$ y $P(b)$, para ver que `lefts` tiene los mismos elementos que $x[0, \dots, a - 1]$, y `rights` tiene los mismos elementos que $x[a, \dots, n - 1]$. Luego, su concatenación `lefts + rights` tiene los mismos elementos que x . Vemos entonces que llamando a `merge(lefts, rights)`, tendremos una lista ordenada de forma no-decreciente, que tiene los mismos elementos que `lefts + rights`, que a su vez son los mismos elementos que x . Esto es precisamente la semántica que queríamos para `mergesort`, y luego vale $P(k)$.

□

Ejercicio 3.3.7

Se tienen n objetos de pesos p_1, \dots, p_n no-negativos, y valores v_1, \dots, v_n no-negativos, y una mochila en la que caben varios objetos, pero aguanta como máximo un peso P .

1. Diseñar un algoritmo basado en programación dinámica que encuentre el máximo valor alcanzable poniendo objetos en la mochila.
2. Demostrar que el algoritmo es correcto.
3. Demostrar su complejidad temporal y espacial, en el peor caso. El mejor algoritmo que conocemos tiene complejidad espacial $O(P)$ y complejidad temporal $O(np)$.

◆

Primero una explicación de cómo podemos pensar esto, y luego una solución como la que se espera que escriban en un parcial.

En principio, el espacio de búsqueda que tenemos es el conjunto de subconjuntos de los n objetos. Por ejemplo, si $n = 3$, y los objetos son $X = \{a, b, c\}$, el espacio de búsqueda que tenemos es $\mathcal{P}(X) = \{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Una estrategia simple que usa esta estructura sería *backtracking*.

Si queremos ver si un problema puede ser resuelto con programación dinámica, tenemos que ver si podemos parametrizar al conjunto de subproblemas de tal manera que tengan un orden, y se cumplan las dos propiedades clásicas de subestructura óptima y subproblemas compartidos.

En este caso, podemos definir los subproblemas como pares $G = \{(i, x) \mid 0 \leq i \leq n, 0 \leq x \leq P\}$, donde un subproblema (i, x) significa «El mayor valor que podemos obtener usando peso a lo sumo x , y usando sólo los primeros i objetos.» Llamemos $v^* : G \rightarrow \mathbb{R}$ la función que nos da tal valor. Vemos entonces que la respuesta al problema entero es la respuesta al subproblema (n, P) . Más aún, vemos que se cumplen las dos condiciones:

1. $v^*(i, x)$ es o bien $v^*(i - 1, x)$, si una solución al subproblema (i, x) no usa el i -ésimo objeto, o $v^*(i - 1, x - p_i) + v_i$, si una solución al subproblema (i, x) usa el i -ésimo objeto. En el primer

caso, tenemos el mismo peso disponible (x) para usarlo de la mejor forma posible usando sólo los primeros $i - 1$ objetos, y nuestro valor es el valor que nos den los objetos que elegimos de esos primeros $i - 1$. En el segundo caso, si usamos el i -ésimo objeto, tenemos $x - p_i$ peso restante para los otros objetos que vamos a elegir dentro de los primeros $i - 1$, y el valor de esta solución es el valor de la solución a $(i - 1, x - p_i)$, más el valor que obtenemos por haber tomado el i -ésimo objeto, v_i .

2. La función v^* tiene dominio $[0...n] \times [0...P]$, luego hay sólo nP valores posibles. Sin embargo, en una implementación recursiva tradicional, en el peor caso vamos a tener un número exponencial de llamadas. Esto es fácil de ver si tomamos como familia de casos donde $p_i = 0 \forall i$, vemos que $v^*(i, x) = \max(v^*(i - 1, x), v^*(i - 1, x) + v_i)$, y luego una implementación recursiva tradicional tendría $T(i) = 2T(i - 1)$ llamadas, lo cual termina siendo 2^n llamadas.

Como tenemos ambos subestructura óptima, y subproblemas compartidos, podemos usar programación dinámica y esperar mejoras en el tiempo de cómputo.

Más aún, vemos que $v^*(i, x)$ sólo depende de cosas en $v^*(i - 1, \dots)$, luego si usamos programación dinámica bottom-up, sólo necesitamos quedarnos con dos «filas» de la matriz de programación dinámica, porque al llenar una fila, sólo necesitamos ver la fila inmediatamente anterior, para responder el problema sólo necesitamos una entrada en la última fila.

Solución. Primero planteamos la función recursiva, junto con su semántica.

$$f : [0...n] \times [0..P]$$

$f(i, x)$: El máximo valor que puedo obtener usando los primeros i objetos y un peso de a lo sumo x .

La respuesta al enunciado es $f(n, P)$. Las ecuaciones recursivas para f son:

$$f(i, x) = \begin{cases} 0 & , \text{ si } i = 0 \\ f(i - 1, x) & , \text{ si } p_i > x \\ \max(f(i - 1, x), f(i - 1, x - p_i) + v_i) & , \text{ si no} \end{cases}$$

Demostración. Vamos a probar que f es correcta usando inducción. Para poder razonar formalmente, vamos a definir algunos conceptos.

- Definimos $v(S) = \sum_{j \in S} v_j$, la suma de los valores de los elementos de un subconjunto $S \subseteq \{1, \dots, n\}$. Asimismo $p(S) = \sum_{j \in S} p_j$, la suma de los pesos de los elementos de S .
- Definimos $F(i, x) = \{S \subseteq \{1, \dots, i\} \mid p(S) \leq x\}$. Estos son los conjuntos de objetos, de entre los primeros i , que podemos meter en la mochila, con suma de peso a lo sumo x . Algunos elementos de $F(i, x)$ van a tener valor más alto que otros. Vemos que $F(i, x) \subseteq F(i + 1, x)$ para todo $1 \leq i < n$.
- Definimos $v^*(i, x) = \max_{S \in F(i, x)} \{v(S)\}$, el máximo valor que podemos obtener, usando los primeros i objetos, con peso total a lo sumo x .

Sea $P(i) : i \leq n \Rightarrow f(i, x) = v^*(i, x)$. Vamos a probar $P(i) \forall i \in \mathbb{N}$ por inducción.

1. Caso base, $P(0)$. Tenemos que probar que $f(0, x) = v^*(0, x)$. Por definición, $v^*(0, x) = \max_{S \in F(0, x)} \{v(S)\}$, pero $F(0, x) = \{S \subseteq \{1, \dots, 0\} \mid p(S) \leq x\} = \{\emptyset\}$, pues el único subconjunto de los primeros 0 objetos es \emptyset . Luego, $v^*(0, x) = v(\emptyset) = \sum_{j \in \emptyset} v_j = 0$. Nuestra función f efectivamente devuelve 0, en su primer rama, donde $i = 0$. Luego $f(0, x) = 0 = v^*(0, x)$, lo cual demuestra $P(0)$.
2. Paso inductivo. Sabemos que vale $P(t)$, queremos ver que vale $P(t + 1)$. Es decir, queremos probar que $v^*(t + 1, x) = f(t + 1, x)$. Llamemos $i = t + 1$. Si $i > n$, no hay nada que probar, pues «falso implica todo», y estamos probando una implicación ($P(i)$) con antecedente falso. Luego, asumimos que $i \leq n$.

Por definición, $v^*(i, x) = \max_{S \in F(i, x)} \{v(S)\}$.

1. Si $p_i > x$, sea $S \in F(i, x)$. Como $S \in F(i, x)$, sabemos que $p(S) \leq x$. Si i estuviera en S , tendríamos que $p(S) = \sum_{j \in S} p_j \geq p_i > x$, lo cual no sucede. Luego $i \notin S$. Como esto sucede para cualquier $S \in F(i, x)$, ningún $S \in F(i, x)$ contiene a i , y entonces $S \subseteq \{1, \dots, i\} \setminus \{i\} = \{1, \dots, i - 1\}$. Luego, $F(i, x) \subseteq F(i - 1, x)$, y como sabíamos que $F(i - 1, x) \subseteq F(i, x)$, tenemos que $F(i, x) = F(i - 1, x)$. Luego, $v^*(i, x) = \max_{S \in F(i, x)} \{v(S)\} = \max_{S \in F(i - 1, x)} \{v(S)\} = v^*(i - 1, x)$. Como sabemos $P(t)$, es decir $P(i - 1)$, sabemos que $f(i - 1, x) = v^*(i - 1, x)$. Como $f(i, x)$ devuelve $f(i - 1, x) = v^*(i - 1, x)$ cuando $p_i > x$, y en ese caso $v^*(i - 1, x) = v^*(i, x)$, tenemos $f(i, x) = v^*(i, x)$, es decir $P(i)$.
2. Si $p_i \leq w$, podemos particionar $F(i, x)$ en dos subconjuntos, A y B , con $A = \{S \in F(i, x) \mid i \notin S\}$, y $B = \{S \in F(i, x) \mid i \in S\}$. Como A y B partitionan $F(i, x)$, entonces $\max_{F(i, x)} \{v(S)\} = \max(\max_{S \in A} \{v(S)\}, \max_{S \in B} \{v(S)\})$.
 - Tomemos un $S \in A$. Vemos que $A = \{S \in F(i, x) \mid i \notin S\} = \{S \subseteq \{1, \dots, i\} \mid p(S) \leq x \wedge i \notin S\} = \{S \subseteq \{1, \dots, i - 1\} \mid p(S) \leq w\} = F(i - 1, x)$. Luego, $\max_{S \in A} \{v(S)\} = \max_{S \in F(i - 1, x)} \{v(S)\} = v^*(i - 1, x)$.
 - Tomemos ahora un $S \in B$. Como $S \in B$, entonces $i \in S$. Llamemos $S = S' \cup \{i\}$, con $S' \subseteq \{1, \dots, i - 1\}$. Luego, $p(S) = p_i + p(S')$. Como $S \in B \subseteq F(i, x)$, $p(S) \leq x$, y luego $p_i + p(S') \leq x$. Por ende, $p(S') \leq x - p_i$. Luego, $S' \in F(i - 1, x - p_i)$. Luego, cada $S \in B$ se corresponde con un único $S' \in F(i - 1, x - p_i)$, y la biyección es simplemente $\varphi(X) = X \cup \{i\}$. Para transformar los valores luego de esta biyección, vemos que $v(S) = v_i + v(S')$. Luego,

$$\max_{S \in B} \{v(S)\} = \max_{S' \in F(i - 1, x - p_i)} \{v(S') + v_i\} =$$

$$\max_{S' \in F(i - 1, x - p_i)} \{v(S')\} + v_i = v^*(i - 1, x - p_i) + v_i.$$

Juntando ambas ramas, vemos que $v^*(i, x) = \max_{F(i, x)} \{v(S)\} = \max(v^*(i - 1, x), v^*(i - 1, x - p_i) + v_i)$. Por $P(t)$, que es $P(i - 1)$, sabemos que $f(i - 1, x) = v^*(i - 1, x)$, y $f(i - 1, x - p_i) = v^*(i - 1, x - p_i)$. Por lo tanto, como nuestra función devuelve $\max(f(i - 1, x), f(i - 1, x - p_i))$, está devolviendo $v^*(i, x)$, que prueba $P(i)$.

Esto prueba $P(i)$ para todo $i \in \mathbb{N}$. En particular, para todo $i \in \mathbb{N}$, $0 \leq i \leq n$, y para todo $x \in \mathbb{N}$, $0 \leq x \leq P$, tenemos que $f(i, x) = v^*(i, x)$, y luego $f(n, P) = v^*(n, P)$, que muestra que nuestro algoritmo es correcto. \square

Veamos el código ahora en Python:

```

def f(i, x):
    if i == 0: return 0
    if p[i] > x: return f(i - 1, x)
    return max(f(i - 1, x), f(i - 1, x - p[i]) + v[i])

```

Si queremos hacer esto un algoritmo de programación dinámica top-down, lo único que hay que hacer es mecánicamente agregar un cache.

```

def f(i, x, cache = {}):
    if (i, x) not in cache:
        if i == 0: r = 0
        elif p[i] > x: r = f(i - 1, x, cache)
        else: r = max(f(i - 1, x, cache), f(i - 1, x - p[i], cache) + v[i])
        cache[(i, x)] = r
    return cache[(i, x)]

```

El número asintótico de operaciones en este algoritmo es más difícil de analizar, porque está mutando el estado (cache) a medida que hacemos llamadas recursivas. También va a depender del costo que tenga insertar y buscar en la estructura que usamos para el cache.

Para hacer este algoritmo bottom-up, tenemos que pensar en qué orden se llena la tabla, y llenarla nosotros mismos. Vemos que necesitamos leer un valor de $f(i, x)$ sólo cuando estamos escribiendo el valor de $f(i + 1, x')$ para algún x' . Luego, si llenamos la tabla en orden creciente de i , cada vez que querremos leer un valor, ya lo vamos a tener en la tabla.

```

def f(n, P):
    dp = [[0 for _ in range(P + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for x in range(P + 1):
            if x < p[i]:
                dp[i][x] = dp[i - 1][x]
            else:
                dp[i][x] = max(dp[i - 1][x], dp[i - 1][x - p[i]] + v[i])
    return dp[n][P]

```

El comportamiento asintótico de este algoritmo es mucho más fácil de analizar, el número de operaciones está en $\Theta(nP)$ en todos los casos, y el costo espacial es también $\Theta(nP)$, pues la tabla dp tiene $n \times P$ entradas.

Por último, vemos que no hace falta mantener en memoria toda la tabla, si sólo queremos devolver $dp[n][P]$. Para llenar una fila, $dp[i]$, sólo hace falta la fila anterior, $dp[i - 1]$. Luego podemos mantener sólo dos filas a la vez, $dp1$ y $dp2$, donde $dp2 = dp[i]$, y $dp1 = dp[i - 1]$:

```

def f(n, P):
    dp1 = [0 for _ in range(P + 1)]
    dp2 = [0 for _ in range(P + 1)]
    for i in range(1, n + 1):
        for x in range(P + 1):
            if x < p[i]:
                dp2[x] = dp1[x]
            else:
                dp2[x] = max(dp1[x], dp1[x - p[i]] + v[i])
    dp1, dp2 = dp2, dp1
    return dp1[P]

```

El costo temporal es idéntico, pero bajamos el costo espacial a sólo $\Theta(P)$ en todos los casos.

Ejercicio 3.3.8

Sea $X = [x_1, x_2, \dots, x_n]$ una secuencia de n booleanos (1 o 0) y sea $k \in \mathbb{N}$ un número entre 1 y n . Supongamos que se pueden eliminar k ceros, queremos saber la longitud máxima que puede tener una cadena de 1s. Por ejemplo si $k = 2$ y $X = 11001010001$ la respuesta es 3, mientras que si $k = 3$ la respuesta es 4.

1. Diseñar un algoritmo basado en programación dinámica que indique la longitud más larga de una subsecuencia de unos sacando a lo sumo k ceros de S . Debe tener complejidad a lo sumo $O(nk)$.
2. Demostrar que el algoritmo es correcto.
3. Demostrar su complejidad temporal y espacial en el peor caso.

Primero les voy a mostrar en qué pienso al resolver el ejercicio, y luego una resolución.

Hay una estructura de orden ahí, porque si hablo de «puedo eliminar k ceros», al eliminar un cero, caigo a un estado en el que «puedo eliminar $k - 1$ ceros». Ese probablemente va a ser uno de los parámetros de mi función.

Si veo un 1, puedo armar una cadena de unos que termina en ese 1. Pero al hacer una llamada recursiva, no puedo preguntar algo como «la secuencia de 1s más larga borrando k ceros que termina antes de esta posición», porque esa cadena puede terminar mucho más atrás que la posición que estoy viendo, y entonces no podría tomar ese número y sumarle uno, extendiendo esa solución con el 1 que estoy mirando. Luego, tengo que restringir que la cadena de 1s termine exactamente donde estoy parado. Luego haré un pasada por la lista, diciendo que la cadena más larga de 1s borrando k ceros, es la cadena más larga de 1s borrando k ceros que termina exactamente en $i = 0$, o la que termina exactamente en $i = 1$, etc.

Veamos si puedo plantear esto.

- Si veo un 1 en x_i , entonces puedo tomar $f(i - 1, k) + 1$ como la respuesta para esta posición.
- Si veo un 0 en x_i , entonces puedo tomar $f(i - 1, k - 1) + 1$ si $k > 0$, o 0 si no. También puedo tomar una cadena de longitud 0 que terminar en este 0, si no quiero tomar la solución recursiva (que puede no existir, si por ejemplo estoy en un prefijo de más de k ceros).

$$f(i, k) = \begin{cases} -\infty & \text{si } k < 0 \\ 0 & \text{si } i < 0 \\ f(i - 1, k) & \text{si } x_i = 1 \\ \max(0, f(i - 1, k - 1)) & \text{si } x_i = 0 \wedge k > 0 \\ 0 & \text{si no} \end{cases}$$

algo como:

```
def F(x, k):
    def f(i, kk):
        if k < 0: return -99999
        if i < 0: return 0
        if x[i] == 1: return f(i - 1, kk) + 1
        if x[i] == 0 and kk > 0: return f(i - 1, kk - 1)
```

```

    return 0
return max(f(i, k) for i in range(len(x)))

```

Probablemente ni necesito lo de $-\infty$. OK, a escribir.

Solución. Definimos primero una función y su semántica.

$$f : [-1, \dots, n] \times [0, \dots, k]$$

$$f(i, r) = \begin{cases} f(i - 1, r) + 1 & \text{si } i \geq 0 \wedge x_i = 1 \\ f(i - 1, r - 1) & \text{si } i \geq 0 \wedge x_i = 0 \wedge r > 0 \\ 0 & \text{si no} \end{cases}$$

La semántica de $f(i, r)$ es «La longitud de la cadena de unos más larga que termina en la posición i , borrando a lo sumo r ceros.»

El algoritmo devuelve

$$\max_{0 \leq i \leq n} f(i, k)$$

Demostración. Está claro que toda cadena de unos borrando a lo sumo k ceros termina en alguna posición. Luego, si probamos que f es correcta (es decir, que cumple su semántica), estamos probando que nuestro algoritmo es correcto, pues estamos tomando el máximo sobre todas las posibles posiciones donde terminaría tal secuencia.

Vamos a probar que f es correcta usando inducción. Definimos $v^*(i, t)$ como la longitud de cadena de unos más larga, borrando a lo sumo t ceros, que termina exactamente en i , 0 cero si no existen tales cadenas. Definimos $P(i) : i \leq n \Rightarrow (f(i, t) = v^*(i, t) \forall t \in \mathbb{N})$. Por comodidad, vamos a definir $\theta(i, t)$ como el conjunto de cadenas de unos que termina en la posición i , y borra a lo sumo t ceros, y $\theta^*(i, t)$ como el subconjunto de $\theta(i, t)$ que tiene número máximo de unos, para cada i, t .

- 1) Caso base, $P(0)$. La longitud de una cadena de unos más larga que termina en la ($i = 0$)-ésima posición es o bien 1 o 0, dependiendo de si $x_0 = 1$ o $x_0 = 0$. Luego, si $v^*(0, t) = x_0$. Si $x_0 = 1$, nuestra función $f(0, t)$ devuelve $f(-1, t) + 1$, que evalúa a 1 inmediatamente. Si $x_0 = 0 \wedge t > 0$, $f(0, t)$ devuelve $f(-1, t - 1)$, que evalúa a 0 inmediatamente. Finalmente, si $x_0 \wedge t = 0$, entonces $f(0, 0) = 0$. Luego en todos los casos tenemos $f(0, t) = v^{0,t}$, lo que prueba $P(0)$.
- 2) Paso inductivo. Sabemos $P(i)$, queremos probar $P(i + 1)$. Sea $t \in \mathbb{N}$. Sea $T \in \theta^*(i + 1, t)$, y por ende $|T| = v^*(i + 1, t)$. Partimos en casos, dependiendo de quién es x_i :
 - a) Si $x_{i+1} = 1$. Como T termina en $i + 1$, $i + 1 \in T$, puesto que sólo podemos borrar ceros. Sea $T' = T \setminus \{i + 1\}$. Como T' borra a lo sumo t ceros, y termina en i , está en $\theta(i, t)$. Si no estuviera en $\theta^*(i, t)$, podríamos tomar cualquier $S \in \theta^*(i, t)$, y por lo tanto $|S| > |T'|$, con lo cual $|S + \{i + 1\}| > |T|$, pero esto no puede pasar, porque T está en $\theta^*(i + 1, t)$, luego tiene el número máximo de unos. Luego, $T' \in \theta^*(i, t)$, y luego $|T'| = v^*(i, t)$. Como sabemos $P(i)$, esto es $|T'| = f(i, t)$, y por ende, $|T| = f(i, t) + 1$, que es precisamente lo que devuelve $f(i + 1, t)$, y por ende vale $P(i + 1)$.

- b) Si $x_{i+1} = 0 \wedge t > 0$. Entonces $i + 1 \notin T$. Entonces, como $t > 0$, y T termina en $i + 1$, vemos que $T \in \theta(i, t - 1)$, y luego $v^*(i + 1, t) = |T| \leq v^*(i, t - 1)$. Tomemos ahora cualquier $S \in \theta^*(i, t - 1)$. Podemos expandir S a terminar en $i + 1$, borrando el elemento $i + 1$, con lo cual S también está en $\theta(i + 1, t)$. Luego $v^*(i, t - 1) = |S| \leq v^*(i + 1, t)$. Luego $v^*(i, t - 1) = v^*(i + 1, t)$. Por $P(i)$, $f(i, t - 1) = v^*(i, t - 1)$, y entonces como $f(i + 1, t)$ devuelve $f(i, t - 1)$, devuelve $v^*(i, t - 1) = v^*(i + 1, t)$, que muestra $P(i + 1)$.
- c) Si $x_{i+1} = 0 \wedge t = 0$, entonces T es una secuencia de unos que termina en un $i + 1$, pero no puede borrar ningún cero pues $t = 0$. Luego T no existe, y por definición, $v^*(i + 1, 0) = 0$ en este caso, que es precisamente lo que devuelve nuestra función.

Luego tenemos $P(i) \forall i \in \mathbb{N}$. □

El código en Python para la función recursiva es:

```
def F(x, k):
    def f(i, t):
        if i < 0: return 0
        if x[i] == 1: return f(i - 1, t) + 1
        if x[i] == 0 and t > 0: return f(i - 1, t - 1)
        return 0
    return max(f(i, k) for i in range(len(x)))
```

La manera que formulamos el problema como un conjunto de subproblemas y un orden entre ellos cumple dos propiedades:

1. Subestructura óptima. Para resolver un problema (i, t) , necesitamos resolver algún número de subproblemas más pequeños $((i - 1, t)$ o $(i - 1, t - 1)$). Esto es esencialmente lo que nos deja usar recursión.
2. Subproblemas compartidos. Podemos ver que el peor caso sucede cuando k es muy grande o x está lleno de unos (y nunca nos quedamos «sin presupuesto» de ceros para borrar). En esos casos, llamar a $f(i, t)$ resulta en i llamadas recursivas. Como luego tomamos un \max para cada i entre 1 y n , vamos a computar $\frac{n(n-1)}{2}$ problemas en el peor caso, muchos de los cuales son idénticos.

Como siempre, para hacer esto un algoritmo de programación dinámica top-down, mecánicamente agregamos un cache:

```
def F(x, k):
    def f(i, t, cache = {}):
        if (i, t) not in cache:
            if i < 0: r = 0
            elif x[i] == 1: r = f(i - 1, t, cache) + 1
            elif x[i] == 0 and t > 0: r = f(i - 1, t - 1, cache)
            else: r = 0
            cache[(i, t)] = r
        return cache[(i, t)]
    return max(f(i, k) for i in range(len(x)))
```

Esto evita computar subproblemas dos veces, pero hace difícil el análisis de complejidad temporal, dado que estamos mutando estado (cache), y el tiempo que va a tomar una llamada

va a depender del estado cuando es llamada. Asimismo, agregamos ahora el costo adicional de leer y escribir la estructura cache.

Para hacer más claro el análisis algoritmo, y bajar su complejidad en la práctica cuando vamos a llenar cache enteramente de todos modos, podemos usar programación dinámica bottom-up. Esto implica ver en qué orden se llena cache en la versión top-down, y llenarlo nosotros mismos en ese orden. En este caso, vemos que llenamos una entrada $\text{cache}[(i, t)]$ sólo luego de llamar a $f(i - 1, \dots)$, que va a escribir $\text{cache}[(i - 1, \dots)]$. Luego, si llenamos cache en orden creciente de i , vamos a estar llenando la estructura en un orden que garantiza siempre tener escritos los valores que queremos leer, al momento de querer leerlos.

```
def F(x, k):
    n = len(x)
    dp = [[0 for _ in range(k + 1)] for _ in range(n)]
    for i in range(n):
        for t in range(k + 1):
            if x[i] == 1:
                if i == 0: dp[i][t] = 1
                else: dp[i][t] = dp[i-1][t] + 1
            else:
                if t > 0:
                    if i == 0: dp[i][t] = 0
                    else: dp[i][t] = dp[i - 1][t - 1]
                else:
                    dp[i][t] = 0
    return max(dp[i][k] for i in range(n))
```

Esa es una traducción totalmente literal, donde quedaron varios condicionales porque no podemos leer $\text{dp}[-1]$. Podemos limpiar el código un poco:

```
def H(x, k):
    n = len(x)
    dp = [[0 for _ in range(k + 1)] for _ in range(n)]
    for t in range(k + 1): dp[0][t] = x[0]
    for i in range(1, n):
        for t in range(k + 1):
            if x[i] == 1: dp[i][t] = dp[i - 1][t] + 1
            elif x[i] == 0 and t > 0: dp[i][t] = dp[i - 1][t - 1]
            else: dp[i][t] = 0
    return max(dp[i][k] for i in range(n))
```

Vemos que el número de operaciones que hacemos en todos los casos es $\Theta(nk)$, pues eso cuesta construir el array dp . Los dos ciclos anidados hacen $\Theta(nk)$ operaciones, y el ciclo de inicialización de $\text{dp}[0]$ hace $\Theta(k)$ operaciones. El ciclo final, que computa \max , hace $\Theta(n)$ operaciones.

El costo espacial del algoritmo es, en todos los casos, $\Theta(nk)$.

Ejercicio 3.3.9

Sea $v = (v_0, v_1, \dots, v_{n-1})$ un vector de números enteros. Diseñar un algoritmo que indique la mínima cantidad de números que hay que eliminar del vector para que cada número que permanezca sea múltiplo del anterior (excepto el primero). Por ejemplo, para los vectores $(-5, 5, 0)$, $(0, 5, -5)$, y $(0, 5, -5, 2, 15, 15)$, los resultados deberían ser respectivamente 0, 1, y 2. El algoritmo debe tener complejidad temporal $O(n^2)$ y estar basado en programación dinámica.

1. Demostrar que el algoritmo es correcto.
2. Demostrar su complejidad temporal y espacial.

Solución. Definimos una función recursiva y su semántica.

$$f : [0, \dots, n] \rightarrow N$$

$f(i) =$ La longitud de la subsecuencia de v más larga,
donde cada elemento es múltiplo del anterior,
y que termina exactamente en v_i .

La respuesta que devuelve el algoritmo es

$$A = n - \max_{0 \leq i < n} \{f(i)\}$$

Y ahora las ecuaciones que definen f .

$$f(i) = \begin{cases} 1 & \text{si } i = 0 \\ 1 + \max_0 \{f(j) \mid j \in [0, i), v_j \mid v_i\} & \text{si no} \end{cases}$$

donde definimos \max_0 como max, excepto que en el conjunto vacío devuelve 0. Tenemos que probar dos cosas:

1. La función f cumple la semántica que le dimos.
2. Si la función f cumple la semántica que le dimos, entonces A es la respuesta correcta al enunciado.

Probemos ambas, entonces.

Demostración.

1. Vamos a probar que f cumple la semántica que le dimos por (repitan conmigo) inducción. Definimos entonces $S(i)$ como el conjunto de subsucesiones de v que terminan en v_i y cada elemento es múltiplo del anterior, y definimos $g(i) = \max\{|s| \mid s \in S(i)\}$. Queremos probar la proposición $P(i) : i < n \Rightarrow f(i) = g(i)$, para todo $i \in \mathbb{N}$.
 1. Caso base, $P(0)$. Queremos ver que $f(0) = g(0)$. Por definición, $f(0) = 1$. $g(0)$ es la longitud de la subsecuencia de v más larga, donde cada elemento es múltiplo del anterior, y que termina en exactamente en v_0 . Pero v_0 es el primer elemento, luego hay una sola tal subsucesión, y es $[v_0]$, que tiene longitud 1. Por lo tanto, $g(0) = 1$, y tenemos $f(0) = g(0)$, probando $P(0)$.

2. Paso inductivo. Sabemos $P(j)$ para todo $j < i$, queremos ver $P(i)$. Si $i \geq n$, entonces vale $P(i)$ trivialmente, pues $P(i)$ es una implicación con antecedente $i < n$, y falso implica todo. Luego, sabemos que $i < n$, y tiene sentido hablar de v_k , con $0 \leq k \leq i$. Sea $s \in S(i)$. Como $s \in S(i)$, sabemos que s termina con v_i . Sea s' el prefijo de s , es decir, $s = s' + [v_i]$, y por ende $|s| = |s'| + 1$. Entonces s' es una subsucesión de v , donde cada elemento es múltiplo del anterior. No sabemos si $s' = []$, pero si no lo es, termina en algún v_j , con $j < i$, $v_j \mid v_i$. Luego, si $s' \neq []$, entonces s' pertenece a la unión disjunta de $S(j)$, para algún $0 \leq j < i$. Si $s' = []$, entonces $|s| = |s'| + 1 = 0 + 1 = 1$. Ahora razonamos:

$$\begin{aligned}
g(i) &= \max\{|s| \mid s \in S(i)\} \\
&= \max\{\{1 + |s'| \mid s' \in S(j), 0 \leq j < i, v_j \mid v_i\} \cup \{1 + |s'| \mid s' \in \{[]\}\}\} \\
&= 1 + \max\{\{|s'| \mid s' \in S(j), 0 \leq j < i, v_j \mid v_i\} \cup \{|s'| \mid s' \in \{[]\}\}\} \\
&= 1 + \max\{\{|s'| \mid s' \in S(j), 0 \leq j < i, v_j \mid v_i\} \cup \{0\}\} \\
&= 1 + \max\{\{\max\{|s'| \mid s' \in S(j)\} \mid 0 \leq j < i, v_j \mid v_i\} \cup \{0\}\}, \text{ pues } S(j) \neq \emptyset \forall j \\
&= 1 + \max\{\{g(j) \mid 0 \leq j < i, v_j \mid v_i\} \cup \{0\}\} \\
&= 1 + \max\{\{f(j) \mid 0 \leq j < i, v_j \mid v_i\} \cup \{0\}\}, \text{ usando } P(j), \text{ pues } j < i \\
&= 1 + \max_0\{\{f(j) \mid 0 \leq j < i, v_j \mid v_i\}\} \\
&= f(i)
\end{aligned}$$

Luego vale $P(i)$.

3. Probemos ahora que, sabiendo que f tiene la semántica que dijimos, A es la respuesta al enunciado. Toda forma de borrar k elementos, dejando una subsecuencia de $n - k$ elementos sin borrar, tal que cada uno es múltiplo del anterior, es lo mismo que encontrar esa subsecuencia de $n - k$ elementos, y dejar sólo esos. Luego, la manera que borre *menos* elementos (que minimice k), es la manera que encuentre la subsecuencia *más* larga (maximice $n - k$).

Luego, podemos enfocarnos en encontrar la subsecuencia más larga donde cada elemento es múltiplo del anterior. El problema nos pide devolver cuántos elementos borramos, y eso es $n - k$, con k la longitud de tal secuencia.

Toda tal subsecuencia termina en alguna posición i . Luego, si tomamos $k = \max_{0 \leq i < n} f(i)$, sabiendo la semántica de f , habremos encontrado esa secuencia. Finalmente, al devolver $A = n - k$, estamos correctamente respondiendo el enunciado.

□

El código, en Python:

```

def F(v):
    n = len(v)
    def f(i):
        return 1 + max((f(j) for j in range(i))

```

```

        if v[j] != 0 and v[i] % v[j] == 0),
    default=0)
return n - max(f(i) for i in range(n))

```

Este algoritmo va a computar muchas veces cada valor de f . En el peor caso, donde tenemos $v = (1, 1, \dots, 1)$, cada vez que llamemos a $f(i)$ vamos a llamar a $f(j)$ para todo $0 \leq j < i$. Si T es el número de operaciones que hace, entonces $T(i) = O(n) + \sum_{j=0}^{i-1} T(j)$. Esto termina siendo $T \in \Theta(2^n)$.

Vemos que se cumplen las dos condiciones para usar programación dinámica:

1. Subestructura óptima. Para resolver $f(i)$, basta con resolver los problemas más pequeños que i , en particular, con calcular $f(j)$ para todo $0 \leq j < i$.
2. Subproblemas compartidos. A pesar de que hay sólo n posibles valores de f , la solución recursiva simple mira 2^n subproblemas en el peor caso (donde v es todos unos, esto sale de resolver la recurrencia $T(0) = 1; T(n) = 1 + \sum_{i=0}^{n-1} T(i)$). Luego hay muchos subproblemas compartidos, y programación dinámica probablemente haga más rápido nuestro algoritmo.

La manera mecánica de usar programación top-down es agregar un cache. Esto se convierte en:

```

def F(v):
    n = len(v)
    def f(i, cache={}):
        if i not in cache:
            cache[i] = 1 + max((f(j) for j in range(i)
                                if v[j] != 0 and v[i] % v[j] == 0),
                                default=0)
        return cache[i]
    return n - max(f(i) for i in range(n))

```

Es difícil analizar la complejidad temporal de este algoritmo, por estar mutando estado (cache), y depender su complejidad temporal del estado de cache en cada llamada. Podemos, entonces, usar programación dinámica bottom-up, llenando el cache en el mismo orden que se llenaría normalmente, pero a mano.

```

def F(v):
    n = len(v)
    dp = [1 for _ in range(n)]
    for i in range(1, n):
        dp[i] = 1 + max((dp[j] for j in range(i)
                          if v[j] != 0 and v[i] % v[j] == 0),
                          default=0)
    return n - max(dp[i] for i in range(n))

```

Esto nos deja entender el comportamiento asintótico muy fácilmente. El primer ciclo hace $\Theta(n^2)$ operaciones en todos los casos, y el segundo $\Theta(n)$ operaciones. Luego el algoritmo entero hace $\Theta(n^2)$ operaciones en el peor caso. Finalmente, la complejidad temporal del algoritmo es $\Theta(n)$, que es el costo de guardar la tabla dp , más variables auxiliares, que cuestan $O(1)$ cada una.

3.1 Ejercicios

Ejercicio 3.3.10

Se tienen dos arrays de n naturales, A y B . A está ordenado de manera creciente, y B de manera decreciente. Ningún valor aparece más de una vez en el mismo array. Para cada posición i , consideramos la diferencia absoluta entre los valores de los arrays, $|A[i] - B[i]|$. Se desea buscar el mínimo valor posible de dicha cuenta. Por ejemplo, si los arrays son $A = [1, 2, 3, 4]$, y $B = [6, 4, 2, 1]$, los valores de las diferencias son $[5, 2, 1, 3]$, y el resultado es 1.

1. Diseñar un algoritmo basado en divide-and-conquer que resuelva este problema.
2. Demostrar que es correcto.
3. Dar una cota superior ajustada de su complejidad temporal asintótica.

Ejercicio 3.3.11

Probar que el siguiente algoritmo multiplica dos enteros x, y dados, para cualquier valor entero de $c \geq 2$.

```
1: procedure F( $x \in \mathbb{Z}, y \in \mathbb{N}$ )
2:   if  $y = 0$  then
3:     return 0
4:   end
5:    $t \leftarrow F(c \times x, \lfloor \frac{y}{c} \rfloor)$ 
6:   return  $t + x \times (y \bmod c)$ 
7: end
```

Ejercicio 3.3.12

Diseñar un algoritmo que, dada una lista de longitud n con los primeros n números naturales, en orden, excepto un elemento faltante, encuentre tal elemento faltante. Por ejemplo, para la lista $[0, 1, 3, 4, 5]$, debe devolver 2, y para la lista $[1, 2, 3]$, debe devolver 0.

Probar formalmente que es correcto, y dar una cota superior ajustada de su complejidad temporal asintótica. Dicha cota debe estar en $O(\log n)$.

Ejercicio 3.3.13

Un array se dice monotónico si está compuesto por un prefijo de enteros creciente, y luego un sufijo de enteros decreciente. Por ejemplo, $[5, 8, 9, 3, 1]$ es unimodal.

Diseñar un algoritmo que, dado un array unimodal de longitud n , encuentre su valor máximo en tiempo $O(\log n)$. Demostrar formalmente que es correcto, y dar una cota superior ajustada de su complejidad temporal asintótica en el peor caso.

Ejercicio 3.3.14

Se tiene una escalera de n escalones. En cada momento, podemos subir de a un escalón, o de a tres escalones. Por ejemplo, si $n = 9$, desde el cuarto escalón podemos ir o bien al quinto, o al séptimo. Si estamos en el séptimo u octavo escalón, sólo podemos subir de a un escalón, y si estamos en el noveno escalón hemos terminado.

- Diseñar un algoritmo basado en programación dinámica que calcule de cuántas maneras distintas se puede subir la escalera entera.
- Demostrar formalmente que es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(n)$ operaciones en todo caso, y $O(1)$ memoria.

Ejercicio 3.3.15

Se tiene una grilla de $n \times m$ casillas. Empezamos en la casilla superior izquierda. En cada casilla nos podemos mover a la casilla de abajo, o a la casilla de la derecha. Por ejemplo, si estamos en la casilla $(2, 3)$, podemos ir a la casilla $(3, 3)$ o a la casilla $(2, 4)$. Si estamos en la última fila, sólo podemos movernos a la derecha, y si estamos en la última columna, sólo podemos movernos hacia abajo. Si llegamos a la casilla inferior derecha hemos terminado.

- Diseñar un algoritmo basado en programación dinámica que, dados dos enteros positivos n y m , calcule de cuántas maneras distintas se puede llegar a la casilla inferior derecha.
- Demostrar formalmente que es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(nm)$ operaciones en todo caso, y $O(\min(n, m))$ memoria.

Ejercicio 3.3.16

Similar al ejercicio anterior, pero ahora en cada casilla tenemos un entero, el entero en la celda (i, j) está en $A[i][j]$. Si el entero es positivo o cero, indica la ganancia que obtenemos al pasar por esa casilla. Si el entero es negativo, indica que no podemos pasar por esa casilla.

- Diseñar un algoritmo basado en programación dinámica que, dada una matriz A de $n \times m$ enteros, calcule la máxima ganancia posible al llegar a la casilla inferior derecha, partiendo desde la superior izquierda, y nuevamente yendo siempre o hacia abajo, o hacia la derecha.
- Demostrar formalmente que es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(nm)$ operaciones en todo caso, y $O(\min(n, m))$ memoria.

Ejercicio 3.3.17

Se tiene un array AA de l cadenas de ceros y unos, por ejemplo $A = [10,0001, 111, 100101, 111001, 1, 0]$, con $l = 7$. Dados enteros positivos n y m , encontrar el máximo número de cadenas de A con a lo sumo n unos y m ceros.

Sea k la suma de las longitudes de las cadenas en A , es decir, $k = \sum_{i=1}^l |A_i|$.

- Diseñar un algoritmo basado en programación dinámica que resuelva este problema.
- Demostrar formalmente que es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(lnm + k)$ operaciones en todo caso, y $O(nm)$ espacio.

Ejercicio 3.3.18

Dada una matriz A de $n \times m$ de enteros, encontrar la longitud del camino creciente más largo en A . El camino va desde una celda hacia arriba, abajo, a la derecha, o a la izquierda, pero no en diagonal.

Por ejemplo, en la siguiente matriz:

9	9	4
6	6	8
2	1	1

El camino creciente más largo es $[1, 2, 6, 9]$, de longitud 4.

- Diseñar un algoritmo basado en programación dinámica que resuelva este problema.
- Demostrar formalmente que es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(nm)$ operaciones en todo caso, y $O(nm)$ espacio.

Ejercicio 3.3.19

Tenemos un árbol arraigado, con n vértices. Podemos instalar cámaras en algunos vértices. Cada cámara puede monitorear al vértice donde está instalada, y a sus vecinos inmediatos, es decir su padre (si existe) y todos sus hijos inmediatos (si existen). Queremos saber cuántas cámaras como mínimo necesitamos para monitorear todos los vértices del árbol.

La entrada consiste de dos líneas. La primera línea contiene n , el número de vértices del árbol. La segunda línea contiene n enteros. El i -ésimo entero es el índice del vértice padre del vértice i , o -1 si se trata de la raíz del árbol.

Por ejemplo,

```
4  
-1 0 1 1
```

Esto representa el siguiente árbol:

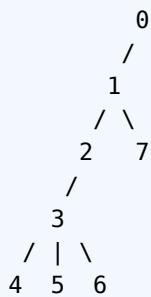


Y la respuesta correcta es 1, pues podemos instalar una cámara en el vértice 1 y monitorear todos los vértices.

La entrada

```
8  
-1 0 1 2 3 3 3 1
```

representa el siguiente árbol:



Y la respuesta correcta es 2, pues podemos instalar cámaras en los vértices 1 y 3.

- Diseñar un algoritmo basado en programación dinámica que calcule el número mínimo de cámaras necesarias para monitorear todos los vértices del árbol.
- Demostrar formalmente que es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(n)$ operaciones en todo caso, y $O(n)$ espacio.

Ejercicio 3.3.20

Se nos dan precios de acciones en días consecutivos en un array A , donde $A[i]$ es el precio de una acción el día i . También se nos da un entero k .

Queremos maximizar la ganancia total haciendo a lo sumo k transacciones. Una transacción es comprar una acción en un día, y venderla en un día posterior. No podemos tener más de una acción a la vez, es decir, debemos vender la acción antes de comprar otra.

Por ejemplo, si $A = [3, 2, 6, 5, 0, 3]$ y $k = 2$, podemos comprar en el día 2 (precio 2) y vender en el día 3 (precio 6), ganando 4, y luego comprar en el día 5 (precio 0) y vender en el día 6 (precio 3), ganando 3, para un total de 7.

- Diseñar un algoritmo basado en programación dinámica que calcule la máxima ganancia posible haciendo a lo sumo k transacciones.
- Demostrar formalmente que es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(nk)$ operaciones en todo caso, y $O(k)$ espacio.

Ejercicio 3.3.21

Se tiene una grilla de m filas y n columnas, con $m < n$. Cada celda debe ser pintada de color blanco o negro. Dos celdas son consideradas vecinas si comparten un borde y tienen el mismo color. Dos celdas A y B se consideran en la misma componente si son vecinas, o si hay un vecino de A en la misma componente que B .

Llamamos a una forma de pintar la grilla «linda» si tiene exactamente k componentes.

- Diseñar un algoritmo que, dados n y m , determine el número de formas lindas de pintar la grilla.
- Demostrar que el algoritmo es correcto.
- Dar una cota superior ajustada de su complejidad temporal asintótica, y demostrar que es correcta. Su algoritmo debería usar $O(n^2 4^m)$ operaciones en todo caso.

Sugerencia: Resolver para $m = 1$ y $m = 2$ antes de intentar el caso general.

4 Backtracking

A veces nuestra solución a un problema va a ser una exploración de un espacio de sub-soluciones, donde vamos construyendo la solución mediante una serie de elecciones a cada paso. Un algoritmo de backtracking es uno en el cual, ante una elección con varias opciones, probamos tomar una de las posibles opciones, nos fijamos si es posible extender esta opción a una solución final, y si no lo es, deshacemos esta elección y probamos con otra de las opciones.

Un ejemplo clásico es el de colocar n reinas en un tablero de ajedrez con n filas y n columnas, sin que se ninguna pueda atacar a otra en un movimiento. La idea de la solución es colocar una reina en una posición (i, j) del tablero, y resolver recursivamente el problema de colocar las $n - 1$ reinas restantes en el tablero resultante. Si no es posible hacer esto, sacamos a la reina, y la colocamos en otra posición. Si no existe una posición que podemos recursivamente completar, entonces decimos que no se puede continuar con el tablero que nos dan. El código queda así:

1: **procedure** N-QUEENS($n \in \mathbb{N}$)

```

2:   ▷  $c[i]$  contiene la columna donde pusimos una reina en la  $i$ -ésima fila, o 0 si no pusimos una
   reina en esa fila todavía.
3:    $c[1..n] \leftarrow 0$ 
4:   ▷  $f[i]$  indica si hay una reina en la fila  $i$ .
5:    $f[1..n] \leftarrow \text{FALSE}$ 
6:   ▷  $v[i]$  indica si hay una reina en la  $i$ -ésima diagonal «decreciente».
7:    $v[1..(2n)] \leftarrow \text{FALSE}$ 
8:   ▷  $w[i]$  indica si hay una reina en la  $i$ -ésima diagonal «creciente».
9:    $w[1..(2n)] \leftarrow \text{FALSE}$ 
10:  return Backtrack( $1, n, c, f, v, w$ )
11: end
12: procedure BACKTRACK( $r \in \mathbb{N}, n \in \mathbb{N}, c[1..n], f[1..n], v[1..(2n)], w[1..(2n)]$ )
13:   ▷ Caso base: Ya pusimos  $n$  reinas.
14:   if  $r > n$  then
15:     return  $c$ 
16:   end
17:   ▷ Intentamos poner una reina en la fila  $r$ , en cada columna  $j$ .
18:   for  $j = 1$  to  $n$  do
19:      $d_1 \leftarrow r - j + n$ 
20:      $d_2 \leftarrow r + j$ 
21:     ▷ Si no hay reinas que atacarían a una reina en  $(r, j)$  ya puestas.
22:     if  $\neg f[j]$  and  $\neg v[d_1]$  and  $\neg w[d_2]$  then
23:       ▷ Ponemos una reina en  $(r, j)$ .
24:        $c[r] \leftarrow j$ 
25:        $f[j] \leftarrow \text{TRUE}$ 
26:        $v[d_1] \leftarrow \text{TRUE}$ 
27:        $w[d_2] \leftarrow \text{TRUE}$ 
28:       ▷ Resolvemos recursivamente el tablero que queda.
29:        $z \leftarrow \text{Backtrack}(r + 1, n, c, f, v, w)$ 
30:       ▷ Si encontramos una solución recursivamente, la devolvemos.
31:       if  $z \neq \text{null}$  then
32:         return  $z$ 
33:       end
34:       ▷ Si no, deshacemos la elección que hicimos para la fila  $r$ .
35:        $f[j] \leftarrow \text{FALSE}$ 
36:        $v[d_1] \leftarrow \text{FALSE}$ 
37:        $w[d_2] \leftarrow \text{FALSE}$ 
38:     end
39:   end
40:   ▷ Si no devolvimos durante el ciclo, no hay forma de poner una reina en la fila  $r$ . El tablero que
      nos pasaron no es resoluble, y devolvemos NULL.
41:   return NULL
42: end

```

Vemos ahí las características clásicas de un algoritmo de backtracking:

- Construimos una solución global paso a paso, eligiendo opciones en cada paso, agregándolos a una sub-solución que tenemos.
- Al hacer una elección, nos fijamos recursivamente si podemos completar nuestra sub-solución actual con esta decisión, a una solución global.
- Si no pudimos hacer esto, deshacemos nuestra elección e intentamos con otra.
- Si ninguna opción funciona, el sub-problema que estamos resolviendo no es resoluble, y lo informamos al que nos llamó.

Demostrar la correctitud de estos algoritmos es un caso particular de correctitud de algoritmos

recursivos.¹⁷ Veamos cómo demostrar la correctitud de nuestro algoritmo.

Proposición 3.4.1

Dado un $n \in \mathbb{N}$:

- Si es posible colocar n reinas en un tablero de ajedrez de $n \times n$, de tal forma que ninguna pueda atacar a otra en un movimiento, `N-QUEENS(n)` devuelve una representación de algún tal tablero. El algoritmo devuelve una lista c de longitud n , tal que $1 \leq c[i] \leq n$ es la columna donde hay una reina en la fila i .
- Si no es posible, el algoritmo devuelve `NULL`.



Como lo único que hace `N-QUEENS` es llamar a `BACKTRACK`, vamos a demostrar una proposición sobre `BACKTRACK`.

Sea $n \in \mathbb{N}$, y $r \in \mathbb{N}$ tal que $1 \leq r \leq n + 1$. Definimos $P(r)$ como:

Definición 3.4.2

$P(r)$: Sea (c, f, v, w) una representación de un tablero de ajedrez de $n \times n$, con $r - 1$ reinas ya colocadas, y asumiendo que ningún par de las reinas ya colocadas se pueden atacar entre sí en un movimiento. Si es posible completar el tablero agregando $n - r + 1$ reinas de tal forma que ningún par de ellas se ataque en un movimiento, entonces `BACKTRACK(r, n, c, f, v, w)` devuelve una lista c' que representa un tal tablero. En particular, para cada $1 \leq i \leq n$, $c'[i]$ es la columna donde hay una reina en la fila i . Si no es posible completar el tablero, `BACKTRACK` devuelve `NULL`.



Lo primero que vemos es que `N-QUEENS` llama a `BACKTRACK` con $r = 1$, y una representación de un tablero vacío. $P(1)$ nos dice que `BACKTRACK` nos dice si es posible agregar $n - r + 1 = n - 1 + 1 = n$ a un tablero vacío (que es lo mismo que colocar n reinas), de tal forma que ningún par se ataque. Luego, si probamos $P(1)$, sabremos que `N-QUEENS` es correcto.

Demostración. Vamos a probar $P(k)$ para todo $1 \leq k \leq n + 1$ por inducción. Nuestro caso base va a ser $k = n + 1$, y vamos a usar $P(k + 1)$ para probar $P(k)$. Si los incomoda el hecho de hacer inducción «hacia atrás», pretendan que estamos probando $Q(k) : 1 \leq k \leq n \Rightarrow P(n - k + 1)$, pero realmente no hay ningún problema. Estamos construyendo una cadena de implicaciones, fundada en un caso base, como en toda inducción.

1. Caso base, $P(n + 1)$. Si $r = n + 1$, entonces ya hemos colocado $r - 1 = n$ reinas en el tablero que nos pasan, y sabemos que ningún par se ataca. Luego, existe un tal tablero que es solución al problema entero. Podemos devolver c , que indica en qué columna está la reina de cada fila, y es la representación de tal tablero. Vemos que `BACKTRACK($n + 1, n, c, f, v, w$)` devuelve precisamente c , y luego vale $P(n + 1)$.
2. Paso inductivo. Tenemos $r \in \mathbb{N}$, $1 \leq r \leq n$. Asumimos $P(r + 1)$, y queremos probar $P(r)$. Sea T el tablero que nos pasan, el representado por la 4-tupla (c, f, v, w) . Hay dos opciones, o bien T se puede completar a un tablero con n reinas, o no se puede.

¹⁷Por su estructura inductiva, donde la solución al problema global se crea tomando pasos en una sub-solución de un sub-problema similar, prácticamente siempre los algoritmos de backtracking están escritos de forma recursiva. No es *necesario*, pero sí lo más común.

- Si se puede completar, entre todos los tableros solución que son completaciones de este, sea T' cualquier tablero donde la reina que hay en la r -ésima fila está en la menor columna, y sea j tal columna. Vemos que el ciclo que hace BACKTRACK prueba las columnas posibles donde poner la r -ésima reina en orden creciente, y luego como ningún tablero se puede completar poniendo una reina en $c[r] = j'$ con $j' < j$, todas esas iteraciones van a terminar sin devolver nada. Luego, llegaremos a la j -ésima iteración. En esa iteración, vamos a poner una reina en $c[r] \leftarrow j$, y escribimos f, v , y w correspondientemente. Como T' existe, y el tablero que acabamos de armar es un sub-tablero de T' y también una completación parcial de T , estamos llamando a BACKTRACK $(r+1, \dots)$ con un tablero que es completable a una solución global T' . Como asumimos $P(r+1)$ por inducción, sabemos que esta llamada a BACKTRACK va a devolver un tablero solución (no necesariamente T' !). Es decir, $z \neq \text{null}$. Luego, al devolver z , BACKTRACK está devolviendo un tablero solución que completa T , que es el comportamiento esperado, y prueba $P(r)$.
- En el caso contrario, no existe ninguna manera de completar T para obtener un tablero con n reinas. Si el ciclo devolviera en algún momento, tendríamos por $P(r+1)$ un tablero z que extiende a T a n reinas, pero no existe. Luego, en ningún momento del ciclo podemos devolver, y el ciclo recorre todas sus iteraciones y termina. BACKTRACK entonces devuelve NULL , que es la respuesta correcta en este caso. Esto entonces prueba $P(r)$.

En ambos casos probamos $P(r)$, partiendo de $P(r+1)$. Por inducción, probamos $P(1)$, y esto completa la demostración.

□

Analizar el comportamiento asintótico de este algoritmo es también simple. Notemos cómo, al haber retornos dentro del ciclo, no vamos a poder saber exactamente cuántas iteraciones va a hacer en cada llamada. Vamos a definir $X(n)$ como el número de operaciones que realiza $\text{N-QUEENS}(n)$. Vamos a encontrar una función T tal que $X \in O(T)$.

Proposición 3.4.3

$T(n) = (n+1)!$ es tal que $X \in O(T)$.

♥

Demostración. Sea $n \in \mathbb{N}$. Veamos cuántas operaciones realiza $\text{BACKTRACK}(r, n, \dots)$, sabiendo que tenemos $r \leq n+1$ durante todo el algoritmo. Llamemos $B(k)$ a una cota superior al número de operaciones que realiza, cuando $k = n - r + 1$. Vamos a definirla por inducción.

- Si $k = 0$, entonces $r = n+1$. En este caso, BACKTRACK retorna inmediatamente, realizando un número constante de operaciones. Luego $B(0) = q$ para algún número de operaciones fijo q .
- Si $k > 0$, entonces $r < n+1$, y BACKTRACK va a hacer algunas iteraciones, y en algunas va a llamarse recursivamente. Podríamos acotar este número de llamadas recursivas por n trivialmente, pues el ciclo es de 1 a n , pero si tenemos más cuidado, vemos que ya hay $r-1$ reinas en el tablero que nos dan. Por lo tanto, al verificar si $\neg f[j]$, va a haber $r-1$ valores para los cuales $f[j] = \text{true}$, y por lo tanto no haremos llamadas recursivas. Entonces, hay a lo sumo $n - (r-1) = n - r + 1 = k$ llamadas recursivas, no n . Por hipótesis inductiva,

cada llamada recursiva de la forma $\text{BACKTRACK}(r + 1, n, \dots)$, realiza a lo sumo $B(k - 1)$ operaciones (pues aumentar r es disminuir k). Luego, como también tenemos un costo de iterar n veces y escribir/leer arrays (que cuestan $O(1)$ operaciones), realizaremos a lo sumo $B(k) = kB(k - 1) + O(n)$ operaciones.

Luego, podemos acotar por arriba el número de operaciones que realiza $\text{BACKTRACK}(r, n, \dots)$ por $B(n - r + 1)$, donde B es la siguiente función:

$$B(k) = \begin{cases} q & \text{si } k = 0 \\ kB(k - 1) + O(n) & \text{si } k > 0 \end{cases}$$

Para ser claros, lo que estamos diciendo explícitamente es que existe una función $g : \mathbb{N} \rightarrow \mathbb{N}$, con $g \in O(n)$, tal que

$$B(k) = \begin{cases} q & \text{si } k = 0 \\ kB(k - 1) + g(n) & \text{si } k > 0 \end{cases}$$

Intentemos encontrar una forma cerrada para B . Si $g(n) = 0$, es fácil ver que $B(k) = k!q$. Veamos entonces qué pasa comparando $B(k)$ con $k!$. Definimos $C(k) = \frac{B(k)}{k!}$. Cuando $k > 0$, tenemos

$$\begin{aligned} C(k) &= \frac{B(k)}{k!} \\ &= \frac{B(k - 1)}{(k - 1)!} + \frac{g(n)}{k!} \\ &= C(k - 1) + \frac{g(n)}{k!} \end{aligned}$$

Entonces tenemos $C(k) = q + \sum_{i=1}^k \frac{g(n)}{i!}$. Luego, $B(k) = k!C(k) = k!\left(q + \sum_{i=1}^k \frac{g(n)}{i!}\right) = k!\left(q + g(n) \sum_{i=1}^k \frac{1}{i!}\right)$. Recordando que $\sum_{i=0}^{\infty} \frac{1}{i!} = e$, tenemos que $B(k) \leq k!(q + g(n)(e - 1)) \leq k!(q + 2g(n))$. Finalmente, vemos que $B \in O(n!(q + 2g(n))) \subseteq O(g(n)n!) \subseteq O(nn!) \subseteq O((n + 1)!)$.

Recordando que $\text{N-QUEENS}(n)$ llama a $\text{BACKTRACK}(r = 1, n, \dots)$, y este hace a lo sumo $B(n - r + 1) = B(n - 1 + 1) = B(n)$ operaciones, tenemos que $X(n) \leq B(n)$, con $B \in O((n + 1)!)$, y luego $X \in O((n + 1)!)$. Luego, existe una función $T(n) = (n + 1)!$, tal que $X \in O(T)$. □

5 Greedy

Estos algoritmos son similares a los de backtracking, excepto que al tomar una decisión, nunca la deshacemos. Son entonces «más rápidos» que una solución de backtracking, dado que exploran menos del espacio de sub-soluciones. Sin embargo, demostrar que son correctos va a requerir demostrar que las elecciones que hacen nunca son incorrectas. Es decir, nunca nos llevan desde una sub-solución que puede completarse a una solución global, a una que no puede completarse de tal forma.

Para muchos problemas, una solución greedy es fácil de imaginar. Por ejemplo, en el problema de la mochila, podemos pensar en «siempre tomo el objeto más valioso», o «siempre tomo el objeto más liviano». Lo difícil está en probar su correctitud.

Hay varias maneras de probar la correctitud de los algoritmos greedy. En general, van a tener mucha flexibilidad para probar este tipo de algoritmos correctos. Tres formas clásicas son:

- Argumentos de intercambio, donde tomamos una solución óptima mejor a la nuestra lo más parecida posible a la nuestra, y hacemos un intercambio en esa solución que la hace o mejor que lo que era, o más parecida a la nuestra.
- Argumentos de liderazgo («greedy stays ahead»), donde definimos una métrica para sub-soluciones, y una noción de «longitud» para las mismas. Luego argumentamos que nuestra sub-solución siempre mejor, en esta métrica, a cualquier otra sub-solución de la misma «longitud».
- Argumentos de completación, donde argumentamos que en todo momento nuestra sub-solución puede ser completada a una solución óptima.

Vamos a mostrarles algunos ejemplos de cada una.

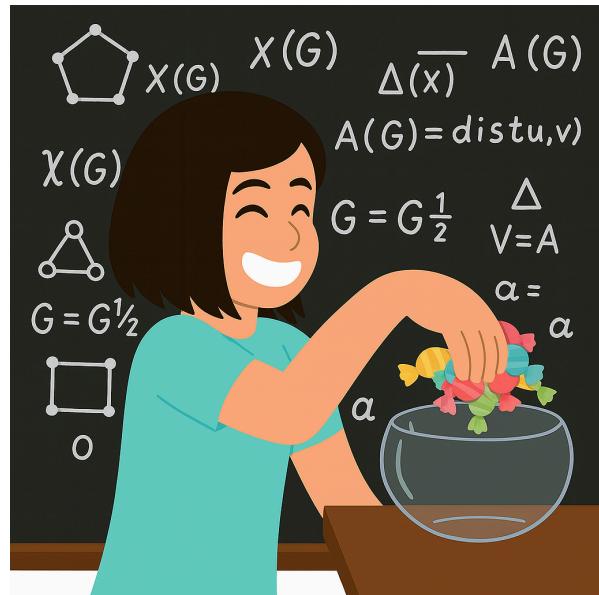
Ejercicio 3.5.1

Tomás quiere viajar de Buenos Aires a Mar del Plata en su flamante Renault 12. Como está preocupado por la autonomía de su vehículo, se tomó el tiempo de anotar las distintas estaciones de servicio que se encuentran en el camino. Modeló el mismo como un segmento de 0 a M , donde Buenos aires está en el kilómetro 0, Mar del Plata en el $M > 0$, y las distintas estaciones de servicio están ubicadas en los kilómetros $0 = x_1 \leq x_2 \leq \dots \leq x_n = M$. Razonablemente, Tomás quiere minimizar el número de paradas para cargar nafta. Él sabe que su auto es capaz de hacer hasta C kilómetros con el tanque lleno, y que al comenzar el viaje este está vacío.

1. Diseñar un algoritmo greedy que le indique a Tomás en qué estaciones debe detenerse para cargar nafta, de forma tal que el número de paradas sea mínimo.
2. Demostrar su correctitud.
3. Mostrar la complejidad temporal y espacial del algoritmo. El mejor algoritmo posible tiene complejidad temporal y espacial $O(n)$.

Solución. Un posible algoritmo greedy para este problema es el siguiente:

```
def g(estaciones: list[int], C: int) -> list[int] | None:
    n = len(estaciones)
    i = 0
    cargas = []
    while i != n - 1:
        j = i
```



```

while j < n - 1 and estaciones[j+1] - estaciones[i] <= C:
    j += 1
if j == i: return None
cargas.append(i + 1)
i = j
return cargas

```

Vamos a dar tres demostraciones de la correctitud de este algoritmo. La primer demostración es un argumento del estilo greedy-stays-ahead.

Demostración. Definimos una solución como una sucesión estrictamente creciente de índices, $S = (s_1, \dots, s_h) \subseteq \{1, \dots, n\}$, tal que $s_1 = 1$ (se carga en el kilómetro cero, la primer estación), $s_h = n$ (se llega a Mar Del Plata desde la última estación), y tal que para todo $1 \leq j < h$, $|x_{s_{j+1}} - x_{s_j}| \leq C$ (se puede llegar de una estación a la otra con el tanque lleno). Notar que podemos simplificar la última condición a $x_{s_{j+1}} - x_{s_j} \leq C$, pues las estaciones están ordenadas crecientemente en x . En general, el auto se puede mover de la estación x_i a la estación x_j cuando $|x_i - x_j| \leq C$.

Si el problema no tiene solución, es porque ni siquiera cargando en cada estación podemos llegar. Esto implica que existe al menos una estación i , tal que $x_{i+1} - x_i > C$. En este caso, nuestro algoritmo devuelve `None` al considerar `if j == i: return None`, y es la respuesta correcta. Por otra parte, si nuestro algoritmo devuelve `None` es porque no puede llegar a la próxima estación desde la actual (i), y por lo tanto no existe ninguna solución, pues toda solución tiene que pasar por x_i , cargue o no cargue ahí, y llegar a x_{i+1} . Luego, en este caso nuestro algoritmo es correcto, pues devuelve `None` si y sólo si no existe solución.

Asumimos entonces que el problema tiene solución. Sea $G = \{g_1, \dots, g_k\}$ una solución producida por nuestro algoritmo. Sea $O = \{o_1, \dots, o_h\}$ cualquier solución óptima, con lo cual $h \leq k$. Vamos a demostrar la siguiente proposición por inducción, para todo $t \in \mathbb{N}, t \geq 1$.

$$P(t) : \text{Si } t \leq h, \text{ entonces } o_t \leq g_t.$$

1. Caso base, $t = 1$. Como empezamos con el tanque vacío en $x_1 = 0$, y $M > 0$, el auto tiene que moverse y no tiene combustible al empezar. Luego toda solución debe cargar en la primer estación, y $g_1 = o_1 = 1$. Luego vale $P(1)$.
2. Paso inductivo. Sabemos que vale $P(t)$, queremos demostrar $P(t + 1)$. Si $t = h$, entonces $P(t + 1)$ es trivial, pues es una implicación con antecedente falso. Luego basta considerar $t < h$, y por lo tanto $t + 1 \leq h$. Sabemos por $P(t)$ que $o_t \leq g_t$. Sea $A_t = \{j \mid j > g_t, x_j - x_{g_t} \leq C\}$ el conjunto de estaciones alcanzables yendo hacia adelante desde g_t . Como el problema tiene solución, $A_t \neq \emptyset$, y podemos llegar al menos a la próxima estación. Sea entonces $j^* = \max A_t$. Nuestro algoritmo elige $g_{t+1} = j^*$.

Como O es una solución válida, $|x_{o_{t+1}} - x_{o_t}| \leq C$, y como los valores de O están ordenados de forma creciente, $x_{o_{t+1}} > x_{o_t}$, y por lo tanto $x_{o_{t+1}} - x_{o_t} \leq C$. Luego, como $o_t \leq g_t$, tenemos $x_{o_t} \leq x_{g_t}$, y luego $x_{o_{t+1}} - x_{g_t} \leq C$. Notemos que o_{t+1} puede

ser mayor o menor que g_t , y por lo tanto el lado izquierdo de esta suma puede ser negativo. Partimos en casos:

- Si $o_{t+1} < g_t$, entonces $x_{g_{t+1}} \geq x_{g_t} \geq x_{o_{t+1}}$. Como las estaciones están ordenadas crecientemente en x , concluimos $g_{t+1} \geq o_{t+1}$, y luego vale $P(t+1)$.
- Si $o_{t+1} \geq g_t$, entonces $o_{t+1} \in A_t$, y por lo tanto $o_{t+1} \leq j^* = g_{t+1}$. Luego vale $P(t+1)$.

Luego vale $P(t)$ para todo $t \in \mathbb{N}, t \geq 1$. En particular, vale $P(h)$, y vemos que $o_h \leq g_h$. Como O es una solución, $o_h = n$. Como nuestro algoritmo sólo devuelve estaciones, $g_h \leq n$. Luego, $g_h = n$, y G es una solución. Como $g_h = n - 1$, nuestro algoritmo agregó $g_h = i + 1$, y por lo tanto $i = n - 1$. Nuestro algoritmo se detiene cuando $i = n - 1$, con lo cual $k = h$. Luego, como $|G| = k = h = |O|$, y O es una solución óptima, G también es una solución óptima. \square

La siguiente demostración usa un argumento de intercambio.

Demostración. En la demostración anterior vimos que si el problema no tiene solución, nuestro algoritmo devuelve **None**, y si devuelve **None**, el problema no tiene solución. Luego, en este caso el algoritmo es correcto. Asumimos entonces que el problema tiene solución.

Sea $G = \{g_1, \dots, g_k\}$ la solución que devuelve nuestro algoritmo. De todas las soluciones óptimas, consideremos la solución $O = \{o_1, \dots, o_h\}$ que tenga un prefijo más largo en común con G . Sea i la longitud de ese prefijo en común. Como O es óptima, $h \leq k$. Como i es la longitud de un prefijo de O , $i \leq h$, y a fortiori $i \leq k$.

Si $i = h$, entonces o_i es la estación final. Como $g_i = o_i$, nuestro algoritmo habría llegado a la estación final, y terminaría, con lo cual $i = k$, y tendríamos $G = O$, mostrando el algoritmo correcto.

Si no, entonces $i < h$, y existe o_{i+1} . Como $i < h$ y $h \leq k$, entonces $i < k$, y también existe g_{i+1} . Como i es la longitud del prefijo en común entre G y O , entonces $g_{i+1} \neq o_{i+1}$.

Nuestro algoritmo eligió a g_{i+1} como la estación más lejana a la que puede llegar, habiendo cargado en g_i . Por ende, como O es una solución válida, y $g_i = o_i$, debemos tener $o_i = g_i \leq o_{i+1} < g_{i+1}$. Como O está ordenada crecientemente, entonces $g_{i+1} \notin \{o_1, \dots, o_i\}$. Consideremos entonces $X = \{o_1, o_2, \dots, o_i, g_{i+1}\}$. Al tener $X_1 = o_1, \dots, X_i = o_i$, y O siendo una solución, el prefijo de longitud i de X es válido. Para ver que $X_{i+1} - X_i \leq C$, basta ver que $X_i = o_i = g_i$ y $X_{i+1} = g_{i+1}$, y sabíamos que $g_{i+1} - g_i \leq C$ pues nuestro algoritmo sólo hace saltos con distancia menor o igual a C . Por lo tanto, el salto de X_i a X_{i+1} es válido.

Sea $t = \min\{t \mid t \in [i+2, \dots, h], o_t > g_{i+1}\}$. Primero vemos que el conjunto no es vacío: De otra forma, g_{i+1} sería la última estación (recordando que x_n debe estar en O para ser solución válida), y luego $i+1 = k$ es donde termina el algoritmo. Como $h =$

$|O| \geq i + 1 = k$, entonces G sería una solución óptima. Basta entonces considerar el caso en que el conjunto no es vacío, y por lo tanto t está bien definido.

Consideremos entonces $Y = \{o_t, o_{t+1}, \dots, o_h\}$. Claramente los saltos son válidos pues es un sufijo de O . Finalmente, vamos a considerar $O^* = X \cup Y$. Lo único que tenemos que ver para ver que O^* es una solución válida es que $Y_1 - X_{i+1} \leq C$. Esto es ver que $o_t - g_{i+1} \leq C$. Como O es una solución, $o_t - o_{t-1} \leq C$. Por cómo definimos t , $o_{t-1} \leq g_{i+1}$. Por lo tanto, tenemos $o_t - g_{i+1} \leq o_t - o_{t-1} \leq C$, que es lo que queríamos demostrar.

Luego, O^* es una solución válida. Su longitud es $|X| + |Y|$, con $|X| = i + 1$, y $|Y| = h - t + 1$. Como $t \geq i + 2$, tenemos que $|O^*| = i + 1 + h - t + 1 \leq i + 2 + h - (i + 2) = h$, con lo cual O^* es una solución óptima.

Vemos que O^* es una solución óptima que tiene un prefijo de longitud $i + 1$ en común con G , pero esto no puede suceder, pues O era una solución óptima con el más largo prefijo en común con G , y ese prefijo tenía longitud i . Luego tal O no existe, y G ya es óptima. \square

Finalmente, veamos una demostración usando un argumento de completación.

Demostración. Al igual que antes, vamos a dar por demostrado (porque lo hicimos en la primer demostración) que el algoritmo es correcto cuando no hay solución. Luego vamos a asumir acá que hay solución.

Vamos a usar el teorema del invariante para probar la correctitud de nuestro algoritmo. Entre iteraciones del ciclo exterior, nuestro algoritmo mantiene un array cargas = $[g_1, \dots, g_t]$, donde t es el número de iteraciones completadas. Como vamos a usar el teorema del invariante, necesitamos explicitar las siguientes proposiciones:

- Guarda $B(i)$: $i \neq n - 1$.
- Función variante $V(i) = n - 1 - i \in \mathbb{N}$.
- Precondición P : $0 = x_1 \leq x_2 \leq \dots \leq x_n = M$, $C \geq 0$, y el problema tiene solución.
- Postcondición Q : G es una solución óptima.
- Invariante $I(i)$:
 1. $0 \leq i < n$.
 2. Si $t > 0$, entonces $x_{i+1} - x_{g_t} \leq C$. Es decir, podemos llegar desde la última estación donde cargamos ($x_{g_p} = \text{estaciones}[\text{cargas}[t - 1] - 1]$), hasta la estación actual ($x_{i+1} = \text{estaciones}[i]$). El -1 al indexar `estaciones` es porque `cargas` contiene índices comenzando en 1, mientras que en Python los índices de arrays comienzan en 0.
 3. Si $t > 0$ y $B(i)$, entonces para todo $r > i + 1$, $x_r - x_{g_t} > C$. Es decir, mientras que podemos llegar a x_{i+1} , pero no podemos llegar a ninguna estación posterior.
 4. $\exists O = \{o_1, \dots, o_h\}$ solución óptima $|(o_1, \dots, o_t) = (g_1, \dots, g_t)$

Y necesitamos demostrar las siguientes proposiciones:

- La función variante decrece en cada iteración. Como siempre asignamos $i \leftarrow j$, con $j > i$, vemos que $V(i) = n - 1 - i$ decrece en cada iteración.
- Si la función variante se anula, la guarda es falsa. Si $V(i) = 0$, entonces $i = n - 1$, que es precisamente $\neg B(i)$.
- La negación de la guarda, y el invariante, implican la postcondición. Si tenemos la negación de la guarda ($\neg B(i)$), tenemos $\neg(i \neq n - 1) \Leftrightarrow i = n - 1$, y como si también vale el invariante ($I(i)$) tenemos $I(n - 1)$. Luego existe una solución óptima O tal que $(o_1, \dots, o_t) = (g_1, \dots, g_t)$, y como $i = n - 1$, y si $t > 0$, estaciones[i] – estaciones[cargas[t – 1] – 1] = estaciones[n – 1] – estaciones[cargas[t – 1] – 1] = $x_n - x_{g_t} \leq C$, entonces podemos llegar desde la última estación de cargas hasta la estación final, Mar del Plata. Como existe una solución óptima O que es extensión de G , y G ya puede llegar a Mar del Plata, entonces O debe ser G . Luego, $G = \{g_1, \dots, g_t\}$ es una solución válida. Como O es una solución óptima, entonces O no puede tener ninguna estación posterior a la n -ésima, y O no sólo *extiende* a G , sino que *es* G . Luego G es una solución óptima. Luego vale la postcondición.
- El invariante vale inicialmente, $I(0)$. El estado inicial es $i = 0$, cargas = $[]$. La lista vacía es trivialmente extensible a cualquier solución óptima. Por la precondición, existe al menos una solución, y luego existe una solución óptima. Las otras condiciones de $I(0)$ son o simples (pues $0 \leq 0 < n$) o trivialmente ciertas (pues $\neg(t > 0)$). Luego, $I(0)$ vale.
- Si el invariante vale antes de una iteración, y la guarda es verdadera, entonces el invariante vale después de la iteración. Asumimos que valen $I(i) \wedge B(i)$. Vamos a probar $I(j)$. Notemos que j es el valor que obtiene i al terminar cada iteración. Tenemos un pequeño ciclo interior que calcula $j = \max\{l \mid i \leq l < n, x_{l+1} - x_{i+1} \leq C\}$ (recordemos que los índices de x empiezan en 1, mientras que en Python estamos indexando estaciones a partir de 0, por eso los +1).
 1. Por cómo definimos j , tenemos que $j \geq i$, y luego $j \geq i \geq 0$. Asimismo definimos j como el máximo de un conjunto cuyo elementos más grande es a lo sumo $n - 1$, luego $j \leq n - 1$, es decir $j < n$. Luego $0 \leq j < n$, que es la primer parte de $I(j)$.
 2. Llamemos $p = i + 1$. Estamos agregando p a G , obteniendo $G' = \{g_1, \dots, g_t, i + 1\}$. Tenemos que ver que $x_{j+1} - x_p \leq C$. Esto es justamente cómo definimos j , como el mayor índice que cumple $x_{j+1} - x_p \leq C$. Esta es la segunda cláusula de $I(j)$, recordando que $g_t = i + 1$ pues lo acabamos de insertar.
 3. Si $B(j)$, entonces $j \neq n - 1$. Si ocurriese $x_n - x_{i+1} \leq C$, entonces j no sería el máximo índice que cumple $x_{j+1} - x_{i+1} \leq C$, pues $n - 1$ también lo cumpliría, y dijimos que $j \neq n - 1$. Esto no puede suceder, entonces $x_n - x_{i+1} > C$. Esta es la tercera cláusula de $I(j)$.
 4. Sabemos por $I(i)$ que existe una solución óptima O que extiende a G . Si $t = 0$, entonces al comenzar con el tanque vacío, vamos a agregar la primer parada, $x_1 = 1$. Como toda solución tiene que hacer esto, en particular cualquier solución óptima tiene que hacer esto, y tenemos que G sigue siendo extensible a una solución óptima. Si $t > 0$, entonces como vale $B(i)$, por $I(i)$ tenemos que $x_n - x_{g_t} > C$. Luego toda solución óptima, en particular O , necesita al menos una estación más, y luego existe $o_{t+1} > g_t$. Sea $p = i + 1$.

Por $I(i)$ y $B(i)$, sabemos que para todo $r > i + 1$, $x_r - x_{g_t} > C$. También sabemos que $g_t = o_t$, y $p = i + 1$. Luego, para todo $r > p$, tenemos $x_r - x_{o_t} > C$. Como O es una solución válida, O no puede entonces seguir a o_t con x_r para ningún $r > p$, y tenemos $s \leq p$.

Si $s = p$, ya tenemos en O un prefijo de longitud $i + 1$ en común con G , y G sigue siendo extensible a una solución óptima. Luego debemos ver el caso $s < p$.

Si $s < p$, entonces O carga antes de p . Recordemos que O tiene la forma $O = \{o_1, \dots, o_h\} = \{g_1, \dots, g_t, s, o_{t+2}, \dots, o_h\}$. Quisiéramos reemplazar a s por p , pero no sabemos si $o_{t+2} \geq p$, y si esto no sucediera, no estaríamos formando una solución válida (recordando que los índices en una solución son crecientes).

Consideremos el primer índice j en O , tal que $j > t + 1$, y $o_j > p$.

Si j no existe, entonces O está llegando a Mar del Plata desde s , y como $p > s$, también vamos a poder llegar a Mar del Plata desde p . El algoritmo encontrará entonces que $j == i$, y terminará, con lo cual tenemos una solución $G = \{g_1, \dots, g_t, p\}$ con longitud $t + 1$, que es también la longitud de $O = \{o_1, \dots, o_t, s\}$.

Si por otro lado j existe, consideremos $O' = \{o_1, \dots, o_t, p, o_j, o_{j+1}, \dots, o_h\}$. Claramente $G' = \{g_1, \dots, g_t, p\}$ es un prefijo de O' . El único salto que hay que justificar en O' es entre p y o_j . Como j es el primer índice mayor que $t + 1$ tal que $o_j > p$, tenemos que $o_{j-1} \leq p$. Como O es una solución válida, $x_{o_j} - x_{o_{j-1}} \leq C$. Luego, como $o_{j-1} \leq p$, tenemos $x_{o_j} - x_p \leq C$. Luego, todos los saltos de O' son válidos. También por cómo definimos j , tenemos $o_j \geq p$, y luego O' está ordenada crecientemente. Finalmente, $|O'| \leq |O|$ pues removimos los elementos $\{o_{t+2}, \dots, o_{j-1}\}$ (puede ser vacío este conjunto), y como O es óptima, O' también lo es.

Luego, O' es una solución óptima que extiende a G' , y tenemos la última cláusula de $I(j)$.

Por el teorema del invariante, entonces, vale la postcondición al terminar el ciclo, y nuestro algoritmo es correcto. \square

Ejercicio 3.5.2

Queremos devolver el vuelto a un cliente, y tenemos monedas de 1, 5, 10, y 25 centavos. Diseñar un algoritmo greedy que resuelva el problema. Demostrar su correctitud.

Demostración. Un algoritmo como el que nos piden es el siguiente, donde n es el número de centavos que queremos devolver:

```

1: procedure GREEDYCHANGE( $n \in \mathbb{N}$ )
2:    $C \leftarrow [1, 5, 10, 25]$ 
3:    $v \leftarrow []$ 
4:   while  $n > 0$  do

```

```

5:    $c \leftarrow \max_{c \in C} \{c \mid c \leq n\}$ 
6:    $n \leftarrow n - c$ 
7:    $v \leftarrow v + [c]$ 
8: end
9: return  $v$ 
10: end

```

La semántica que le vamos a asignar a GREEDYCHANGE es que $\text{GREEDYCHANGE}(n)$ devuelve una lista de denominaciones de monedas, tal que la suma de las denominaciones es n , y el número de monedas es mínimo entre todas las formas de sumar n con esas denominaciones.

Primero, veamos que GREEDYCHANGE termina. En cada iteración, n decrece en como mínimo 1, pues $1 \leq n$ si entramos al ciclo, puesto que la guarda es $n > 0$, es decir $n \geq 1$. n nunca se hace negativo, porque siempre seleccionamos un c tal que $c \leq n$, y luego $n - c \geq 0$. Luego, como en cada iteración decrece, vuelve al ciclo cada vez que n es positivo, y n nunca se hace negativo, sabemos que al final del ciclo, $n = 0$. Como cada vez que restamos c a n , agregamos c a v , vemos que $\sum_{c \in v} c = n$, y luego v es una forma de devolver el vuelto de n centavos. Asimismo, vemos que como n decrece en cada iteración, y c siempre es la máxima denominación menor a n , entonces c no puede crecer de una iteración a la otra, y luego v es llenado en orden de mayor a menor denominación.

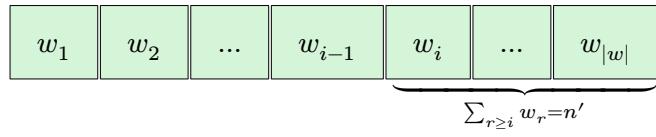
Ahora veamos que GREEDYCHANGE devuelve una lista v de mínima longitud, tal que v contiene sólo elementos de C , y $\sum_{c \in v} c = n$. Vamos a hacer esto mediante un **argumento de intercambio**. Supongamos que existe una forma w de devolver el vuelto de n centavos, con menos monedas que v . De todas las posibles w de mínima longitud, tomemos cualquier que, al ser ordenada de forma no-creciente, tenga el máximo número de elementos en común con v . Es decir, w maximiza $\max\{i \mid 0 \leq i \leq \min(|w|, |v|), \forall 0 \leq j < i, w_j = v_j\}$, donde estamos ordenando w de forma no-creciente, entre todas las soluciones óptimas.

Sea i ese número. Entonces sabemos que antes de i , w y v tienen los mismos elementos, mientras que $v_i \neq w_i$. Como GREEDYCHANGE produjo v de forma no-creciente, al elegir v_i , teníamos que $v_i = \max_{c \in C} \{c \mid c \leq n'\}$, con $n' < n$ el cambio que hace falta hacer todavía. Entonces, $n' = n - \sum_{j=1}^{i-1} v_j = \sum_{j=1}^{i-1} w_j$, porque dijimos que para todos esos índices j , $v_j = w_j$. Luego, como $n = \sum_{j=1}^{|w|} w_j \geq \sum_{j=1}^i w_j = n - n' + w_i$, o vemos que $0 \geq -n' + w_i$, o también, $w_i \leq n'$. Como elegimos v_i como el máximo $c \in C$ tal que $c \leq n'$, y $w_i \in C$, sabemos que $w_i < v_i$.

Vamos a querer obtener, en cada caso, una forma de obtener una solución de menor largo que w , o de igual largo pero que tiene un prefijo más largo en común con v . Como w era una forma óptima que tiene el prefijo más largo con común con v , esto es una contradicción. Por lo tanto w no puede existir, y luego v es una solución óptima. Recordemos que $|w| < |v|$, por cómo definimos w .

v_1	v_2	\dots	v_{i-1}	v_i	\dots	$v_{ v }$
		:		‡		

$\overbrace{\quad \quad \quad \quad \quad \quad \quad}^{\sum_{j \geq i} v_j = n'}$



Partimos en casos:

1. Si $v_i = 25$, entonces $n' \geq 25$, y $w_i < 25$. Como ordenamos w de forma no-creciente, y el resto de w tiene que sumar $n' \geq 25$, consideremos qué monedas está usando w , a partir de w_i . Llamemos a estas monedas $z = [w_j \mid |w| \geq j \geq i]$. z va a ser de la forma $z = \left[\underbrace{10, \dots, 10}_{a \text{ dieces}}, \underbrace{5, \dots, 5}_{b \text{ cincos}}, \underbrace{1, \dots, 1}_{c \text{ unos}} \right]$, con $10a + 5b + c = n'$.
 1. Si $a \geq 3$, entonces podemos reemplazar tres 10s por [25, 5], obteniendo una solución más corta que w .
 2. Si $a = 2, b \geq 1$, entonces podemos reemplazar dos 10s y un 5 por [25]. Si $b = 0$, podemos reemplazar dos 10s y cinco 1s por [25].
 3. Si $a = 1, b \geq 3$, podemos reemplazar un 10 y tres 5s por [25]. Si $b = 2$, podemos reemplazar un 10, dos 5s y cinco 1s por [25]. Si $b = 1$, podemos reemplazar un 10, un 5 y diez 1s por [25]. Si $b = 0$, podemos reemplazar un 10 y quince 1s por [25].
 4. Si $a = 0$, vemos que cualquier combinación no-creciente de 5s y 1s que sume $n' \geq 25$ va a tener un prefijo que sume 25, y podemos ahí reemplazar ese prefijo monedas por [25]. En todos los casos, obteniendo una solución más corta que w .
2. Si $v_i = 10$, entonces $n' \geq 10$, y como $w_i < v_i$, entonces con la misma construcción que arriba, obtenemos $z = \left[\underbrace{5, \dots, 5}_{a \text{ cincos}}, \underbrace{1, \dots, 1}_{b \text{ unos}} \right]$, con $5a + b = n' \geq 10$.
 1. Si $a \geq 2$, entonces podemos reemplazar dos 5s por [10].
 2. Si $a = 1$, entonces podemos reemplazar un 5 y cinco 1s por [10].
 3. Si $a = 0$, entonces podemos reemplazar diez 1s por [10].
3. Si $v_i = 5$, entonces $n' \geq 5$, y como $w_i < v_i$, entonces con la misma construcción que arriba, obtenemos $z = \left[\underbrace{1, \dots, 1}_{a \text{ unos}} \right]$, con $a = n' \geq 5$. Podemos entonces reemplazar cinco 1s por [5].
4. No podemos tener $v_i = 1$, porque no es posible tener $w_i \in C, w_i < 1$.

En todos los casos, obtenemos una solución más corta que w , lo cual contradice que w era una solución óptima. Luego, no puede existir tal w , y por lo tanto v es una solución óptima. \square

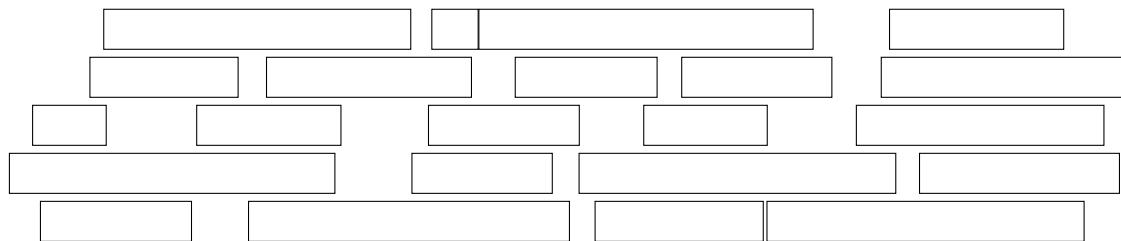
Ejercicio 3.5.3

Tenemos un aula a nuestra disposición, y n clases que se pueden dar en ese aula. La i -ésima clase empieza a la hora s_i , y termina a la hora f_i . Dos clases no se pueden dar al mismo tiempo en ese aula. Queremos saber cual es el máximo número de clases que se pueden dictar en ese aula.

Diseñar un algoritmo greedy que resuelva este ejercicio. Probar que es correcto y probar su complejidad temporal y espacial asintótica.

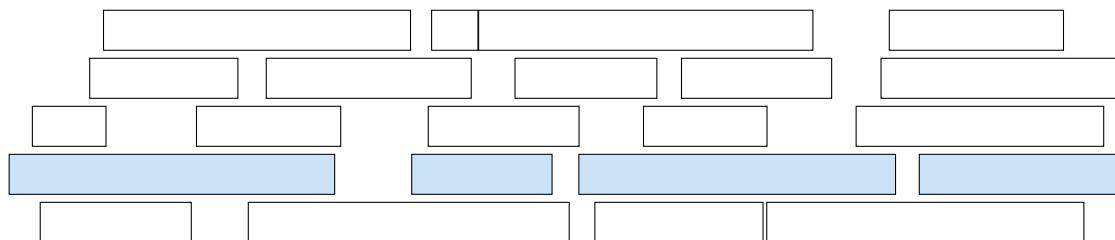
Cuando encontramos un problema nuevo y queremos ver si existe una estrategia greedy para resolverlo, lo primero que tenemos que hacer es intentar definir una estructura de «subproblemas», y adivinar cuál va a ser una estrategia de selección local (es decir, que toma decisiones en cada subproblema, sin replantearlas si «sale mal»).

Sirve entonces plantearse ejemplos, y ver cómo nuestras ideas funcionan o no.

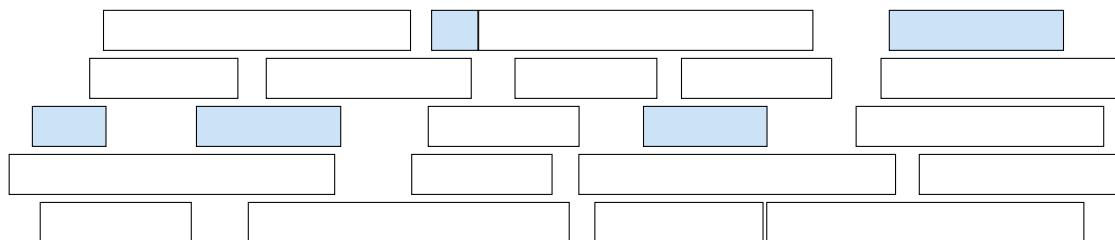


Pensemos en un par de estrategias para este ejemplo.

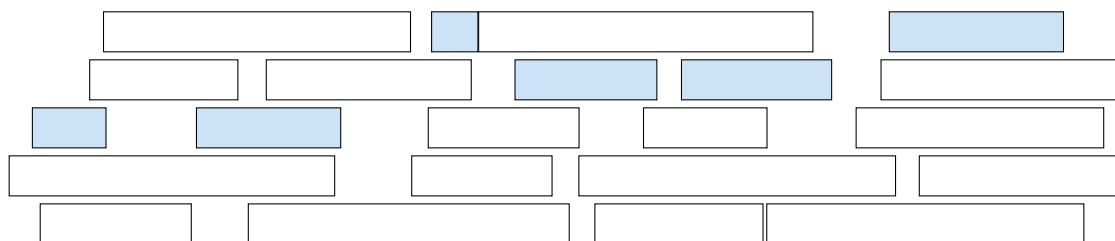
- Podemos ordenar las materias por hora de comienzo, y agarrar la materia que empieze lo antes posible, que no tenga conflictos con materias ya seleccionadas. Esto nos da un subconjunto de tamaño 4.



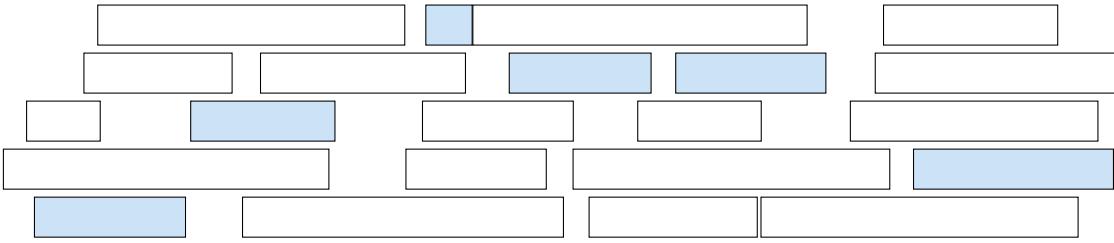
- Podemos ordenar las materias por duración, eligiendo las más cortas que no causen conflictos con materias ya seleccionadas. Esto nos da un subconjunto de tamaño 5.



- Podemos ordenar las materias por hora de finalización, y agarrar la materia que termine lo antes posible, que no tenga conflictos con materias ya seleccionadas. Esto nos da un subconjunto de tamaño 6.



Si buscamos a fuerza bruta, una solución óptima para este ejemplo tiene tamaño 6.



Esto nos sugiere intentar probar que la estrategia de ordenar por momento de finalización, creciente, puede ser buena.

Solución. Proponemos el siguiente algoritmo.

```

1: procedure GREEDYCOURSES( $M = [(s_1, f_1), \dots, (s_n, f_n)] : \text{List}[\mathbb{N} \times \mathbb{N}]$ )
2:    $A \leftarrow []$ 
3:    $I \leftarrow \text{sort}([1, \dots, n], \text{key: } \lambda i. f_i)$ 
4:    $c \leftarrow 0$ 
5:   for  $i \in I$  do
6:     if  $s_i \geq c$  then
7:        $c \leftarrow f_i$ 
8:        $A \leftarrow A + [i]$ 
9:     end
10:   end
11:   return  $A$ 
12: end
```

Demostración. Para esta demostración les vamos a mostrar un argumento de liderazgo («greedy stays ahead»). Para esto tenemos que definir una estructura de «subproblemas» que nuestro algoritmo greedy resuelve en orden. Luego tenemos que dar una estructura de sub-solución a cualquier solución. Finalmente, vamos a dar una noción de «valor» para las sub-soluciones, y probamos que la i -ésima sub-solución de nuestro algoritmo tiene «mejor» valor que la i -ésima sub-solución de cualquier solución. «Mejor» puede lo vamos a definir a veces como «menor», y a veces como «mayor», dependiendo del problema.

Sean $A = \{i_1, \dots, i_k\}$ los índices de las materias que devuelve nuestro algoritmo, en el orden en que fueron agregados. Sea $O = \{j_1, \dots, j_m\}$ los índices de las materias en una solución óptima, ordenados por momento de finalización. Sea $F(X) = \max_{x \in X} f_x$ el máximo tiempo de finalización entre todas las materias en un conjunto X . Notar que por cómo definimos A y O , tenemos que $F(\{i_1, \dots, i_r\}) = F(\{i_r\}) = f_{i_r}$ y $F(\{j_1, \dots, j_r\}) = F(\{j_r\}) = f_{j_r}$, para todo r .

Las sub-soluciones de nuestro algoritmo son $\{i_1, \dots, i_r\}$, para cada r . Las sub-soluciones de una solución óptima son $\{j_1, \dots, j_r\}$, para cada r . La noción de valor que vamos a usar es F , el máximo tiempo de finalización entre las materias en la sub-solución. Vamos a mostrar, entonces, que para todo $1 \leq r \leq k$, tenemos $P(r) : F(\{i_1, \dots, i_r\}) \leq F(\{j_1, \dots, j_r\})$. Probemos P .

1. Caso base, $P(1)$. Como A eligió inicialmente la materia que antes termina en M , sabemos que $f_{i_1} \leq f_r \forall r$. En particular, vemos que $f_{i_1} \leq f_{j_1}$.
2. Paso inductivo. Sea $k \geq t \geq 2 \in \mathbb{N}$, sabemos que $P(t-1)$, queremos probar $P(t)$. $P(t-1)$ nos dice que $f_{i_{t-1}} \leq f_{j_{t-1}}$. Todas las materias que pueden seguirle a j_{t-1} en O empiezan después de $f_{j_{t-1}}$, luego como eso viene después que $f_{i_{t-1}}$, también son candidatas para agregar a A . Luego, si i_t es la siguiente materia que agrega A , entonces $f_{i_t} \leq f_{j_t}$, pues A toma la que antes termine, de todas las candidatas. Esto muestra que $P(t)$ es cierto.

Eso demuestra que vale $P(r)$ para todo $1 \leq r \leq k$. En particular, vale para $r = k$, y tenemos que $F(A) = F(\{i_1, \dots, i_k\}) \leq F(\{j_1, \dots, j_k\})$.

Si $m > k$, entonces existe j_{k+1} , una materia que la solución óptima O eligió, que no está en A . Como ordenamos O , debemos tener que $s_{j_{k+1}} \geq f_{j_k}$. Por $P(k)$, sabemos que $f_{j_k} \geq f_{i_k}$. Por lo tanto, j_{k+1} es una materia que empieza después de que termine la última materia que agregó A . Es más, como k fue la última materia que agregó A , j_{k+1} empieza después que *todas* las materias en A . Entonces, al iterar por $i = j_{k+1}$, A la hubiera agregado, y no lo hizo. Esto no puede suceder, y por lo tanto $m \leq k$. Luego, A es una solución óptima. \square

Ejercicio 3.5.4

Tenemos n items, cada uno con valor v_i y peso p_i . Tenemos una mochila que puede aguantar un peso máximo P . Para este problema, podemos tomar pedazos fraccionarios de cada objeto. Por ejemplo, si tenemos una manzana de 300 gramos y valor 6, podemos tomar un tercio de la manzana, de peso 100 gramos y valor 2.

Queremos maximizar el valor de los objetos que llevamos en la mochila, sin exceder el peso máximo P . Podemos asumir que $\frac{v_i}{p_i} \neq \frac{v_j}{p_j} \forall i, j$, es decir, que no hay dos objetos con el mismo valor por unidad de peso.

Diseñar un algoritmo greedy que resuelva este ejercicio. Probar que es correcto y probar su complejidad temporal y espacial asintótica.



Solución. Primero formalicemos un poco la consigna. Lo que queremos hacer es encontrar un vector $x = (x_1, \dots, x_n)$, donde $0 \leq x_i \leq 1$ para cada i , que maximice la suma

$$\sum_{i=1}^n v_i x_i$$

sujetos a

$$\sum_{i=1}^n p_i x_i \leq P$$

Una estrategia greedy para resolver esto va a ser repetidamente elegir algún elemento, mientras que quepa en la mochila. Si sólo cabe una fracción del objeto, metemos sólo esa fracción en la mochila. Veamos un ejemplo, con peso máximo $P = 40$.

Pesos	4	50	14	52	13	1	45	3
Valores	2	42	4	24	2	3	42	10

Algunas ideas para cómo elegir los objetos:

1. Podemos elegir el objeto con mayor valor, y meterlo en la mochila. Si no cabe entero, metemos la fracción que quepa. En este problema, elegiríamos ordenaríamos los objetos como $[2, 7, 4, 8, 3, 6, 1, 5]$. Luego intentaríamos meter el 2do objeto, que tiene peso 50 y valor 42. Como no cabe entero puesto que su peso es mayor a $P = 40$, meteríamos $\frac{40}{50}$ de este objeto, que aporta un valor de $\frac{40}{50} \times 42 = 33.6$.
2. Podemos elegir el objeto con menor peso, y meterlo en la mochila. Si no cabe entero, metemos la fracción que quepa. En este problema, ordenaríamos los objetos como $[6, 8, 1, 5, 3, 7, 2, 4]$. Podemos meter los objetos 6, 8, 1, 5, y 3, enteros, obteniendo un valor de $3 + 10 + 2 + 2 + 4 = 21$. Para cuando vemos el 7mo objeto, nuestra mochila ya tiene un peso de $1 + 3 + 4 + 13 + 14 = 35$, luego sólo podemos meter $\frac{5}{45}$ de este objeto, que nos agrega un valor de $\frac{5}{45} \times 42 = 4.67$, obteniendo un valor final de $21 + 4.67 = 25.67$.
3. Podemos elegir el objeto con mayor valor por unidad de peso ($\frac{v_i}{w_i}$), y meterlo en la mochila. Si no cabe entero, metemos la fracción que quepa. En este caso, ordenaríamos los objetos como $[8, 6, 7, 2, 1, 4, 3, 5]$. Agregamos los objetos 8 y 6, obteniendo un valor de 13, y un peso de 4. Luego vemos el 7mo objeto, y como tenemos 36 de peso para llenar, y el objeto pesa 45, sólo podemos meter $\frac{36}{45}$ de este objeto, que nos agrega un valor de $\frac{36}{45} \times 42 = 33.6$, obteniendo un valor final de $13 + 33.6 = 46.6$.

De estas tres ideas, la última fue la que mejor valor nos dio. Intentemos, entonces, ese algoritmo.

```

1: procedure FRACTIONALKNAPSACK( $P \in \mathbb{N}$ ,  $p : \mathbb{N}^n$ ,  $v : \mathbb{N}^n$ )
2:    $A \leftarrow []$ 
3:    $I \leftarrow \text{sort}([1, \dots, n], \text{key: } \lambda i. \frac{v_i}{p_i})$ 
4:    $c \leftarrow 0$ 
5:    $v \leftarrow 0$ 
6:   for  $i \in I$  do
7:     if  $c + p_i \leq P$  then
8:        $c \leftarrow c + p_i$ 
9:        $v \leftarrow v + v_i$ 
10:       $A \leftarrow A + [(i, 1.0)]$ 
11:    else
12:       $x \leftarrow P - \frac{c}{p_i}$ 
13:       $v \leftarrow v + x * v_i$ 
14:       $A \leftarrow A + [(i, x)]$ 
15:      BREAK
16:    end
17:  end
18:  return  $(A, v)$ 
19: end

```

Demostración. Para este problema vamos a usar un argumento de intercambio. Vamos a considerar una solución óptima que es distinta y mejor a la nuestra, y vamos a ver que podemos mejorar esta solución óptima, lo cual nos diría que no puede existir.

ⓘ Nota

Notemos que en general *no* vamos a poder probar que nuestra solución es la *única* solución óptima. Puede haber muchas, y nuestro algoritmo va a elegir una sola. No vamos a poder probar, entonces, que no existe una solución óptima distinta que la nuestra. Lo que vamos a poder probar es que no hay una solución *mejor* que la nuestra.

Ordenemos los objetos por su valor por unidad de peso, $\frac{v_i}{p_i}$, de forma no-creciente. Luego, tenemos que $\frac{v_i}{p_i} \geq \frac{v_j}{p_j}$ para todo $i \leq j$.

Sea $A = [x_1, \dots, x_n]$ las fracciones de cada elemento que eligió nuestro algoritmo, y sea $O = [y_1, \dots, y_n]$ una solución óptima. Por cómo ordenamos los objetos en esta demostración, nuestro algoritmo eligió primero un valor para x_1 , luego un valor para x_2 , etcétera. Notemos que $0 \leq x_i \leq 1$ y $0 \leq y_i \leq 1$ para todo i , porque estamos eligiendo fracciones de los objetos, y no podemos elegir más de un objeto entero.

Supongamos que A no es óptima. Entonces $\sum_{i=1}^n v_i x_i < \sum_{i=1}^n v_i y_i$. Sea i el primer índice donde $x_i \neq y_i$. Luego, $\sum_{j=1}^{i-1} x_j p_j = \sum_{j=1}^{i-1} y_j p_j$. Al momento de elegir x_i , nuestro algoritmo podía usar un peso máximo de $P' = P - \sum_{j=1}^{i-1} x_j p_j$. Nuestro algoritmo elige $x_i = \frac{P'}{p_i}$.

Asimismo, como O es una solución, tenemos que $\sum_{j=1}^n y_j p_j \leq P$. Restando $\sum_{j=1}^{i-1} x_j p_j$ de cada lado, tenemos que $\sum_{j=i}^n y_j p_j \leq P'$. Como todos los pesos son positivos, así como también los y_j , esto implica que $y_i p_i \leq P'$. Esto es lo mismo que decir $y_i \leq \frac{P'}{p_i} = x_i$. Como $y_i \leq x_i$ y $y_i \neq x_i$, sabemos que $y_i < x_i$. Como O es óptima, y tiene más peso disponible en la mochila, tiene que existir algún $j > i$ tal que $y_j > x_j$. Si no fuera así, entonces O no es óptima, porque A tiene más valor que O .

Sea $\varepsilon = \min\left(1 - y_i, y_j \frac{p_j}{p_i}\right)$. Vemos que, como $\varepsilon \leq 1 - y_i$, entonces $y_i + \varepsilon \leq 1$.

También, como $\varepsilon \leq y_j \frac{p_j}{p_i}$, entonces $y_j - \varepsilon \frac{p_i}{p_j} \geq 0$. Por lo tanto, podemos definir una nueva solución z , cuyos valores están siempre entre 0 y 1, como:

$$z_r = \begin{cases} y_i + \varepsilon, & \text{si } r = i \\ y_j - \varepsilon \frac{p_i}{p_j}, & \text{si } r = j \\ y_r, & \text{si } r \neq i \wedge r \neq j \end{cases}$$

Veamos que el peso total de z es el mismo que el de O .

$$\begin{aligned}
\sum_{r=1}^n z_r p_r &= \sum_{r=1}^n y_r p_r - y_i p_i - y_j p_j + (y_i + \varepsilon) p_i + \left(y_j - \varepsilon \frac{p_i}{p_j} \right) p_j \\
&= \sum_{r=1}^n y_r p_r - y_i p_i - y_j p_j + y_i p_i + \varepsilon p_i + y_j p_j - \varepsilon p_i \\
&= \sum_{r=1}^n y_r p_r
\end{aligned}$$

Y luego z es una solución válida. Sin embargo, veamos que el valor de z es mayor al de O :

$$\begin{aligned}
\sum_{r=1}^n z_r v_r &= \sum_{r=1}^n y_r v_r - y_i v_i - y_j v_j + (y_i + \varepsilon) v_i + \left(y_j - \varepsilon \frac{p_i}{p_j} \right) v_j \\
&= \sum_{r=1}^n y_r v_r - y_i v_i - y_j v_j + y_i v_i + \varepsilon v_i + y_j v_j - \varepsilon \frac{p_i}{p_j} v_j \\
&= \sum_{r=1}^n y_r v_r + \varepsilon \left(v_i - \frac{p_i}{p_j} v_j \right)
\end{aligned}$$

Ahora bien:

$$\begin{aligned}
v_i - \frac{p_i}{p_j} v_j &\geq 0 \\
\frac{v_i}{p_i} &\geq \frac{v_j}{p_j}
\end{aligned}$$

Pues así ordenamos los objetos. El problema nos dice que no hay dos objetos con el mismo valor por unidad de peso, entonces ese \geq es en realidad un $>$, pues $j > i$, y luego son objetos distintos.

Por otro lado, sabemos que $y_i < x_i \leq 1$, y luego $y_i < 1$, y entonces $1 - y_i > 0$.

Asimismo, como $y_j > x_j \geq 0$, entonces $y_j > 0$, y luego $y_j \frac{p_j}{p_i} > 0$. Luego, al ser $\varepsilon = \min(1 - y_i, y_j \frac{p_j}{p_i}) > 0$, tenemos que $\varepsilon \left(v_i - \frac{p_i}{p_j} v_j \right) > 0$.

Luego, $\sum_{r=1}^n z_r v_r > \sum_{r=1}^n y_r v_r$, lo cual contradice que O es óptima. Luego, no puede existir una tal solución óptima, O , que es estrictamente mejor que la nuestra, A . \square

Ejercicio 3.5.5

Tenemos dos conjuntos de personas y para cada persona sabemos su habilidad de baile. Queremos armar la máxima cantidad de parejas de baile, sabiendo que para cada pareja debemos elegir exactamente una persona de cada conjunto de modo que la diferencia de habilidad sea menor o igual a 1 (en modulo). Ademas, cada persona puede pertenecer a lo sumo a una pareja de baile. Por ejemplo, si tenemos un multiconjunto con habilidades $\{1, 2, 4, 6\}$ y otro con $\{1, 5, 5, 7, 9\}$, la maxima cantidad de parejas es 3. Si los multiconjuntos de habilidades son $\{1, 1, 1, 1, 1\}$ y $\{1, 2, 3\}$, la máxima cantidad es 2.

Diseñar un algoritmo greedy que resuelva este ejercicio. Probar que es correcto y probar su complejidad temporal y espacial asintótica.

Demostración.

Veamos un algoritmo recursivo simple que resuelve el problema.

```
def F(G: [int], B: [int]) -> int:
    return f(sorted(G), sorted(B))

def f(G: [int], B: [int]) -> int:
    if not G or not B:
        return 0
    g = G[0]
    b = B[0]
    if b < g - 1:
        return f(G, B[1:])
    if g < b - 1:
        return f(G[1:], B)
    return 1 + f(G[1:], B[1:])
```

Queremos ver que $F(G, B)$ devuelve el máximo número de parejas que se pueden armar con los multiconjuntos (en este programa, listas) de habilidades dados por G y B . Vamos a probar que $f(G', B')$ devuelve la respuesta que $F(G, B)$ tiene que dar, asumiendo que G' y B' tienen los mismos elementos que G y B respectivamente, sólo permutando su orden, lo cual no cambia la solución esperada, dado que estamos cambiando la representación del multiconjunto, pero no sus elementos.

Lo vamos a hacer por inducción. Como en f estamos sacando siempre un elemento de B o de G (y a veces de ambos), vamos a definir:

$tamaño(G, B) = \text{len}(G) + \text{len}(B)$

Esto nos va a ayudar porque nuestras llamadas recursivas siempre llaman a f con instancias de menor tamaño que la que recibe. Esto es lo que necesitamos para hacer inducción.

Definimos entonces:

$P(k) : f(G, B)$ es correcta para todos los argumentos (G, B)
de tamaño(G, B) a lo sumo k .

1. Caso base

El caso base es $P(0)$. Tenemos que probar que $f(G, B)$ es correcto para todos los argumentos de tamaño a lo sumo $k = 0$. Si $\text{tamaño}(G, B) = 0$, entonces $\text{len}(G) + \text{len}(B) = 0$, pero entonces como ambos $\text{len}(G)$ y $\text{len}(B)$ son enteros no-negativos, tenemos que $\text{len}(G) = \text{len}(B) = 0$. Luego, $G = B = []$. Entonces, no hay parejas posibles, porque una pareja tendría que tener un elemento de G y otro de B . Luego la respuesta correcta es 0 , y efectivamente nuestro programa devuelve 0 , en:

```
if not G or not B:  
    return 0
```

2. Paso inductivo

Ahora probemos el paso inductivo. Asumo $P(k)$, y quiero probar $P(k + 1)$.

Me dan un par (G, B) con $\text{tamaño}(G, B) = k + 1$. Si uno de los dos está vacío (no pueden ambos estar vacíos porque $\text{tamaño}(G, B) = k + 1 \geq 1$), no hay parejas posibles, luego la respuesta correcta es 0 , y el programa efectivamente devuelve eso

```
if not G or not B:  
    return 0
```

Si ninguno está vacío, ambos tienen un primer elemento, llamémoslo $g_0 \in G$ y $b_0 \in B$. Como G y B están ordenados crecientes, $g_0 \leq g \forall g \in G$, y $b_0 \leq b \forall b \in B$.

Consideremos ahora b_0 versus g_0 .

- Si $b_0 < g_0 - 1$, entonces b_0 no puede estar aparejado con g_0 . Pero más aún, como $g_0 \leq g \forall g \in G$, todos los elementos de G son al menos tan grandes como g_0 , y luego b_0 no puede estar aparejado con nadie. Luego, toda solución a (G, B) es lo mismo que una solución a $(G, B \setminus \{b_0\})$. Como $\text{tamaño}(G, B \setminus \{b_0\}) = k < k + 1$, sabemos por inducción que f es correcta para ese caso, y luego nuestra f es también correcta para este caso de (G, B) , porque devolvemos exactamente $f(G, B \setminus \{b_0\})$:

```
if b < g - 1:  
    return f(G, B[1:])
```

- Con un argumento similar, si $g_0 < b_0 - 1$, g_0 no puede estar aparejado con b_0 , y b_0 es menor o igual que todos los elementos en B , luego g_0 no puede estar aparejado con nadie en B , y luego toda solución a (G, B) deja a g_0 sin usar, y luego es idéntica a una solución a $(G \setminus \{g_0\}, B)$. Luego nuestra f es correcta para este caso de (G, B) , porque devolvemos exactamente $f(G \setminus \{g_0\}, B)$, y como $\text{tamaño}(G \setminus \{g_0\}, B) = k < k + 1$, por hipótesis inductiva f devuelve una solución óptima para ese caso:

```
if g < b - 1:  
    return f(G[1:], B)
```

Si ninguna de las dos condiciones vale, entonces $g_0 \geq b_0 - 1$, y $b_0 \geq g_0 - 1$. Esto es lo mismo que $b_0 - g_0 \leq 1$, y $g_0 - b_0 \leq 1$, o lo que es lo mismo, $|b_0 - g_0| \leq 1$, es decir, b_0 y g_0 son compatibles. Notemos que en nuestro programa esta es la última rama, donde devolvemos $1 + f(G[1:], B[1:])$. Vamos a probar dos lemas:

1. Si existe una solución óptima para (G, B) donde (g_0, b_0) están aparejados, entonces f es correcta para (G, B) .
2. Existe una solución óptima para (G, B) donde (g_0, b_0) están aparejados.

Está claro que si probamos ambos lemas, probamos que f es correcta para (G, B) . Como (G, B) era cualquier argumento de tamaño $k + 1$, esto prueba que $P(k) \Rightarrow P(k + 1)$, que es lo que queríamos demostrar.

Lema 3.5.6

Queremos probar que «Si existe una solución óptima para (G, B) donde (g_0, b_0) están aparejados, entonces f es correcta para (G, B) .»



Demostración. Supongamos que existe una solución óptima para (G, B) donde (g_0, b_0) están aparejados. Sea S esa solución. Luego, $S' = S \setminus \{(g_0, b_0)\}$ es una forma de aparejar a $G \setminus \{g_0\}$ y $B \setminus \{b_0\}$.

Podemos decir algo más fuerte: S' es óptima para $(G \setminus \{g_0\}, B \setminus \{b_0\})$. Si no lo fuera, sea S^* una solución a $(G \setminus \{g_0\}, B \setminus \{b_0\})$ que tiene más elementos que S' . Como S^* no menciona a g_0 ni a b_0 , podemos construir $S^* \cup \{(g_0, b_0)\}$, que tiene $|S^*| + 1 > |S'| + 1 = |S|$ elementos, y es una solución a aparejar a G y B . Esto no puede pasar, porque S era óptima para (G, B) , no puedo tener a $S^* \cup \{(g_0, b_0)\}$ más grande que S .

Luego, S' es óptima para $(G \setminus \{g_0\}, B \setminus \{b_0\})$. Por inducción, f es correcta para $(G \setminus \{g_0\}, B \setminus \{b_0\})$, porque tamaño($G \setminus \{g_0\}, B \setminus \{b_0\}$) = $k - 1 < k + 1$.

Luego $|S'| = |f(G \setminus \{g_0\}, B \setminus \{b_0\})|$, y luego $|S| = 1 + |S'| = 1 + |f(G \setminus \{g_0\}, B \setminus \{b_0\})|$. Luego f es correcta para (G, B) , dado que devolvemos exactamente eso:

```
return 1 + f(G[1:], B[1:])
```



Lema 3.5.7

Queremos probar que «Existe una solución óptima para (G, B) donde (g_0, b_0) están aparejados.» Sabemos que g_0 y b_0 son compatibles.



Demostración. Sea S cualquier solución óptima a (G, B) . Sean S_G y S_B los elementos de G y B que *no* aparecen en S , respectivamente. Leér «single girls, single boys».

Partimos en casos:

- Si $g_0 \in S_G$ y $b_0 \in S_B$, podríamos considerar $S' = S \cup \{(g_0, b_0)\}$, que tiene un elemento más que S , y es solución a (G, B) . Como S era solución óptima, esto no puede pasar.
- Si $g_0 \in S_G$ pero $b_0 \notin S_B$, sabemos que existe una pareja $(x, b_0) \in S$. Luego, podemos considerar $S' = (S \setminus \{(x, b_0)\}) \cup \{(g_0, b_0)\}$. Esto deja sin aparejar a x ,

pero empareja a g_0 y b_0 , y como tiene el mismo número de elementos que S , vemos que S' es una solución óptima para (G, B) donde g_0 y b_0 están aparejados.

- Si $b_0 \in S_B$ pero $g_0 \notin S_G$, sabemos que existe una pareja $(g_0, y) \in S$. Luego, podemos considerar $S' = (S \setminus \{(g_0, y)\}) \cup \{(g_0, b_0)\}$. Esto deja sin aparejar a y , pero empareja a g_0 y b_0 , y como tiene el mismo número de elementos que S , vemos que S' es una solución óptima para (G, B) donde g_0 y b_0 están aparejados.
- Si $g_0 \notin S_G$ y $b_0 \notin S_B$, entonces ambos están aparejados. Hay dos opciones:
 - Si están aparejados el uno al otro, es decir, (g_0, b_0) está en S , ya está, conseguimos una solución óptima que contiene a (g_0, b_0) , y es S .
 - Si no, existen $(x, b_0) \in S$ y $(g_0, y) \in S$. Quisieramos aparejar a b_0 con g_0 , y a x con y , y para eso tenemos que probar que x e y son compatibles. Como b_0 era el elemento en B más chico, tenemos que $y \geq b_0$. De la misma manera sabemos que $x \geq g_0$. Veamos qué puede pasar:
 - Si $b_0 = g_0$, es decir, tienen la misma altura, entonces $y \geq b_0 = g_0$, y como en S están aparejados y con g_0 , y sólo puede ser g_0 o $g_0 + 1$. Por el mismo motivo, $x \geq g_0 = b_0$, y en S están aparejados x con b_0 , y entonces x sólo puede ser b_0 o $b_0 + 1$. Como b_0 y g_0 son el mismo número, ambos x e y sólo pueden ser b_0 o $b_0 + 1$, y luego x e y son compatibles.
 - Si $b_0 = g_0 + 1$, entonces $y \geq b_0 = g_0 + 1$, y como en S están aparejados y con g_0 , tenemos que y es *exactamente* $g_0 + 1$. Luego tenemos que $y = g_0 + 1 = b_0$. Como x y b_0 son compatibles, y $b_0 = y$, vemos que x e y son compatibles.
 - Si $g_0 = b_0 + 1$, pasa lo mismo que en el caso anterior. Tenemos $x \geq g_0 = b_0 + 1$, y como x y b_0 están aparejados en S , tenemos que x es *exactamente* $b_0 + 1$. Luego tenemos que $x = b_0 + 1 = g_0$. Como y y g_0 son compatibles, y $g_0 = x$, vemos que x e y son compatibles.

Como en todos los casos x e y son compatibles, podemos considerar $S' = (S \setminus \{(x, b_0), (g_0, y)\}) \cup \{(g_0, b_0), (x, y)\}$. Esta es una solución a (G, B) que apareja a g_0 y b_0 , y que tiene el mismo número de elementos que S , y por lo tanto también es óptima para (G, B) .

Vemos entonces que siempre existe una solución óptima para (G, B) donde g_0 y b_0 están aparejados.

□

□

Ejercicio 3.5.8

Resolver el ejercicio anterior dando un algoritmo iterativo, en vez de recursivo.



Demostración. Veamos ahora una resolución iterativa del mismo ejercicio.

Nota: Se asume acá que los multiconjuntos están representados por listas no-decrecientes.

```
def f(G: [int], B: [int]) -> int:
    n = len(G)
    m = len(B)
    i = 0
    j = 0
    res = 0
    while i < n and j < m:
        g = G[i]
        b = B[j]
        if b < g - 1:
            j += 1
        elif g < b - 1:
            i += 1
        else:
            i += 1
            j += 1
            res += 1
    return res
```

Nuestro invariante va a ser que $0 \leq i \leq n$, $0 \leq j \leq m$, y que existe un conjunto S , un conjunto T , y una solución óptima S^* , tal que:

1. $S \subseteq G[0...i] \times B[0...j]$
2. $T \subseteq G[i...n] \times B[j...m]$
3. $S^* = S \sqcup T$, con \sqcup siendo la unión disjunta
4. $\text{res} = |S|$

Esto se puede entender como que S es «extensible» a una solución óptima S^* , usando sólo elementos que vienen no antes que i en G y j en B . Llamemos a esta noción « (i, j) -extensible».

El invariante vale inicialmente

Claramente vale el invariante antes de entrar al ciclo, dado que $i = j = 0$, y $G[0 \dots i] = G[0 \dots 0] = []$, $B[0 \dots j] = B[0 \dots 0] = []$. Con ambos G y B vacíos, no se puede formar ninguna pareja, y luego definimos $S = \emptyset$, cuyo tamaño es exactamente $\text{res} = 0$. Asimismo, vemos que *toda* solución global S^* , es una extensión de \emptyset , agregándole parejas en $G[i...n] \times B[j...m] = G[0...n] \times B[0...m] = G \times B$. En particular, definimos $T = S^*$, y tenemos que $S^* = \emptyset \sqcup T$.

Los despiertos habrán notado que ese párrafo asume que *existe* una solución óptima. Un «para todo x en X , vale $P(x)$ » sólo implica «existe x en X tal que $P(x)$ » cuando X no es vacío. El conjunto de todos los emparejamientos posibles no es vacío (por ejemplo, podemos tomar el emparejamiento vacío), y es finito (está incluido en $\mathcal{P}(G \times B)$, el conjunto de partes de $G \times B$). Luego tiene al menos un elemento de tamaño máximo, y luego *existe* al menos una solución óptima.

El invariante es preservado por las iteraciones

Ahora veamos que si vale la guarda, y vale el invariante al entrar al cuerpo del ciclo, vale el invariante al finalizar el cuerpo del ciclo. Como vale la guarda, sabemos que $0 \leq i < n$, y $0 \leq$

$j < m$. Vemos fácilmente que al final del cuerpo del loop sigue valiendo $0 \leq i \leq n, 0 \leq j \leq m$, porque sólo los aumentamos en uno en el cuerpo del loop, y antes eran ≥ 0 y $< n$ y $< m$, respectivamente. Como los índices están en rango, tiene sentido referirse a $g = G[i]$, y $b = B[j]$. Partamos en los tres casos que parte el algoritmo:

- Si $b < g - 1$, entonces b no es compatible con g , puesto que $|g - b| = g - b > 1$. Sabemos que G está ordenado crecientemente, luego para todo $g' \in G[i...n]$, $g' \geq g$, y luego $|g' - b| = g' - b \geq g - b > 1$, y por lo tanto b es también incompatible con todos los g' que quedan. Luego, sea S^* la extensión de S que existe por el invariante. Sabemos que S^* se descompone como $S^* = S \sqcup T$, con $S \subseteq G[0...i] \times B[0...j]$, y $T \subseteq G[i...n] \times B[j...m]$. Como $b = B[j]$, $b \notin B[0...j]$, y luego b no aparece emparejado en S . Vimos arriba que b no es compatible con nadie en $G[i...n]$, y luego b no puede estar en T , porque su pareja debería estar en $G[i...n]$. Luego, tenemos que

- $S \subseteq G[0...i] \times B[0...j + 1]$, trivialmente, porque ya estaba incluído en $G[0...i] \times B[0...j]$.
- $T \subseteq G[i...n] \times B[j + 1...m]$, porque sabíamos que estaba incluído en $G[i...n] \times B[j...m]$, y probamos que b no aparece emparejado con nadie en T , luego podemos restringir la segunda componente de T a $B[j + 1...m]$.
- $S^* = S \sqcup T$, que ya valía antes. Luego, como nuestro código dice:

```
j += 1
```

Estamos manteniendo el invariante, porque cambiamos de j a $j + 1$.

- Si $g < b - 1$, pasa algo exactamente análogo al caso anterior, y como decimos $i += 1$, mantenemos el invariante.
- Si no, tenemos que $g \geq b - 1$, y $b \geq g - 1$. Esto implica que $|g - b| \leq 1$, y por lo tanto son compatibles. Definimos $S' = S \cup \{(g, b)\}$, con $S' \subseteq G[0...i + 1] \times B[0...j + 1]$, porque $S \subseteq G[0...i] \times B[0...j]$, y S' usa $g = G[i]$ y $b = B[j]$. Queremos mostrar que S' es $(i + 1, j + 1)$ -extensible, para probar que se mantiene el invariante. Definamos S_G^* como los elementos de G que **no** menciona S^* , y S_B^* análogamente para B . Leér «single girls, single boys». Partimos en casos.
 - Si $g \in S_G^* \wedge b \in S_B^*$. Esto no puede suceder, porque (g, b) son compatibles, luego $S^* \cup \{(g, b)\}$ sería una solución con tamaño mayor a S^* , con S^* siendo definida como de tamaño máximo.
 - Si $g \in S_G^*$, pero $b \notin S_B^*$. Esto significa que existe un emparejamiento $(x, b) \in S^*$, con $x \neq g$, y que g no aparece emparejada en S^* . Como b no aparece emparejado en S , tiene que ser que $(x, b) \in T$. Consideremos entonces $T' = T \setminus \{(x, b)\}$. Definamos $S^{\{*\}'} = S' \sqcup T'$. Tenemos que $|S^{\{*\}'}| = |S'| + |T'| = |S| + 1 + |T| - 1 = |S| + |T| = |S^*|$, y luego $S^{\{*\}'}$ también es óptima, porque tiene el mismo tamaño que S^* . Notamos que agregar (g, b) es válido, porque g no aparecía emparejada en S^* , y rompimos la pareja de b cuando creamos T' . Notar también que $T' \subseteq G[i + 1...n] \times G[j + 1...m]$, porque g no aparece emparejado en S^* (a fortiori en T , y en T'), y le sacamos a T la mención de b . Por lo tanto, S' es $(i + 1, j + 1)$ -extensible.
 - Si $b \in S_B^*$, pero $g \notin S_G^*$. Esto es totalmente análogo al caso anterior.

4. Si $b \notin S_B^*, g \notin S_G^*$. Pueden pasar dos cosas, que estén emparejados entre sí, o que cada uno esté emparejado con alguien más.
 1. Si $(g, b) \in S^*$. Ya sabíamos que $(g, b) \notin S$, y luego como $S^* = S \sqcup T$, tenemos que $(g, b) \in T$. Definiendo $T' = T \setminus \{(g, b)\}$, vemos que $S^* = S' \sqcup T'$, y T' no usa ni $g = G[i]$ ni $b = B[j]$, y luego $T' \subseteq G[i+1\dots n] \times B[j+1\dots m]$. Luego S' es $(i+1, j+1)$ -extensible.
 2. Si no, entonces existen (g, x) y (y, b) en S^* . Como $g = G[i]$ y $b = B[j]$ no pueden estar en S porque $S \subseteq G[0\dots i] \times B[0\dots j]$, esas parejas tienen que estar en T . Notamos que $x \geq b$ y que $y \geq g$, porque x e y vienen después que b y g en B y G respectivamente, y B y G están ordenados de forma no-decreciente. Vamos a probar que x e y son compatibles.
 1. Si $g = b$, entonces $y \geq g = b$. Como $(y, g) \in T$, son compatibles, y luego $y = g$, o $y = g + 1$. Por el mismo motivo, $x \geq b = g$, y como $(g, x) \in T$, son compatibles, luego $x = g$, o $x = g + 1$. Luego, $\{x, y\} \subseteq \{g, g+1\}$, luego están a distancia a lo sumo 1 del otro, y luego son compatibles.
 2. Si $g > b$, entonces $g = b + 1$, porque g y b son compatibles. Tenemos $y \geq g > b$, y luego $y = b + 1$, porque (y, b) son compatibles, estando emparejados en T . Como x es compatible con $g = b + 1$, estando emparejados en T , tenemos que x es compatible con y .
 3. Si $b > g$, tenemos algo análogo al caso anterior.

En todos los casos, x e y son compatibles. Luego, podemos considerar $T' = T \cup \{(x, y)\} \setminus \{(g, x), (y, b)\}$, y definimos $S^{\{*\}'} = S' \sqcup T'$, con $|S^{\{*\}'}| = |S'| + |T'| = |S| + 1 + |T| + 1 - 2 = |S| + |T| = |S^*|$, y luego $S^{\{*\}'}$ también es una solución óptima. Como T sólo usa elementos que vienen *después* de $(g, b) = (G[i], B[j])$, tenemos que $T' \subseteq G[i+1\dots n] \times B[j+1\dots m]$, y luego S' es $(i+1, j+1)$ -extensible.

En todos los casos, tenemos que existe S' , un conjunto $(i+1, j+1)$ -extensible, con $|S'| = |S| + 1$. Recordemos que `res`, antes de entrar al cuerpo de la iteración, era igual a $|S|$, por el teorema del invariante. Luego, como nuestro código dice:

```
i += 1
j += 1
res += 1
```

`res` es ahora $|S'|$, y el invariante es preservado.

El ciclo termina

El cuerpo del ciclo siempre aumenta o i o j , empezando ambos en cero. Luego, si llegásemos a $n+m-1$ iteraciones, tendríamos $i+j = n+m-1$. Esto haría que la guarda no se cumpla y el ciclo termine - veamos por qué.

1. Si $i \geq n$, la guarda no se cumple.
2. Si $i < n$, y sabemos que $n+m-1 = i+j$, entonces $n+m+1 < n+j$, y restando n de cada lado obtenemos $m+1 < j$, o lo que es lo mismo, $j \geq m$, con lo cual la guarda no se cumple.

El invariante es suficiente para demostrar lo que queremos

Al final del ciclo, vale la negación de la guarda, y el invariante. La negación de la guarda es que o bien $i \geq n$, o bien $j \geq m$. Como vale el invariante, tenemos que $i \leq n$ y $j \leq m$. Luego, sabemos que o bien $i = n$, o bien $j = m$ (pueden pasar las dos juntas). Partimos en casos:

1. Si $i = n$, entonces S es (n, j) -extensible para algún j . Esto significa que existe una solución óptima S^* , y un conjunto $T \subseteq G[n \dots n] \times B[j \dots m]$, tal que $S^* = S \sqcup T$. Pero $G[n \dots n] = \emptyset$, y luego $T = \emptyset$, y luego $S = S^*$. Luego, S es una solución óptima.
2. Si $j = m$, pasa algo análogo con $B[m \dots m] = \emptyset$.

Luego, como sabemos que $\text{res} = |S|$, y devolvemos res :

```
return res
```

Nuestro algoritmo devuelve el tamaño de una solución óptima, y luego es correcto. □

Parte IV: Teoría de grafos

1 Definiciones básicas

TODO: This.

2 Árboles

Ejercicio 4.2.1

Un bosque es un grafo acíclico. Demostrar que cualquier bosque con n vértices y k árboles tiene $n - k$ aristas.



Demostración. Sea G un bosque, y sean $G_1 = (V_1, E_1), \dots, G_k = (V_k, E_k)$ sus k componentes conexas, con $1 \leq k \leq n$. Cada componente conexa es acíclica, pues G lo es, y es conexa.

Luego cada componente conexa es un árbol, y luego $|E_i| = |V_i| - 1$ para todo $1 \leq i \leq k$.

Como ninguna arista puede cruzar componentes conexas (pues las conectaría), cada arista en E está en exactamente un E_i para algún i , y tenemos que $E = \bigsqcup_{1 \leq i \leq k} E_i$. Cada vértice, mientras tanto, está en exactamente una componente conexa, y luego $V = \bigsqcup_{1 \leq i \leq k} V_i$.

Luego,

$$\begin{aligned}|E| &= \left| \bigsqcup_{1 \leq i \leq k} E_i \right| \\&= \sum_{i=1}^k |E_i| \\&= \sum_{i=1}^k |V_i| - 1 \\&= \left(\sum_{i=1}^k |V_i| \right) - k \\&= \left| \bigsqcup_{1 \leq i \leq k} V_i \right| - k \\&= |V| - k \\&= n - k\end{aligned}$$

que es lo que queríamos demostrar. □

Ejercicio 4.2.2

Probar que todo árbol con $n \geq 2$ vértices tiene al menos 2 hojas.



Demostración. Sea $G = (V, E)$ un árbol, con $n = |V| \geq 2$, y $m = |E|$. Luego, G es conexo y $m = n - 1$. Una hoja es un vértice de grado 1. Sea $d : V \rightarrow \mathbb{N}$ la función de grado de cada vértice. No puede haber vértices v con $d(v) = 0$, pues como $n > 1$, G no sería conexo. Luego, $d(v) \geq 1$ para todo $v \in V$. Sea $H \subseteq V$ el conjunto de hojas de T . Luego, $d(v) \geq 2$ para todo vértice en $V \setminus H$.

$$\begin{aligned}
\sum_{v \in V} d(v) &= \sum_{v \in H} d(v) + \sum_{v \in V \setminus H} d(v) \\
&= |H| + \sum_{v \in V \setminus H} d(v) \\
&\geq |H| + 2|V \setminus H| \\
&= |H| + 2(|V| - |H|) \\
&= 2|V| - |H|
\end{aligned}$$

También sabemos que $\sum_{v \in V} d(v) = 2|E| = 2m$, que vale para todo grafo. Como G es un árbol, $m = n - 1$, y luego $\sum_{v \in V} d(v) = 2m = 2(n - 1) = 2|V| - 2$.

Luego, $2|V| - 2 \geq 2|V| - |H|$, y luego $2 \leq |H|$, con lo cual G tiene al menos dos hojas. \square

Ejercicio 4.2.3

Un puente es una arista que, al ser removida, aumenta el número de componentes conexas en un grafo. Mostrar que en un bosque, todas las aristas son puentes.



Demostración. Una forma simple de demostrar esto es usando el Ejercicio 4.2.1, y viendo que al sacar una arista, el número de componentes conexas aumenta. Vamos a dar otra demostración.

Sea $G = (V, E)$ un bosque, y consideremos las componentes conexas G_1, \dots, G_k de G . Cada G_i es un árbol, para todo $1 \leq i \leq k$, por ser conexo y acíclico. Sea $e = \{u, v\}$ una arista en E . Esa arista pertenece a exactamente un G_i , pues de no ser así, e conectaría vértices de dos componentes conexas, que no puede suceder. Sea $G_i = (V_i, E_i)$ el árbol al que pertenece e .

Claramente el sacar e de G_i no va a cambiar nada sobre G_j con $j \neq i$. Al sacar una e de G_i , estamos desconectando u, v , que antes estaban conectados. Veamos que hay dos componentes conexas en $G'_i = G_i \setminus \{e\}$. Sea u un vértice en $G'_i = G_i \setminus \{e\}$.

1. Si hay un camino P entre w y u en G_i que no usa v , entonces P sigue estando en G'_i , y por tanto w pertenece en G'_i a la componente conexa de u .
2. Si no, el único camino P de w a u en G_i pasaba por v . Escribimos $P = [w, x_1, \dots, x_r, v, \dots, u]$. Entonces el camino $[w, x_1, \dots, x_r, v]$ existe en G'_i entre w y v (que no pasa por u). Luego w está en la componente conexa de v .

Si hubiera un camino en G'_i entre u y v , concatenar e a ese camino nos daría un ciclo en G_i , pero G_i era un árbol. Luego, u y v están en componentes conexas en G'_i , y todos los otros vértices de G'_i están en una de dos componentes conexas.

Luego G'_i tiene exactamente dos componentes conexas. Como antes de remover e de G teníamos una sola componente conexa en G_i , y al resto de los G_j con $j \neq i$ no le hacemos nada al remover e de G , entonces $G - \{e\}$ tiene exactamente una componente conexa más que G , y luego e es un puente. \square

Ejercicio 4.2.4

Sea $G = (V, E)$ un grafo conexo pesado, con todos sus pesos distintos. Sea $(U_1, U_2) = V$ una partición de los vértices de G . Sea e una arista de mínimo peso que cruza las particiones. Entonces e está en algún árbol generador mínimo de G .



Demostración. Sea $e = (u, v)$ tal arista. Llamemos $w : E \rightarrow \mathbb{R}$ a la función de peso de G , y por comodidad escribamos $w(H) = \sum_{v \in V(H)} w(v)$ dado un subgrafo H de G . Sea T un árbol generador mínimo de G .

Como T es árbol generador, existe en T un único camino P entre u y v . Como conecta u y v , con $u \in U_1$ y $v \in U_2$, en algún momento P cruza las particiones. Sea f una arista en P que cruza esas particiones. El enunciado nos dice que $w(e) \leq w(f)$.

Tomemos ahora $T' = T - f + e$. Al remover f , estamos desconectando las particiones U_1 y U_2 , pero como e cruza esas mismas particiones, agregar e las reconecta. Luego T' sigue siendo un árbol generador de G . Sin embargo, ahora tenemos que $w(T') = w(T) - w(f) + w(e) \leq w(T)$. Como T es un árbol generador mínimo, y T' un árbol generador, debemos tener $w(T) \leq w(T')$. Luego, $w(T) = w(T')$, y T' es un árbol generador mínimo que incluye a e . \square

Ejercicio 4.2.5

El algoritmo de Kruskal (resp. Prim) con orden de selección es una variante del algoritmo de Kruskal (resp. Prim) donde a cada arista e se le asigna una prioridad $q(e)$ además de su peso $p(e)$. Luego, si en alguna iteración del algoritmo de Kruskal (resp. Prim) hay más de una arista posible para ser agregada, entre esas opciones se elige alguna de mínima prioridad.

1. Demostrar que para todo árbol generador mínimo T de G , si las prioridades de selección están definidas por la función:

$$q_T(e) = \begin{cases} 0 & \text{si } e \in T \\ 1 & \text{si } e \notin T \end{cases}$$

entonces se obtiene T como resultado del algoritmo de Kruskal (resp. Prim) con orden de selección ejecutado sobre G (resp. cualquiera sea el vértice inicial en el caso de Prim).

2. Usando el inciso anterior, demostrar que si los pesos de G son todos distintos, entonces G tiene un único árbol generador mínimo.



Demostración.

1. Probemos primero que el algoritmo de Kruskal encuentra todos los árboles generadores mínimos, y luego lo mismo para el algoritmo de Prim. Llamemos $w(e) = (p(e), q_T(e))$, donde ordenamos las tuplas por orden lexicográfico. Es decir, $w(e) \leq w(e') \Leftrightarrow p(e) < p(e') \vee (p(e) = p(e') \wedge q_T(e) < q_T(e'))$.

Lema 4.2.6

El algoritmo de Kruskal con selección devuelve T .

Demostración. Por «Calamardo» en Telegram. Sea K el árbol generador que devuelve el algoritmo de Kruskal con selección. Asumimos que $K \neq T$. Luego, existe una primer arista e que el algoritmo agregó a K que no está en T . Consideremos $T' = T \cup \{e\}$. Este subgrafo de G contiene un único ciclo C , con $e \in C$. Como K es un árbol, $C \not\subseteq K$, puesto que un árbol no tiene ciclos. Luego existe alguna arista $f \in C$ tal que $f \notin K$. Como $f \notin K$ pero $e \in K$, tenemos que $f \neq e$. Como $f \neq e$, y $f \in C \subseteq T + e$, entonces $f \in T$. Consideremos entonces $T'' = T' \setminus \{f\} = (T \cup \{e\}) \setminus \{f\}$. Esto es un árbol generador, puesto que rompimos el único ciclo, C , que había en T' . Vemos que $p(T'') = p(T) + p(e) - p(f)$. Como T es un árbol generador mínimo, debemos tener $p(T'') \geq p(T)$, con lo cual $p(e) \geq p(f)$.

Como $e \notin T$, y $f \in T$, tenemos $q_T(f) = 0$, $q_T(e) = 1$. Luego, $w(f) < w(e)$, y el algoritmo de Kruskal con selección vio primero a f . Sea j el paso en el que el algoritmo vio (y decidió agregar) a e , e i el paso en el que el algoritmo vio a f . Tenemos que $j > i$. Dada una iteración t , sea F_t el conjunto de aristas que el algoritmo agregó al comenzar la iteración t . Tenemos que $F_t \subseteq F_{t+r}$ para todo $r \geq 1$. Como $j > i$, tenemos que $F_i \subseteq F_j$. El algoritmo de Kruskal con selección va a agregar a una arista x si y sólo si, al verla en un paso k , x no genera ciclos con F_k .

Como e es la primer arista que el algoritmo agrega que no está en T , entonces $F_j \subseteq T$ (y $F_{j+1} \not\subseteq T$). Como $f \in T$, y T es un árbol, f no genera ciclos con nadie en T , menos aún con aristas en F_j , y como $F_i \subseteq F_j$, menos aún va a generar ciclos con F_i . Luego, al ver a f en el paso i , el algoritmo pudo haber agregado a f , y no lo hizo, pero eso no puede pasar.

Luego, e no existe, y $K = T$. □

Lema 4.2.7

El algoritmo de Prim con selección devuelve T .

Demostración. Sea P el árbol generador que construye el algoritmo de Prim con selección. Recordemos que este algoritmo mantiene una partición (S_i, \overline{S}_i) de $V(G)$, y un conjunto de aristas F_i , donde F_i genera S_i (la componente conexa). Inicialmente $S_0 = \{v_0\}$ para algún v_0 arbitrario, y $F_0 = \emptyset$. En cada iteración, el algoritmo busca la arista de mínimo w que cruza la partición. Si esta arista es $\{u, v\}$, con $u \in S_i$ y $v \notin S_i$, tenemos $S_{i+1} = S_i \cup \{v\}$, y $F_{i+1} = F_i \cup \{e\}$. El algoritmo se detiene cuando $S_{n-1} = V(G)$, y devuelve F_{n-1} , donde $n = |V(G)|$.

Asumimos que $P \neq T$. Luego, existe una primer arista e que el algoritmo agregó a la componente conexa, que no está en T . Sean $\{u, v\} = e$ los extremos de e . Asumimos sin pérdida de generalidad que u es el extremo que está en S_i , y v el extremo que está en \overline{S}_i . Entonces, tenemos $S_{i+1} = S_i \cup \{v\}$, $F_{i+1} = F_i \cup \{e\}$.

Como T es un árbol generador, pero $e \notin T$, existe un (único) camino Q en T entre u y v . Como u y v están en diferentes lados de la partición (S_i, \overline{S}_i) , entonces en Q existe alguna

arista f que cruza la partición. Consideremos entonces $T' = (T \setminus \{f\}) \cup \{e\}$. Esto es un árbol generador, por haber separado a T en dos pedazos cuando sacamos f , uno que genera S y otro que genera \bar{S} , y luego haberlos juntado al agregar e . Como T es un árbol generador mínimo, tenemos que $p(T) \leq p(T') = p(T) - p(f) + p(e)$, con lo cual $p(e) \geq p(f)$.

Como f está en T , tenemos que $q_T(f) = 0$. Como e no está en T , tenemos que $q_T(e) = 1$. Como $p(f) \leq p(e)$, y $q_T(f) < q_T(e)$, tenemos que $w(f) < w(e)$. Como ambas aristas cruzan la partición (S_i, \bar{S}_i) , al considerar la arista e , el algoritmo también consideró la arista f . Pero entonces, el algoritmo no eligió a la arista que cruza la partición, de menor w . Esto no puede suceder.

Luego, e no existe, y $P = T$. □

2. Lo que nos dice el ejercicio anterior es que para todo árbol generador mínimo T , ambos el algoritmo de Kruskal y el algoritmo de Prim tienen ejecuciones que devuelven T , dependiendo sólo de cómo se desempata cuando encuentran aristas del mismo peso, en cada ejecución. Es decir, el conjunto de resultados posibles del algoritmo de Kruskal sobre G (resp. Prim), es igual al conjunto de árboles generadores mínimos de G

Luego, si para un grafo G en ningún momento hace falta desempatar, porque todos los pesos de las aristas de G son distintos, entonces en toda ejecución el resultado del algoritmo de Kruskal (resp. Prim) es único.

Como el conjunto de árboles generadores mínimos es igual al conjunto de resultados de correr estos algoritmos, y este último tiene un sólo elemento cuando los pesos de G son todos distintos, entonces G tiene un único árbol generador mínimo. □

3 Recorridos

TODO: This.

4 Caminos mínimos

Ejercicio 4.4.1

Tenemos un mapa con ciudades y rutas entre pares de ciudades. Para cada ruta e , tenemos un número $0 \leq p(e) < 1$ que nos indica la probabilidad de que nos caiga un rayo mientras estamos en esa ruta.

Diseñar un algoritmo que, dado un tal mapa y dos ciudades a y b , devuelva un camino entre a y b , que minimice la probabilidad de que nos caiga un rayo.



Demostración. Si tenemos dos eventos independientes, x e y , entonces $P(x \wedge y) = P(x)P(y)$. En general, si tenemos eventos disjuntos $\{x_i\}_{i \in I}$, la probabilidad de que pasen todos es $\prod_{i \in I} P(x_i)$.

Consideremos ahora cualquier camino $P = [e_1, e_2, \dots, e_k]$ desde a hasta b . La probabilidad de que no nos caiga un rayo en ningún momento es la probabilidad de que no nos caiga un rayo mientras estamos en e_1 , y que no nos caiga un rayo mientras estamos en e_2 , etc. Luego, esta

probabilidad es $X(P) = \prod_{e \in P} (1 - p(e))$. Luego, si queremos maximizar la probabilidad de que no nos caiga un rayo, como la función logaritmo es monótona creciente, podemos maximizar $\log X(P) = \sum_{e \in P} \log (1 - p(e))$. Esto es lo mismo que minimizar $-\log X(P) = -\sum_{e \in P} \log (1 - p(e)) = \sum_{e \in P} -\log (1 - p(e))$.

Luego, podemos tomar cada probabilidad $p(e)$, y transformarla en una función de peso, w , con $w(e) = -\log (1 - p(e))$. Como $0 \leq p(e) < 1$, entonces $1 \geq 1 - p(e) > 0$, y luego $0 \leq -\log (1 - p(e)) < \infty$. Entonces vemos que $w : \mathbb{E} \rightarrow \mathbb{R}_{\geq 0}$.

Luego, consideramos el grafo pesado $G = (V, E, w)$, donde los vértices V son las ciudades, las aristas E son las rutas entre ciudades, y w nos da el peso de cada camino. Como vimos, maximizar la probabilidad de que no nos caiga un rayo es lo mismo que encontrar un camino P desde a hasta b que minimice $\sum_{e \in P} -\log (1 - p(e))$. Si usamos w como pesos en las aristas, esto es encontrar un camino de suma de pesos mínima entre todos los caminos entre a y b . Para esto podemos usar el algoritmo de Dijkstra para caminos mínimos.

```
from math import log
def F(Ciudades: set[int],
      Rutas: dict[int, list[int]],
      p: Callable[[int, int], float],
      a: int,
      b: int):
    def w(i, j):
        return -log(1.0 - p(i, j))
    G = (Ciudades, Rutas, w)
    return Dijkstra(G, a)[b]
```

□

Ejercicio 4.4.2

Demostrar la correctitud del algoritmo de caminos mínimos de Dijkstra.

▲

Este va a ser un ejemplo de una demostración con menos formalismo que el típico para algoritmos con ciclos (es decir, el teorema del invariante), pero el mismo rigor. Tenemos que tener cuidado cuando hacemos esto, porque arriesgamos caer en «porque el vértice que no era el último sino el anteúltimo que en la anterior iteración cambiamos al valor que el otro vértice tenía antes de ser agregado cuando sabíamos el valor de d para todos los otros vértices», y demás oraciones incomprensibles.

Demostración. Recordemos el algoritmo de Dijkstra para caminos mínimos.

```
1: procedure DIJKSTRA( $G = (V, E)$ ,  $s \in V$ ,  $w : E \rightarrow \mathbb{R}_{\geq 0}$ )
2:    $Q \leftarrow V$ 
3:    $d[v] \leftarrow \infty \quad \forall v \in V \setminus \{s\}$ 
4:    $p[v] \leftarrow \text{NULL} \quad \forall v \in V$ 
5:    $d[s] \leftarrow 0$ 
6:   while  $Q \neq \emptyset$  do
7:      $u \leftarrow \arg \min_{u \in Q} d[u]$ 
```

```

8:    $Q \leftarrow Q \setminus \{u\}$ 
9:   for  $(u, v) \in E$  do
10:    if  $d[u] + w(u, v) < d[v]$  then
11:       $d[v] \leftarrow d[u] + w(u, v)$ 
12:       $p[v] \leftarrow u$ 
13:    end
14:  end
15: end
16: return  $d, p$ 
17: end

```

Vamos a demostrar y usar los dos invariantes del algoritmo.

Lema 4.4.3

En todo momento, para todo $v \in V$, si $d[v] < \infty$, entonces $d[v]$ es la longitud de algún camino desde s a v . En particular, $d[v] \geq \delta(s, v)$, con $\delta(x, y)$ la distancia en G entre los vértices x e y .

Además, si $d[v] < \infty$ y $v \neq s$, entonces $p[v]$ es un vértice inmediatamente anterior a v en algún camino desde s a v con longitud $d[v]$. Si $d[v] = \infty$ o $v = s$, entonces $p[v] = \text{NULL}$.

Demostración.

- Antes de comenzar el ciclo, $d[s] = 0$, y $d[v] = \infty$ para todo $v \in V \setminus \{s\}$. Como hay un camino de longitud 0 de s a s , y $\delta(s, s) = 0$ al ser todas las aristas de peso positivo, $d[s] = \delta(s, s)$. Para todos los otros vértices no hace falta probar nada sobre d (pues el antecedente en «si $d[v] < \infty$, entonces $d[v]$ es la longitud de algún camino entre s y v » es falso para ellos), probamos el caso base. Vemos también que $p[v] = \text{NULL}$ para todo $v \in V$, luego lo que dijimos sobre p es correcto.
- Ahora en cualquier iteración del ciclo, cuando asignamos $d[v] \leftarrow d[u] + w(u, v)$, por hipótesis inductiva $d[u]$ es la longitud de algún camino P desde s hasta u . Luego, como hay una arista $(u, v) \in E$, de peso $w(u, v)$, tenemos un camino $Q = P + [(u, v)]$ de costo $d[u] + w(u, v)$ desde s hasta v , y luego $d[v]$ es la longitud de algún camino entre s y v . Asimismo, este camino tiene a u como antecesor directo de v en este camino de longitud $d[v]$, y luego nuestra asignación de $p[v] \leftarrow u$ es correcta.

Como estas son las únicas dos formas de modificar d y p , probamos que esto se cumple en cualquier iteración del algoritmo. \square

Lema 4.4.4

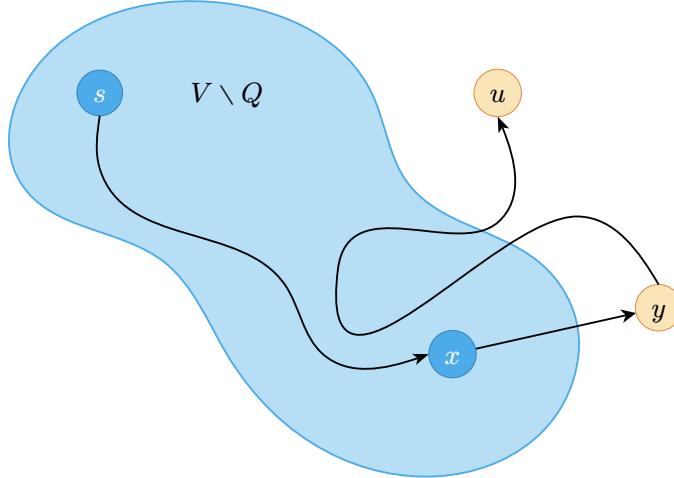
Cuando sacamos a u de Q , $d[u] = \delta(s, u)$.

Demostración. Vamos a probar esto por inducción en el número de iteraciones del ciclo. Sea $P(i)$: Al comenzar la i -ésima iteración del ciclo, tenemos $d[v] = \delta(s, v)$ para todo $v \in V \setminus Q$.

1. $P(0)$. Al comenzar el ciclo, no hay nada que probar, pues $Q = V$, entonces no hay nadie en $V - Q$.
2. $P(1)$. El único vértice que sacamos de Q en la primer iteración fue s , y luego al comenzar la segunda, $Q = V \setminus \{s\}$. En este caso, $d[s] = 0$, y no lo modificamos porque en la primer iteración no vemos aristas (s, s) (G es un grafo, no multigrafo). Como $\delta(s, s) = 0 = d[s]$, vale $P(1)$.
3. Paso inductivo. Sabemos $P(k)$ para todo $k \leq i$, queremos ver que vale $P(i + 1)$. Sea u el vértice que estamos sacando en la i -ésima iteración. Este es el vértice que, al sacar de Q en la i -ésima iteración, estamos «agregando» a $V \setminus Q$. Luego, como para todos los otros vértices v en $V \setminus Q$ sabemos que $d[v] = \delta(s, v)$ por el Lema 4.4.3, y no van a cambiar más porque en cada iteración sólo pueden decrecer, lo único que nos queda probar es que para u , también tenemos que $d[u] = \delta(s, u)$.

Si $d(s, u) = \infty$, es decir no hay ningún camino entre s y u , terminamos, pues por el Lema 4.4.3, sabemos que $d[u] = \infty$.

De otra forma, consideremos un camino P de longitud mínima desde s hasta u . Sea y el primer vértice en P que está en Q , y sea x su predecesor en P (podríamos tener $y = u$, o $x = s$).



Como y aparece no-después que u en P , y P es un camino mínimo con todas las aristas de peso no-negativo, tenemos que $\delta(s, y) \leq \delta(s, u)$. Como $u = \arg \min_{u \in Q} d[u]$, y sabemos que $y \in Q$, entonces $d[u] \leq d[y]$. Por el Lema 4.4.3, sabemos que $\delta(s, u) \leq d[u]$.

Como $x \in V \setminus Q$, fue removido de Q en alguna iteración $j \leq i$. Podemos usar $P(j)$, entonces, para ver que $d[x] = \delta(s, x)$ al removerlo. Durante la iteración j , nos aseguramos que $d[y] \leq d[x] + w(x, y) = \delta(s, x) + w(x, y)$. Como $s \rightsquigarrow x \rightarrow y$ es parte de un camino mínimo P , entonces $s \rightsquigarrow x \rightarrow y$ es también un camino mínimo, y entonces $\delta(s, x) + w(x, y) = \delta(s, y)$. Como $d[y] \leq \delta(s, y)$, y por el Lema 4.4.3

sabemos que $d[y] \geq \delta(s, y)$, tenemos que $d[y] = \delta(s, y)$ al final de la iteración j , y luego por el Lema 4.4.3, sigue valiendo en la iteración $i + 1$.

Luego sabemos que $\delta(s, y) \leq \delta(s, u) \leq d[u] \leq d[y]$, y $d[y] = \delta(s, y)$. Luego, $\delta(s, y) = \delta(s, u) = d[u] = d[y]$. Luego $d[u] = \delta(s, u)$ al terminar la iteración i , es decir, vale también al comenzar la iteración $i + 1$, y luego vale $P(i + 1)$.

□

Como el algoritmo termina cuando $Q = \emptyset$, y empieza con $Q = V$, habremos al final sacado de Q todos los vértices $v \in V$. Por el Lema 4.4.4, Al momento de sacar cada $v \in V$ de Q , tendremos $d[v] = \delta(s, v)$, y luego $d[v]$ es la longitud de un camino mínimo entre s y v . Por el Lema 4.4.3, como cada vez que actualizamos $d[v]$ sólo lo hacemos decrecer, y siempre es la longitud de un camino entre s y v , nunca puede ser actualizado a algo menor a $\delta(s, v)$. Vemos entonces que luego de sacar a v de Q , $d[v]$ nunca más se actualiza, y permanece en $\delta(s, v)$.

Luego, al terminar el algoritmo, tenemos $d[v]$ para todo $v \in V$, y es igual a $\delta(s, v)$. Más aún, para todo $v \in V$, $p[v]$ es o bien `NULL` (cuando $s = v$, o cuando $d[v] = \infty$), o es el antecesor directo de v en algún camino mínimo desde s hasta v .

□

Ejercicio 4.4.5

Sea $G = (V, E)$ un grafo dirigido pesado con pesos positivos, y $s, t \in V$. Queremos encontrar el camino de menor peso entre s y t . De todos los caminos con menor peso, queremos el que menos aristas tenga.

Dar un algoritmo que devuelva un tal camino. Demostrar que es correcto.

♣

Demostración. Para entender cómo resolver este ejercicio es crucial ver la demostración de por qué funciona el algoritmo de Dijkstra, que damos en el Ejercicio 4.4.2.

El invariante que mantiene el algoritmo de Dijkstra es que para todos los vértices que sacamos de la cola Q , sabemos exactamente su distancia desde s , el vértice inicial. Mantenemos este invariante removiendo de Q el vértice u con menor distancia estimada $d[u]$, y actualizando el estimado de las distancias para todos vecinos v de u ($d[v] \leftarrow \min(d[v], d[u] + w(u, v))$).

Si queremos comparar dos cosas, en orden, la estructura típica para esto son los pares, con su orden lexicográfico. En este caso, si queremos comparar primero por distancia (suma de pesos), y habiendo empates, por número de aristas, podemos definir:

- $\delta(s, v)$ como la suma de los pesos en un camino de mínima distancia desde s hasta v ,
- $\kappa(s, v)$ como el mínimo número de aristas entre todos los caminos de mínima distancia entre s y v ,
- $\varepsilon(s, v) = (\delta(s, v), \kappa(s, v))$

Vemos que ε se comporta de manera similar a δ . Si tenemos un camino de mínima distancia y mínimo número de aristas entre todos los caminos mínimos $P = [s, \dots, x, y]$, entre s y y , entonces no solo $\delta(s, y) = \delta(s, x) + w(x, y)$ (que usamos en la demostración del algoritmo de

Dijkstra), sino que $\varepsilon(s, y) = \varepsilon(s, x) + (w(x, y), 1)$, donde definimos la suma componente-a-componente.

Veamos si podemos adaptar el algoritmo original a esta nueva noción de «distancia».

Nota

Como estamos cambiando el algoritmo, tenemos que demostrar que este nuevo algoritmo es correcto con respecto a la nueva especificación que estamos pidiendo. No podemos usar la demostración de un algoritmo distinto, y sólo decir que «sigue valiendo», primero porque no lo demostramos (y en general es muy difícil decir que «siguen valiendo» absolutamente todas las deducciones que se hicieron en otra demostración), y segundo porque el anterior algoritmo cumple un objetivo *distinto* al nuevo.

Tenemos que ir oración por oración de la demostración anterior, ver qué sigue valiendo, y qué no y hay que cambiar y probar. En particular, vamos a necesitar un nuevo lema acá, el Lema 4.4.7.

```

1: procedure DIJKSTRA( $G = (V, E)$ ,  $s \in V$ ,  $w : E \rightarrow \mathbb{R}_{\geq 0}$ )
2:    $Q \leftarrow V$ 
3:    $d[v] \leftarrow (\infty, \infty) \quad \forall v \in V \setminus \{s\}$ 
4:    $p[v] \leftarrow \text{NULL} \quad \forall v \in V$ 
5:    $d[s] \leftarrow (0, 0)$ 
6:   while  $Q \neq \emptyset$  do
7:      $u \leftarrow \arg \min_{u \in Q} d[u]$ 
8:      $Q \leftarrow Q \setminus \{u\}$ 
9:     for  $(u, v) \in E$  do
10:       if  $d[u] + (w(u, v), 1) < d[v]$  then
11:          $d[v] \leftarrow d[u] + (w(u, v), 1)$ 
12:          $p[v] \leftarrow u$ 
13:       end
14:     end
15:   end
16:   return  $d, p$ 
17: end
```

Definición 4.4.6

Llamamos a un camino $[s, \dots, u]$ **ε -óptimo** cuando tiene longitud mínima entre todos los caminos desde s hasta u , y dentro de todos esos caminos de longitud mínima, tiene el mínimo número de aristas.

Lema 4.4.7 (Subestructura óptima de caminos ε -óptimos)

Sea $P = [s, \dots, x, y]$ un camino ε -óptimo, y sea $Q = [s, \dots, x]$ su prefijo. Entonces Q es ε -óptimo.

En particular, $\varepsilon(s, x) \leq \varepsilon(s, y)$.

Demostración. Vamos a probar esto por inducción. Por conveniencia notamos $w(X) = \sum_{e \in X} w(e)$, para cualquier camino X .

Como P es ε -óptimo, entonces tiene mínima distancia hasta y , y luego $w(P) = \delta(s, y)$. Asimismo, vemos que $w(P) = w(Q) + w(x, y)$. Como Q es un camino desde s hasta x , y $\delta(s, x)$ es la mínima distancia desde s hasta x , entonces $w(Q) \geq \delta(s, x)$.

Si $w(Q) > \delta(s, x)$, podemos tomar cualquier camino mínimo Q' entre s y x , con $w(Q') = \delta(s, x)$. Entonces, $Q' + (x, y)$ es un camino entre s e y , con distancia $w(Q') + w(x, y) < w(Q) + w(x, y) = w(P)$, pero P era de mínima distancia entre s e y , entonces Q' no puede existir. Luego, $w(Q) = \delta(s, x)$.

Como Q es un camino de mínima longitud desde s hasta x , y $\kappa(s, x)$ es el mínimo número de aristas entre todos los caminos mínimos desde s hasta x , entonces $|Q| \geq \kappa(s, x)$, con $|Q|$ el número de aristas de Q . Si $|Q| > \kappa(s, x)$, entonces sea Q' cualquier camino de mínima longitud entre s y x , y de entre ellos, uno de mínimo número de aristas (es decir, $\kappa(s, x)$ aristas). Tenemos $w(Q') = w(Q)$, pues ambos son caminos mínimos entre s y x . Luego $Q' + (x, y)$ es un camino de longitud $w(Q') + w(x, y) = w(Q) + w(x, y) = w(P)$, y número de aristas $|Q'| + 1 = \kappa(s, x) + 1 < |Q| + 1 = |P|$. Pero P era un camino ε -óptimo, entonces esto no puede pasar. Luego, Q' no puede existir, y tenemos $|Q| = \kappa(s, x)$. Luego, como $w(Q) = \delta(s, x)$, y $|Q| = \kappa(s, x)$, tenemos que Q es ε -óptimo. \square

Lema 4.4.8

En todo momento, para todo $v \in V$, si $d[v] = (a, b) < (\infty, \infty)$, entonces existe un camino entre s y v , con longitud a , y b aristas. En particular, $d[v] \geq \varepsilon(s, v)$.

Además, si $d[v] \neq (\infty, \infty)$ y $v \neq s$, entonces $p[v]$ es un vértice inmediatamente anterior a v en algún camino desde s a y con longitud $d[v][0]$ y número de aristas $d[v][1]$. Si $d[v] = (\infty, \infty)$ o $v = s$, entonces $p[v] = \text{NULL}$.



Demostración.

- Antes de comenzar el ciclo, $d[s] = (0, 0)$, y $d[v] = (\infty, \infty)$ para todo $v \in V \setminus \{s\}$. Como hay un camino de longitud 0 de s a s , y $\delta(s, s) = 0$ al ser todas las aristas de peso positivo, $d[s] = (\delta(s, s), \kappa(s, s)) = \varepsilon(s, s)$. Para todos los otros vértices no hace falta probar nada sobre d (pues el antecedente en «si $d[v] < (\infty, \infty)$, entonces ...» es falso para ellos). Vemos también que $p[v] = \text{NULL}$ para todo $v \in V$, luego lo que dijimos sobre p es correcto.
- Ahora en cualquier iteración del ciclo, cuando cambiamos $d[v] = d[u] + (w(u, v), 1)$, por hipótesis inductiva, notando $d[u] \leftarrow (a, b)$, a es la longitud de algún camino P desde s hasta u , y b es el número de aristas de P . Luego, como hay una arista $(u, v) \in E$, de peso $w(u, v)$, tenemos un camino $Q = P + [(u, v)]$ de costo $d[u] + w(u, v)$ desde s hasta v , con $b + 1$ aristas. Luego, existe un camino Q tal que $d[v] = (a', b')$,

con a' la longitud de Q , y b' el número de aristas de Q . En P , el antecesor directo de v es u , y luego nuestra asignación $p[v] \leftarrow u$ es correcta.

Como estas son las únicas dos formas de modificar d y p , probamos que esto se cumple en cualquier iteración del algoritmo. \square

Lema 4.4.9

Cuando sacamos a u de Q , $d[u] = \varepsilon(s, u)$.



Demostración. Vamos a probar esto por inducción en el número de iteraciones del ciclo. Sea $P(i)$: Al comenzar la i -ésima iteración del ciclo, tenemos $d[v] = \varepsilon(s, v)$ para todo $v \in V \setminus Q$.

1. $P(0)$. Al comenzar el ciclo, no hay nada que probar, pues $Q = V$, entonces no hay nadie en $V - Q$.
2. $P(1)$. El único vértice que sacamos de Q en la primer iteración fue s , y luego al comenzar la segunda, $Q = V \setminus \{s\}$. En este caso, $d[s] = (0, 0)$, y no lo modificamos porque en la primer iteración no vemos aristas (s, s) (G es un grafo, no multigrafo). Como $\delta(s, s) = 0$, y $\kappa(s, s) = 0$, vale $P(1)$.
3. Paso inductivo. Sabemos $P(k)$ para todo $k \leq i$, queremos ver que vale $P(i + 1)$. Sea u el vértice que estamos sacando en la i -ésima iteración. Este es el vértice que, al sacar de Q en la i -ésima iteración, estamos «agregando» a $V \setminus Q$. Luego, como para todos los otros vértices v en $V \setminus Q$ sabemos que $d[v] = \varepsilon(s, v)$ por el Lema 4.4.8, y no van a cambiar más porque en cada iteración sólo pueden decrecer, lo único que nos queda probar es que para u , también tenemos que $d[u] = \varepsilon(s, u)$.

Si $\delta(s, u) = \infty$, es decir no hay ningún camino entre s y u , usamos el contrarrecíproco del Lema 4.4.8, y al no haber ningún camino, entonces $d[u] = (\infty, \infty)$.

De otra forma, consideraremos un camino ε -óptimo P desde s hasta u . Sea y el primer vértice en P que está en Q , y sea x su predecesor en P (podríamos tener $y = u$, o $x = s$).

Como P es ε -óptimo, por el Lema 4.4.7, $\varepsilon(s, y) \leq \varepsilon(s, u)$. Como $u = \arg \min_{u \in Q} d[u]$, y sabemos que $y \in Q$, entonces $d[u] \leq d[y]$. Por el Lema 4.4.8, sabemos que $\varepsilon(s, u) \leq d[u]$.

Como $x \in V \setminus Q$, fue removido de Q en alguna iteración $j \leq i$. Podemos usar $P(j)$, entonces, para ver que $d[x] = \varepsilon(s, x)$ al removerlo. Durante la iteración j , nos aseguramos que $d[y] \leq d[x] + (w(x, y), 1) = \varepsilon(s, x) + (w(x, y), 1)$. Por Lema 4.4.7, como $s \rightsquigarrow x \rightarrow y$ es parte de un camino ε -óptimo P , entonces $s \rightsquigarrow x \rightarrow y$ es también ε -óptimo, y entonces $\varepsilon(s, x) + (w(x, y), 1) = \varepsilon(s, y)$. Como $d[y] \leq \varepsilon(s, y)$, y por el Lema 4.4.8 sabemos que $d[y] \geq \varepsilon(s, y)$, tenemos que $d[y] = \varepsilon(s, y)$ al final de la iteración j , y luego por el Lema 4.4.8, sigue valiendo en la iteración $i + 1$.

Luego sabemos que $\varepsilon(s, y) \leq \varepsilon(s, u) \leq d[u] \leq d[y]$, y $d[y] = \varepsilon(s, y)$. Luego, $\varepsilon(s, y) = \varepsilon(s, u) = d[u] = d[y]$. Luego $d[u] = \varepsilon(s, u)$ al terminar la iteración i , es decir, vale también al comenzar la iteración $i + 1$, y luego vale $P(i + 1)$.

□

Como el algoritmo termina cuando $Q = \emptyset$, y empieza con $Q = V$, habremos al final sacado de Q todos los vértices $v \in V$. Por el Lema 4.4.9, Al momento de sacar cada $v \in V$ de Q , tendremos $d[v] = \varepsilon(s, v)$, y luego $d[v]$ es la longitud de un camino mínimo entre s y v . Por el Lema 4.4.8, como cada vez que actualizamos $d[v]$ sólo lo hacemos decrecer, y siempre es un par con la longitud de un camino entre s y v y su número de aristas, nunca puede ser actualizado a algo menor a $\varepsilon(s, v)$. Vemos entonces que luego de sacar a v de Q , $d[v]$ nunca más se actualiza, y permanece en $\varepsilon(s, v)$.

Luego, al terminar el algoritmo, tenemos $d[v]$ para todo $v \in V$, y es igual a $\varepsilon(s, v)$. Más aún, para cada $v \in V$, $p[v]$ es o NULL cuando $v = s$ o $d[v] = (\infty, \infty)$, o el antecesor inmediato de v en algún camino ε -óptimo desde s hasta v .

□

Ejercicio 4.4.10

Sea $G = (V, E)$ un grafo dirigido pesado por $w : E \rightarrow \mathbb{R}_{>0}$, y $s \in V$.

Dar un algoritmo que calcule el número de caminos de mínimo peso entre s y v , para cada $v \in V$.

▲

*Demuestra*ción. La idea de este algoritmo es primero encontrar el grafo dirigido acíclico G' de caminos mínimos desde s . Luego, queremos encontrar el número de caminos en G' , desde s hasta v , para cada v . Para la segunda parte, podemos usar un simple algoritmo de programación dinámica. Para la primer parte, tenemos dos opciones:

- Podemos modificar el algoritmo de Dijkstra. Normalmente, el algoritmo devuelve un array p , donde $p[v]$ nos da un vértice inmediatamente anterior a v en un camino mínimo desde s hasta v . Podemos modificar esto para que $p[v]$ guarde un conjunto de *todos* los vértices que están inmediatamente antes que v , en algún camino mínimo desde s hasta v . La modificación sería que al iterar cada arista $(u, v) \in E$, incidente al vértice u que sacamos de Q , hacemos:

```

1: if  $d[v] < d[u] + w(u, v)$  then
2:    $d[v] \leftarrow d[u] + w(u, v)$ 
3:    $p[v] \leftarrow \{u\}$ 
4: else if  $d[v] = d[u] + w(u, v)$  then
5:    $p[v] \leftarrow p[v] \cup \{u\}$ 
6: end

```

Esta es una modificación útil, pero como vimos en el ejercicio anterior, tenemos que volver a demostrar que el algoritmo de Dijkstra con esta modificación es correcto con respecto a su

nueva especificación, que va a hablar sobre *todos* los posibles antecesores inmediatos de cada vértice v .

Otra modificación que podríamos hacer es una que sólo cuente cuántos caminos hay:

```

1: if  $d[v] < d[u] + w(u, v)$  then
2:    $d[v] \leftarrow d[u] + w(u, v)$ 
3:    $p[v] \leftarrow p[u]$ 
4: else if  $d[v] = d[u] + w(u, v)$  then
5:    $p[v] \leftarrow p[v] + p[u]$ 
6: end

```

donde $p[s] = 1$ es el valor inicial, y para todo otro vértice v , $p[v] = 0$ inicialmente.

Esto requiere una demostración aún más extensa. Tendremos que argumentar que el conjunto de caminos desde s hasta v , se partitiona de forma disjunta en los caminos desde s hasta u , y luego la arista (u, v) , para todos los vértices u tal que $\delta(s, v) = \delta(s, u) + w(u, v)$.

- Podemos usar el algoritmo de Dijkstra canónico. Luego, podemos reconstruir G' , usando sólo d , el array de distancias que devuelve el algoritmo de Dijkstra. Hay que escribir más código para esto, pero el código cuya correctitud debemos demostrar es simple y corto.

Como usamos el primer método en el ejercicio anterior, en este vamos a usar el segundo.

El algoritmo que vamos a usar es el siguiente:

```

from collections import defaultdict
def F(G, w, s):
    d, _ = Dijkstra(G, w, s)
    (V, E) = G
    preds = defaultdict(list)
    for (u, v) in E:
        if d[v] == d[u] + w(u, v):
            preds[v].append(u)

    n = len(V)
    x = [-1 for _ in range(n)]
    x[s] = 1
    def rec(v):
        if x[v] == -1:
            x[v] = sum(rec(u) for u in preds[v])
        return x[v]

    for v in range(n): rec(v)
    return x

```

Demostremos que este algoritmo es correcto. La función `rec` es meramente usar programación dinámica top-down, en la siguiente función recursiva:

$$f(v) = \begin{cases} 1 & \text{si } v = s \\ \sum_{\substack{(u,v) \in E, \\ d[v]=d[u]+w(u,v)}} f(u) & \text{si no} \end{cases}$$

Y al correr `f(v)` para todo `v in range(n)`, estamos asegurándonos de llenar el cache `x` para todos los vértices, con lo cual devolvemos un array cuya i -ésima coordenada contiene $f(i)$. La

semántica que le asignamos a la función f es que $f(i)$ **devuelve el número de caminos mínimos desde s hasta i** .

Advertencia

Recordemos que no tiene sentido probar que una función «es correcta» sin decir cuál es su semántica. Una función que siempre devuelve el string "banana" es correcta si nuestra semántica es «una función que siempre devuelve el string "banana"».

Siempre que vamos a probar que una función es correcta, **tenemos** que definir cuál es su semántica. Por esto ven especificación antes de correctitud formal de algoritmos.



Probemos que f es correcta con respecto a esa semántica. Como es una función recursiva, vamos a usar inducción. ¿En qué vamos a hacer inducción? En lo que está decreciendo en cada llamada: El máximo número de aristas en un camino mínimo entre s y v . Por comodidad, llamemos $C(v)$ al conjunto de caminos mínimos entre s hasta v , y llamemos $M(v) = \max_{P \in C(v)} |P|$, el máximo número de aristas en cualquier camino mínimo desde s hasta v .

Sea $P(i)$: Para todo $v \in V$, tal que $M(v) = i$, $f(v) = |C(v)|$.

1. Caso base, $i = 0$. Si v es un vértice tal que $M(v) = 0$, entonces hay un camino de longitud 0 desde s hasta v . Luego, v es s . Como hay un único camino desde s hasta s , vemos que $|C(s)| = |\{\}\} = 1$. Como f precisamente devuelve 1, f es correcta para este caso.
2. Paso inductivo. Sea $i > 0 \in \mathbb{N}$. Podemos asumir $P(k)$ para todo $k < i$, y queremos ver $P(i)$. Sea $v \in V$ tal que $M(v) = i$. Como $M(v) = i > 0$, sabemos que $v \neq s$, pues $M(s) = 0$. Luego, todo camino mínimo Q desde s hasta v , pasa por algún vértice u anterior a v . Notar que u puede ser s mismo, si hay un camino mínimo desde s hasta v que es puramente $[(s, v)]$. También, como los caminos mínimos tienen subestructura óptima, el prefijo de Q tiene que ser un camino mínimo desde s hasta ese vértice u . Notemos que tendremos $\delta(s, u) + w(u, v) = \delta(s, v)$, por definición de δ y que Q y su prefijo hasta u son caminos mínimos. Finalmente, por la postcondición del algoritmo de Dijkstra, sabemos que $d[v] = \delta(s, v)$ para todo $v \in V$.

$$\begin{aligned} C(v) &= \{Q \mid Q \in C(v)\} \\ &= \{Q' + [(u, v)] \mid u \in V, Q' \in C(u), Q' + [(u, v)] \in C(v)\} \\ &= \{Q' + [(u, v)] \mid u \in V, Q' \in C(u), \delta(s, u) + w(u, v) = \delta(s, v)\} \\ &= \{Q' + [(u, v)] \mid u \in V, Q' \in C(u), d[u] + w(u, v) = d[v]\} \end{aligned}$$

Luego,

$$\begin{aligned}
|C(v)| &= |\{Q' + [(u, v)] \mid u \in V, Q' \in C(u), d[u] + w(u, v) = d[v]\}| \\
&= \sum_{\substack{u \in V \\ Q' \in C(u) \\ d[u] + w(u, v) = d[v]}} 1 \\
&= \sum_{\substack{u \in V \\ d[u] + w(u, v) = d[v]}} |C(u)|
\end{aligned}$$

Si probamos que $M(u) < M(v)$ para todo u en esa sumatoria, probamos que f es correcta, porque podemos usar la hipótesis inductiva $P(M(u))$ para saber que $f(u) = |C(u)|$, y llegar a

$$|C(v)| = \sum_{\substack{u \in V \\ d[u] + w(u, v) = d[v]}} f(u)$$

que probaría $P(i)$, pues el lado derecho es la definición exacta de f , y estaríamos probando $|C(v)| = f(v)$. Sea $u \in V$ tal que $d[u] + w(u, v) = d[v]$, y tomemos un camino Q de máximo número de aristas en $C(u)$. Por cómo definimos $M(u)$, tenemos que $M(u) = |Q|$. Vemos que $Q' = Q + [(u, v)]$ es un camino en $C(v)$, y que $|Q'| = |Q| + 1$. Como $M(v) = \max_{R \in C(v)} \{|R|\}$, y Q' es un tal R , tenemos que $M(v) \geq |Q'| = |Q| + 1 > M(u)$. Luego $M(u) < M(v)$, y entonces podemos usar la hipótesis inductiva, $P(M(u))$, que nos deja concluir que f es correcta. \square

Ejercicio 4.4.11

Sea $G = (V, E)$ un grafo dirigido pesado, y $s \in V$.

Queremos encontrar el camino más ligero entre s y cada $v \in V$. Dar una modificación del algoritmo de Bellman-Ford tal que:

- Si G no tiene ciclos de peso negativo alcanzables desde s , devuelve la distancia entre s y v , para todo $v \in V$.
- Si G tiene un ciclo de peso negativo alcanzable desde s , devuelve un tal ciclo.

Demostrar que es correcto. El algoritmo debería seguir haciendo $O(|V| |E|)$ operaciones en el peor caso.



Demostración. Recordemos el invariante del algoritmo de Bellman-Ford para caminos mínimos en grafos pesados.

```

1: procedure BELLMAN-FORD( $G = (V, E)$ ,  $s \in V$ ,  $w : E \rightarrow \mathbb{R}$ )
2:    $d[v] \leftarrow \infty \forall v \in V$ 
3:    $d[s] \leftarrow 0$ 
4:    $p[v] \leftarrow \text{NULL} \forall v \in V$ 
5:   for  $i = 1$  to  $|V| - 1$  do
6:     for  $(u, v) \in E$  do
7:       if  $d[v] > d[u] + w(u, v)$  then
8:          $d[v] \leftarrow d[u] + w(u, v)$ 
9:          $p[v] \leftarrow u$ 
10:      end
11:    end

```

```

12:   end
13:   return  $d, p$ 
14: end

```

El invariante de este algoritmo es que al terminar la i -ésima iteración del ciclo externo, para todo $v \in V$, $d[v]$ es la distancia mínima desde s hasta v usando sólo caminos de a lo sumo i aristas. Luego, al terminar el algoritmo, tenemos que $d[v]$ es la distancia mínima desde s hasta v , usando a lo sumo $|V| - 1$ aristas, para todo $v \in V$.

Si no hay ciclos de peso negativo alcanzables desde s , entonces la distancia «usando a lo sumo $|V| - 1$ aristas» es lo mismo que la distancia, dado que si nuestro camino tiene más de $|V| - 1$ aristas, tiene algún ciclo, que si fuera de peso no-negativo podríamos simplemente remover, y no-empeorar la distancia.

Si hay ciclos de peso negativo alcanzables desde s , entonces las distancias hacia algunos vértices no están bien definidas.

Lema 4.4.12

Hay un ciclo negativo alcanzable desde s si y sólo si al hacer una iteración más del ciclo externo, en algún momento cambiamos d .

Demostración.

- \Leftarrow) Sea $n = |V|$. Por el invariante del algoritmo, al terminar la $n - 1$ -ésima iteración, tenemos en $d[v]$ las distancias desde s hasta v , usando a lo sumo $n - 1$ aristas. Si hacemos una iteración más, y cambiamos $d[v]$, entonces ocurrió que $d[v] > d[u] + w(u, v)$ para algún u tal que $(u, v) \in E$. Por el invariante del algoritmo, sabemos que $d[v]$ es la distancia mínima desde s hasta u , usando a lo sumo $n - 1$ aristas. Luego, si $d[v] > d[u] + w(u, v)$, entonces el mejor camino desde s hasta v usando $n - 1$ aristas es más pesado que un camino que va hasta u usando a lo sumo $n - 1$ aristas, y luego usa la arista (u, v) . Este camino necesariamente debe tener más de $n - 1$ aristas, pues por inducción $d[v]$ ya tiene la menor distancia entre todos los caminos que usan a lo sumo $n - 1$ aristas.

Un camino P de más de $n - 1$ aristas debe tener algún ciclo C . Si ese ciclo fuese de peso no-negativo, podríamos removerlo, y obtener $P - C$, que sigue empezando en s , terminando en v , y tiene peso igual a P , es decir $d[u] + w(u, v)$. Pero entonces $P - C$ es un camino de a lo sumo $n - 1$ aristas, y tiene peso $d[u] + w(u, v) < d[v]$, lo cual no puede pasar porque $d[v]$ era la mínima distancia entre s y v usando caminos de a lo sumo $n - 1$ aristas.

Luego P debe tener algún ciclo de peso negativo, y como P empieza en s y termina en v , entonces v es alcanzable desde s por un ciclo C de peso negativo.

- \Rightarrow) Sea $C = [v_0, \dots, v_k = v_0]$ un ciclo de peso negativo alcanzable desde s . Luego, $\sum_{i=1}^k w(v_{i-1}, v_i) < 0$. Supongamos que no actualizamos ningún d en la n -ésima iteración. Entonces, en esa iteración tenemos que $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ para todo $1 \leq i \leq k$. Sumando todas estas desigualdades, tenemos

$$\sum_{i=1}^k d[v_i] \leq \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i)$$

Vemos también que como $v_0 = v_k$, la suma $\sum_{i=1}^k d[v_i]$ es igual a la suma $\sum_{i=1}^k d[v_{i-1}]$, y en ambas aparecen todos los vértices de C una sola vez. Luego, restando esta suma de cada lado, obtenemos

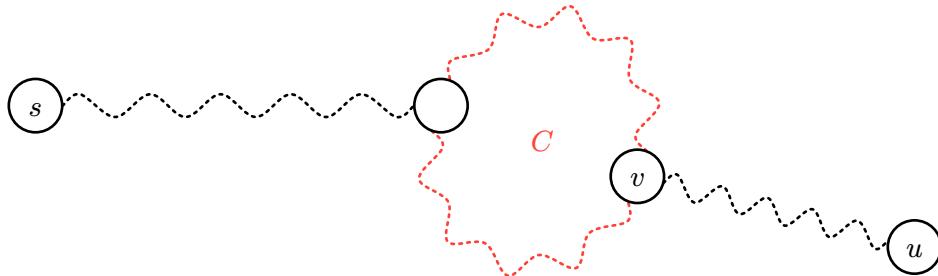
$$0 \leq \sum_{i=1}^k w(v_{i-1}, v_i)$$

Pero esto no puede pasar, pues sabemos que C tiene peso negativo. Luego, en la n -ésima iteración, tuvimos que haber actualizado $d[v]$ para algún $v \in C$.

□

Esto nos dice que para saber si hay algún ciclo de peso negativo alcanzable desde s , podemos hacer una iteración más del ciclo externo, y si en algún momento cambiamos $d[v]$, entonces v es alcanzable desde s por un ciclo de peso negativo. Si no cambiamos nada, entonces no hay ciclos de peso negativo alcanzables desde s . Más aún, si ese ciclo de peso negativo C existe, entonces debemos actualizar $d[v]$ en la n -ésima iteración para algún $v \in C$.

Notemos que no necesariamente vamos a cambiar *sólo* los $d[v]$ donde $v \in C$. Como recorremos $e \in E$ en algún orden, puede ser que primero actualizamos $d[v]$ para algún $v \in C$, y luego actualizamos $d[u]$ para algún u alcanzable desde v . Lo que vamos a saber es que si actualizamos $d[u]$, entonces u es alcanzable desde algún ciclo negativo, que a su vez es alcanzable desde s .



Luego, si actualizamos $d[u]$ en la n -ésima iteración, podemos seguir la cadena de padres, p , y vamos a llegar hasta algún vértice (en este caso v) que pertenece a C .

Cuando actualizamos $d[v]$ en la última iteración, como el mejor camino desde s hasta v ahora tiene un ciclo (pues tiene más de $n - 1$ aristas), si seguimos la cadena de padres p , vamos a encontrar que $v \rightsquigarrow v$, pues el mejor camino usa un ciclo que involucra a v . Luego, podemos retroceder n pasos usando p , y vamos a caer en C , porque la distancia entre v y u es a lo sumo $n - 1$ aristas, y luego siguiendo $p[u], p[p[u]], \dots$, vamos a llegar a v . Una vez que llegamos a v , vamos a pasar por C una y otra vez si seguimos siguiendo p .

Luego, para recuperar un ciclo de peso negativo alcanzable desde s , podemos usar el siguiente algoritmo:

1: **procedure** BELLMAN-FORD-NEGATIVE-CYCLE($G = (V, E)$, $s \in V$, $w : E \rightarrow \mathbb{R}$)

```

2:    $d[v] \leftarrow \infty \forall v \in V$ 
3:    $d[s] \leftarrow 0$ 
4:    $p[v] \leftarrow \text{NULL} \forall v \in V$ 
5:   for  $i = 1$  to  $|V| - 1$  do
6:     for  $(u, v) \in E$  do
7:       if  $d[v] > d[u] + w(u, v)$  then
8:          $d[v] \leftarrow d[u] + w(u, v)$ 
9:          $p[v] \leftarrow u$ 
10:      end
11:    end
12:  end
13:   $z \leftarrow \text{NULL}$ 
14:  for  $(u, v) \in E$  do
15:    if  $d[v] > d[u] + w(u, v)$  then
16:       $z \leftarrow v$ 
17:    end
18:  end
19:  if  $z = \text{NULL}$  then
20:    return  $d, p$ 
21:  end
22:  for  $i = 1$  to  $|V|$  do
23:     $z \leftarrow p[z]$ 
24:  end
25:   $z_0 \leftarrow z$ 
26:   $C \leftarrow [z_0]$ 
27:  while  $p[z] \neq z_0$  do
28:     $z \leftarrow p[z]$ 
29:     $C \leftarrow C + [z]$ 
30:  end
31:  return  $C$ 
32: end

```

□

Ejercicio 4.4.13

Es nuestro primer día trabajando en un banco, en el departamento de comercio internacional. Sabemos que hay n monedas en el mundo, y tenemos una tabla $Q \in \mathbb{R}^{n \times n}$, que nos dice que si tenemos 1 unidad de la moneda i , se puede intercambiar por $0 < Q_{i,j}$ unidades de la moneda j .

Queremos saber si existe una secuencia $S = (s_1, s_2, \dots, s_k)$ de intercambios de monedas que podemos hacer, tal que al intercambiar una moneda s_1 por la moneda s_2 , y luego s_2 por s_3, \dots , y luego s_k por s_1 , terminamos con más dinero que con el que empezamos.

Dar un algoritmo que determine si es posible hacer esto. El algoritmo debería tardar tiempo $O(n^3)$ en el peor caso. Demostrar que es correcto.

Solución. Podemos construir un grafo dirigido pesado $G = (V, E)$, donde $V = \{1, \dots, n\}$ son las monedas, hay una arista entre todo par de vértices, y el peso de la arista (v_i, v_j) es $w(v_i, v_j) = -\log Q_{i,j}$.

Si conseguimos un ciclo de peso negativo en G , entonces sabremos que existe una cadena de transacciones que podemos hacer, yendo de la moneda a_1 a a_2 , dots, a_k , y finalmente a a_1 , tal que:

$$\begin{aligned} w(a_k, a_1) + \sum_{i=1}^{k-1} w(a_i, a_{i+1}) &< 0 \\ \log Q_{a_k, a_1} + \sum_{i=1}^{k-1} \log Q_{a_i, a_{i+1}} &> 0 \\ Q_{a_k, a_1} \times \prod_{i=1}^{k-1} Q_{a_i, a_{i+1}} &> 1 \end{aligned}$$

que es precisamente lo que significa obtener más dinero que con el que empezamos.

Podemos transformar A en B , con $B_{i,j} = -\log A_{i,j}$, y luego usar el algoritmo de Floyd-Warshall en B . Luego, si $B_{i,i} < 0$ para algún i , entonces existe un ciclo de peso negativo (que incluye a v_i). Como vimos arriba, la existencia este ciclo implica que existe una manera de intercambiar monedas que nos deja con más dinero del que empezamos.

Ejercicio 4.4.14

Sea $G = (V, E)$ un grafo dirigido pesado, con $V = \{v_1, \dots, v_n\}$, y sin ciclos de peso negativo. Se tiene una matriz $A \in \mathbb{R}^{n \times n}$, donde $A_{i,j}$ es la distancia entre v_i y v_j en G .

Se agrega un vértice v_{n+1} a G , con algunas aristas, obteniendo G' , con $n+1$ vértices.

Dar un algoritmo que calcule las distancias entre **todo** par de vértices en G' . El algoritmo debe hacer $O(n^2)$ operaciones en el peor caso. Demostrar que es correcto.

Solución. Vamos a construir una matriz $B \in \mathbb{R}^{(n+1) \times (n+1)}$, donde $B_{i,j}$ es la distancia entre v_i y v_j en G' . Realizamos el siguiente procedimiento:

```

1: procedure ADD-VERTEX( $A \in \mathbb{R}^{n \times n}$ ,  $w : E \rightarrow \mathbb{R}$ )
2:    $B \in \mathbb{R}^{(n+1) \times (n+1)} \leftarrow \text{null}$ 
3:   for  $l = 1$  to  $n$  do
4:      $B_{n+1,l} \leftarrow \min_{1 \leq j \leq n} w(n+1, j) + A_{j,l}$ 
5:      $B_{l,n+1} \leftarrow \min_{1 \leq j \leq n} w(j, n+1) + A_{l,j}$ 
6:   end
7:    $B_{n+1,n+1} \leftarrow \min_{1 \leq j \leq n} B_{n+1,j} + B_{j,n+1}$ 
8:   for  $i = 1$  to  $n$  do
9:     for  $j = 1$  to  $n$  do
10:       $B_{i,j} \leftarrow \min(A_{i,j}, B_{i,n+1} + B_{n+1,j})$ 
11:    end
12:  end
13: return  $B$ 
14: end
```

Demostración. Queremos demostrar que el algoritmo correctamente computa las distancias entre todo par de vértices en G' . Inicialmente, sabemos que A contiene las mínimas distancias entre todo par de vértices en G .

Veamos por qué cada una de las partes del algoritmo es correcto.

1. El camino más corto para ir desde v_{n+1} hasta v_l , para cada $1 \leq l \leq n$, empieza con alguna arista (v_{n+1}, v_j) , con $1 \leq j \leq n$, y luego va desde v_j hasta v_l usando el camino más corto en G (es decir, sin volver a pasar por v_{n+1} , pues v_{n+1} no está en G , sino en G'). Por eso es correcto asignar a $B_{n+1,l}$ el valor $\min_{1 \leq j \leq n} w(n+1, j) + A_{j,l}$, dado que A contiene las distancias en G .
2. Algo análogo sucede con $B_{l,n+1}$.
3. La mejor forma de ir desde v_{n+1} hasta v_{n+1} es ir desde v_{n+1} hasta algún vértice v_j con $1 \leq j \leq n$, usando el camino más corto en G , y luego volver a v_{n+1} , usando el camino más corto en G . Por eso es correcto asignar a $B_{n+1,n+1}$ el valor $\min_{1 \leq j \leq n} B_{n+1,j} + B_{j,n+1}$.
4. La mejor forma de ir desde v_i hasta v_j , con $1 \leq i, j \leq n$, puede o usar o no usar v_k . Si no lo usa, entonces es simplemente $A_{i,j}$. Si lo usa, entonces va desde v_i hasta v_k , usando el camino más corto en G , y luego desde v_k hasta v_j , también usando el camino más corto en G . Luego, la mejor forma de ir desde v_i hasta v_j es $\min(A_{i,j}, B_{i,n+1} + B_{n+1,j})$, que es lo que asignamos a $B_{i,j}$.

Este algoritmo inmediatamente nos da un algoritmo para distancias mínimas entre todo par de vértices en un grafo. Empezamos con un subgrafo generador por el primer vértice, y una matriz trivial de distancias (un único escalar, 0). Luego, $n - 1$ veces agregamos un vértice más, con las aristas que corresponde.

Nuestro algoritmo usa $O(n^2)$ operaciones en todos los casos, al ser una simple composición de ciclos. Luego, el algoritmo general para caminos mínimos entre todo par de vértices, que hace ADD-VERTEX $n - 1$ veces, va a usar $O(n^3)$ operaciones en el peor caso. Este algoritmo se conoce como el algoritmo de Dantzig[23]. □

5 Árboles generadores mínimos

Ejercicio 4.5.1

Demostrar la correctitud del algoritmo de Kruskal para árboles generadores mínimos.

Ejercicio 4.5.2

Demostrar la correctitud del algoritmo de Prim para árboles generadores mínimos.

6 Flujo

Definición 4.6.1

Una **red de flujo** G es una 5-tupla $G = (V, E, s, t, c)$, tal que (V, E) es un grafo, $s \in V, t \in V$, y $c : E \rightarrow \mathbb{R}_{\geq 0}$.

Un **flujo** en tal red es una función $f : E \rightarrow \mathbb{R}_{\geq 0}$, tal que:

- $0 \leq f(e) \leq c(e)$ para todo $e \in E$.
- Para todo $v \in V \setminus \{s, t\}$, $\sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u)$.

En tal caso decimos que el **valor de f** es $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$. Un flujo se dice **máximo** cuando tiene valor máximo entre todos los flujos en G .

Dada una tal red de flujo, un **corte** es una partición de V en (S, T) , tal que $s \in S$, y $t \in T$.

Dado un flujo f en esa red, el **flujo neto de (S, T)** se define como $f(S, T) =$

$\sum_{u \in S, v \in T} f(u, v) - \sum_{u \in S, v \in T} f(v, u)$. La **capacidad de (S, T)** se define como $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$.



Ejercicio 4.6.2

Sea $G = (V, E, s, t, c)$ una red de flujo, y f un flujo máximo en G . Sea $\lambda \in \mathbb{R}_{>0}$. Definimos c' como $c'(x) = \lambda c(x)$ para todo $x \in E$.

Sea f' un flujo máximo en $G' = (V, E, s, t, c')$.

Demostrar que $|f'| = \lambda |f|$.



Demostración. Vamos a construir una biyección explícita φ entre flujos en G , y flujos en G' .

Sea $f : E \rightarrow \mathbb{R}_{\geq 0}$ un flujo en G . Definimos $\varphi(f) : E \rightarrow \mathbb{R}_{\geq 0}$ como $\varphi(f)(e) = \lambda f(e)$ para todo $e \in E$. Notemos que $\varphi(f)$ es un flujo en G' , pues como f es un flujo en G , sabemos que $0 \leq f(e) \leq c(e)$ para todo $e \in E$, y luego $0 \leq \varphi(f)(e) = \lambda f(e) \leq \lambda c(e) = c'(e)$ para todo $e \in E$. Además, como f es un flujo, tenemos que para todo $v \in V \setminus \{s, t\}$, se cumple que

$$\begin{aligned}\sum_{u \in V} f(u, v) &= \sum_{u \in V} f(v, u), \text{ y luego } \sum_{u \in V} \varphi(f)(u, v) = \lambda \sum_{u \in V} f(u, v) = \\ \lambda \sum_{u \in V} f(v, u) &= \sum_{u \in V} \varphi(f)(v, u), \text{ y luego } \varphi(f) \text{ es un flujo en } G'.\end{aligned}$$

Notamos que φ es monótona en valores. Es decir, si $|f| \geq |g|$, entonces $|\varphi(f)| = \lambda |f| \geq \lambda |g| = |\varphi(g)|$. Luego, si f es un flujo máximo en G , entonces $\varphi(f)$ es un flujo máximo en G' . Entonces si f' es el valor de cualquier flujo máximo en G' , tenemos que $|f'| = |\varphi(f)| = \lambda |f|$, y luego $|f'| = \lambda |f|$, que es lo que queríamos demostrar. \square

Ejercicio 4.6.3

Sea $G = (V, E, s \in V, t \in V, c : E \rightarrow \mathbb{R}_{\geq 0})$ una red de flujo. Sea $f : E \rightarrow \mathbb{R}_{\geq 0}$ un flujo en G . Sea (S, T) un corte en G .

Mostrar que $|f| = f(S, T) \leq c(S, T)$.



Demostración. Como f es un flujo, sabemos que para todo $u \in V \setminus \{s, t\}$, tenemos $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$. Esto es lo mismo que decir $0 = \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u)$. Por definición, $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$. Si a esta última ecuación le sumamos la primera, por todo $u \in S \setminus \{s\}$, tenemos que

$$\begin{aligned}|f| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S \setminus \{s\}} \left(\sum_{v \in T} f(u, v) - \sum_{v \in T} f(v, u) \right) \\&= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S \setminus \{s\}} \sum_{v \in T} f(u, v) - \sum_{u \in S \setminus \{s\}} \sum_{v \in T} f(v, u) \\&= \sum_{v \in V} \left(f(s, v) \sum_{u \in S \setminus \{s\}} f(u, v) \right) - \sum_{v \in V} \left(f(v, s) + \sum_{u \in S \setminus \{s\}} f(v, u) \right) \\&= \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u)\end{aligned}$$

Ahora usamos que $V = S \sqcup T$,

$$\begin{aligned}&= \left(\sum_{v \in S} \sum_{u \in S} f(u, v) + \sum_{v \in T} \sum_{u \in S} f(u, v) \right) - \left(\sum_{v \in S} \sum_{u \in S} f(v, u) - \sum_{v \in T} \sum_{u \in S} f(v, u) \right) \\&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\&= f(S, T)\end{aligned}$$

Para ver una cota superior de $f(S, T)$, basta usar sólo la primer sumatoria:

$$\begin{aligned}|f| &= f(S, T) \\&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u), \text{ al ser } f(v, u) \geq 0 \quad \forall v, u \in V \\&\leq \sum_{v \in T} \sum_{u \in S} f(u, v) \\&\leq \sum_{v \in T} \sum_{u \in S} c(u, v) \\&= \sum_{u \in S, v \in T} c(u, v) \\&= c(S, T)\end{aligned}$$

□

Ejercicio 4.6.4

Sea $G = (V, E, s, t, c)$ una red de flujo, y f un flujo máximo en G . Sea $n = |V|, m = |E|$.

Ahora elegimos un $e \in E$, y construimos $G' = (V, E, s, t, c')$, tal que:

$$c'(x) = \begin{cases} c(x) & \text{si } x \neq e \\ c(x) + 1 & \text{si } x = e \end{cases}$$

Dar un algoritmo que encuentre un flujo máximo en G' . El algoritmo debe hacer $O(n + m)$ operaciones en el peor caso. Demostrar que el algoritmo es correcto.

Demostración. Primero vamos a citar dos teoremas que ven en la teórica.

Teorema 10

Sea G una red de flujo, y f un flujo en G . Sea G_f la red residual. f es un flujo máximo en G si y sólo si G_f no tiene caminos de aumento.

Por el teorema de flujo máximo - corte mínimo, como f es un flujo máximo, existe un corte (S, T) en G tal que $|f| = c(S, T)$. Veamos ahora cuál es la capacidad del corte (S, T) en G' . Si e no cruza la partición (S, T) , entonces $c'(S, T) = c(S, T)$. Si e cruza la partición, entonces $c'(S, T) = c(S, T) + 1$. En ambos casos, tenemos que $c(S, T) \leq c'(S, T) \leq c(S, T) + 1 = |f| + 1$.

Como f es un flujo en G , también es un flujo en G' , puesto que las condiciones de sumatoria en cada vértice se siguen cumpliendo, y las condiciones de capacidad también (pues meramente *agregamos* capacidad a e). Podemos entonces construir la red residual de f , G'_f .

Esto nos sugiere correr una iteración más del algoritmo de Edmonds-Karp en G'_f , y si encontramos un camino de aumento P , podemos aumentar el flujo en f en $c(P)$, obteniendo un flujo f' , y luego $|f'| = |f| + c(P) = |f| + \min_{e \in P} c(e)$. Como vimos, $|f'| \leq |f| + 1$, entonces $c(P)$ va a ser 1, si P existe, pues todos los flujos encontrados por Edmonds-Karp son de enteros, siendo las capacidades de G' enteros. Si no encontramos un camino, por el Teorema 10 f es un flujo máximo en G' , y luego $|f'| = |f|$. □

7 Matching

TODO: This.

8 Circuitos y caminos

Definición 4.8.1

Sea G un grafo. Un camino en G se dice **camino euleriano** cuando pasa por todas las aristas de G exactamente una vez.

Un ciclo de G se dice **ciclo euleriano** cuando pasa por todas las aristas de G exactamente una vez.

Si G tiene un ciclo euleriano, se dice que **G es euleriano**.

Definición 4.8.2

Sea G un grafo. Un camino en G se dice **camino Hamiltoniano** cuando pasa por todos los vértices exactamente una vez.

Un ciclo de G se dice **ciclo Hamiltiniano** cuando pasa por todos los vértices exactamente una vez.

Si G tiene un ciclo Hamiltoniano, entonces se dice que **G es Hamiltoniano**

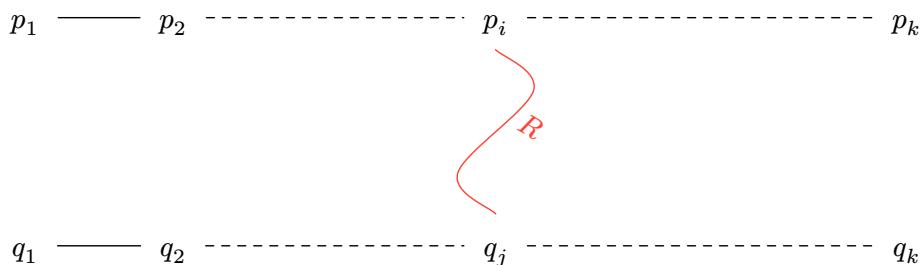
Ejercicio 4.8.3

Sea G un grafo conexo, y sean P y Q dos caminos de longitud máxima en G . Probar que P y Q comparten al menos un vértice.



Demostración. Por contradicción, asumamos que no. Luego, $P = [p_1, \dots, p_k]$ y $Q = [q_1, \dots, q_k]$ no comparten vértices. Notar que como ambos tienen longitud máxima, ambos tienen que tener la misma longitud, que llamamos k .

Como G es conexo, existen vértices p_i y q_j , tal que hay un camino R entre p_i y q_j . Sin pérdida de generalidad, podemos asumir que R no tiene vértices en P ni en Q , salvo p_i y q_j (siempre podemos quedarnos con el sub-camino de R que cruce los caminos, sin tocar a nadie de P ni Q en el medio).



Sin pérdida de generalidad, podemos asumir que $i \geq \lceil \frac{k}{2} \rceil$. Si esto no sucediera, simplemente revertimos el orden de los vértices de P . Por el mismo motivo, podemos asumir que $j \geq \lceil \frac{k}{2} \rceil$. Vemos que $|R| \geq 1$, pues si no, P y Q compartirían vértices.

Consideremos ahora el camino $\alpha = [p_1, p_2, \dots, p_i] + R + [q_j, \dots, q_1]$. Este camino tiene longitud $|\alpha| = i + |R| + j \geq 2\lceil \frac{k}{2} \rceil + |R| \geq k + |R| > k$. Esto no puede suceder, pues P y Q eran caminos de longitud máxima, k , y acabamos de encontrar un camino de longitud mayor a k .

Luego no puede pasar que P y Q sean disjuntos, y tienen que compartir algún vértice. \square

Ejercicio 4.8.4

Probar que si un grafo tiene algún vértice de grado impar, entonces no tiene un circuito euleriano.



Demostración. Un ciclo euleriano usa cada arista exactamente una vez, y termina donde comienza. Cada vez que tomamos una arista $u \rightarrow v$ en el camino, visitando a v , inmediatamente tomamos $v \rightarrow w$, saliendo de v . Luego, cada vez que usamos una arista hacia v , usamos otra que sale desde v . Como las aristas incidentes a v se usan siempre de a pares, y al terminar el ciclo todas las aristas son usadas exactamente una vez, si llamamos $f(v)$ al número de veces que C visita v , tenemos $d_G(v) = 2f(v)$. Luego, $d_G(v)$ es par, para todo $v \in V$. \square

Ejercicio 4.8.5

Probar que si un grafo tiene un camino euleriano, entonces tiene exactamente dos vértices con grado impar.



Demostración. Sea $G = (V, E)$ un grafo, tal que más de dos vértices en V tienen grado impar.

Supongamos que G tiene un camino euleriano, P , desde u a w . Sea $v \in V$, $v \neq u$, $v \neq w$. Si v aparece k veces en P , entonces como P usa todas las aristas de G exactamente una vez, y cada vez que P pasa por v toca dos de sus aristas incidentes, v debe tener grado $2k$. En el caso de u y w , la primera vez que P visita a u , y la última vez que visita a w , P sólo usa una de sus aristas incidentes.

Luego los únicos dos vértices que tienen grado impar son u y w , y todos los otros vértices tienen grado par. Notar que como P es un camino euleriano, y no un ciclo, sabemos que u y w son distintos. \square

Ejercicio 4.8.6

Sea $G = (V, E)$ un grafo dirigido, tal que para todo par de vértices $u, v \in V$, exactamente uno de (u, v) o (v, u) está en E .

Probar que G contiene un camino Hamiltoniano.



Demostración. Vamos a probar esto por inducción. Por comodidad, defino $Q(G = (V, E))$: $\forall u, v \in V. ((u, v) \in E) \neq ((v, u) \in E)$. Definimos entonces $P(n)$: Para todo grafo dirigido G con n vértices tal que $Q(G)$, G tiene un camino Hamiltoniano. Por comodidad, dedo un subconjunto $X \subseteq V$ de vértices de un grafo $G = (V, E)$, definimos $G[X] = (X, \{(a, b) \in E \mid a \in X, b \in X\})$, el subgrafo inducido por X en G .

1. Caso base, $P(0)$. Esto es trivialmente cierto porque no hay ningún grafo sin vértices, luego no existe ningnú tal G .
2. Caso base, $P(1)$. También trivial, si $V = \{v\}$, $[v]$ un camino Hamiltoniano.
3. Paso inductivo. Sea $n \in \mathbb{N}$, $n \geq 2$. Asumimos que vale $P(k)$ para todo $k \in \mathbb{N}$, $k < n$, queremos probar $P(n)$. Sea $G = (V, E)$ un grafo dirigido, con $|V| = n$. Como $n \geq 2$, sea cuquier $v \in V$, y $W = V \setminus \{v\}$. Sea $W^\rightarrow = \{w \in W \mid (w, v) \in E\}$, y $W^\leftarrow = \{w \in W \mid (v, w) \in E\}$. Como para todo $w \in W \subseteq V$ sabemos que o bien (w, v) in E , o bien $(v, w) \in E$, y ocurre exactamente una de las dos, entonces vemos que $W = W^\rightarrow \sqcup W^\leftarrow$, la unión disjunta.

Como $W = V \setminus \{v\}$, tenemos $|W| = |V| - 1 = n - 1 < n$. Como ambos $W^\rightarrow \subseteq W$ y $W^\leftarrow \subseteq W$, tenemos $|W^\rightarrow| < n$, y $|W^\leftarrow| < n$, y es más, $|W^\rightarrow| + |W^\leftarrow| = |W| = n - 1$.

- Si $W^\leftarrow = \emptyset$, entonces $W = W^\rightarrow$, y luego $|W^\rightarrow| = |W| = n - 1 \geq 1$. Construimos $G' = G[W^\rightarrow]$, con $n - 1$ vértices. Como $Q(G)$, sigue valiendo $Q(G')$. Ahora usamos $P(n - 1)$ para obtener un camino Hamiltoniano $R = [r_1, \dots, r_{n-1}]$ en G' . Como es Hamiltoniano, no repite vértices. R existe en G , por ser G' subgrafo de G . Como $r_{n-1} \in W^\rightarrow$, por definición $(r_{n-1}, v) \in E$. Sea $R' = [r_1, \dots, r_{n-1}, v]$. Como R es un camino en G' , y $v \notin$

$V(G')$, entonces $v \notin R$. Luego R' es un camino de longitud n en G que no repite vértices, y luego es Hamiltoniano en G .

- Si $W^\rightarrow = \emptyset$, pasa algo análogo, tomando el subgrafo inducido por W^\leftarrow en G .
- Si ninguna de las dos particiones está vacía, sea $k = |W^\rightarrow|, t = |W^\leftarrow|$, con $1 \leq k, t < n$. Tomemos $G^\rightarrow = G[W^\rightarrow], G^\leftarrow = G[W^\leftarrow]$, y como seguimos teniendo $Q(G^\rightarrow)$ y $Q(G^\leftarrow)$ por ser subgrafos inducidos de G , usamos $P(k)$ y $P(t)$, y vemos que ambos tienen un camino Hamiltoniano. Sean $R^\rightarrow = [x_1, \dots, x_k]$ y $R^\leftarrow = [y_1, \dots, y_t]$ tales caminos en G^\rightarrow y G^\leftarrow , respectivamente. Por ser Hamiltonianos, sabemos que no repiten vértices, y sabiendo que $|W^\rightarrow| + |W^\leftarrow| = n - 1$, sabemos que $k + t = n - 1$. Como $W^\rightarrow \cap W^\leftarrow = \emptyset$, vemos que R^\rightarrow y R^\leftarrow no comparten vértices. Como $x_k \in W^\rightarrow$, tenemos que $(x_k, v) \in E$. Como $y_1 \in W^\leftarrow$, tenemos que $(v, y_1) \in E$. Luego, $R = R^\rightarrow + [v] + R^\leftarrow = \left[\underbrace{x_1, \dots, x_k}_{\in W^\rightarrow}, v, \underbrace{y_1, \dots, y_t}_{\in W^\leftarrow} \right]$ es un camino en G de longitud $k + t + 1 = n - 1 + 1 = n$. Como $v \notin W^\rightarrow$ y $v \notin W^\leftarrow$, R no repite vértices, y luego es Hamiltoniano en G .

□

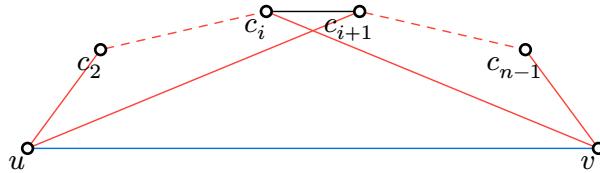
Ejercicio 4.8.7 (Teorema de Bondy-Chvátal[24])

Sea G un grafo de n vértices, y sean u y v vértices no adjacentes, tal que $d_G(u) + d_G(v) \geq n$.

Probar que si $G + \{u, v\}$ es Hamiltoniano, entonces G también lo es.

♦

Demostración. Llamemos $G = (E, V)$, y $n = |V|$. Sea C un ciclo Hamiltoniano de $G + \{u, v\}$. Si $\{u, v\} \notin C$, entonces C también es un ciclo Hamiltoniano en G , y terminamos. Luego, asumamos que $\{u, v\} \in C$. Consideraremos el camino $C - \{u, v\} = [c_1 = u, c_2, \dots, c_n = v]$. Sean $S_u = \{i \mid 1 \leq i \leq n - 1, \{u, c_{i+1}\} \in E\}$, y $S_v = \{i \mid 1 \leq i \leq n - 1, \{c_i, v\} \in E\}$. Vemos que $|S_u| = d_G(u)$, y $|S_v| = d_G(v)$. Luego, $|S_u| + |S_v| \geq n$, por hipótesis, mientras que luego $|S_u \cup S_v| \leq n - 1$, dado que en sus definiciones, $i \in [0, \dots, n - 1]$. Luego, $S_u \cap S_v \neq \emptyset$. Sea entonces $i \in S_u \cap S_v$.



Como $c_1 = u$ y $c_n = v$ no son adjacentes en G , vemos que $2 \leq i \leq n - 2$. Pero ahora, $C' = [c_1 = u, c_2, \dots, c_i, v = c_n, c_{n-1}, \dots, c_{i+1}, c_1 = u]$ es un ciclo Hamiltoniano en G , y luego G es Hamiltoniano.

□

Ejercicio 4.8.8 (Teorema de Dirac)

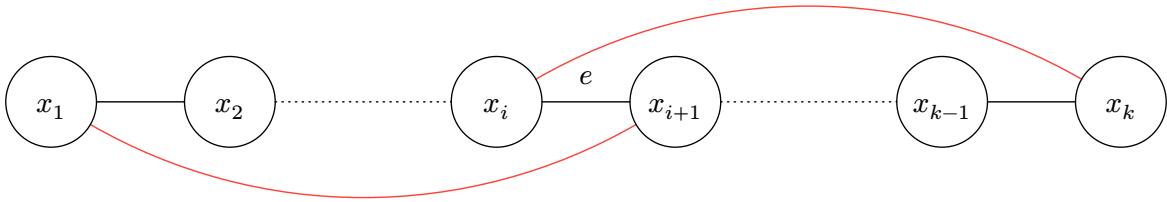
Si G tiene $n \geq 3$ vértices y mínimo grado $\delta(G) \geq \frac{n}{2}$, entonces G es Hamiltoniano.

♦

Demostración. Sea C la componente conexa más chica de $G = (V, E)$, y $v \in C$. Como $\delta(G) \geq \frac{n}{2}$, tenemos $d(v) \geq \frac{n}{2}$. Como al menos $1 + \frac{n}{2}$ vértices (i.e. más de la mitad de los vértices) están en la componente conexa *más chica*, tiene que haber sólo una componente conexa. Luego G es conexo.

Sea $P = [x_1, \dots, x_k]$ un camino de máxima longitud en G . Como $\delta(G) \geq \frac{n}{2}$, entonces $d(x_k) \geq \frac{n}{2}$, y $d(x_1) \geq \frac{n}{2}$. Si algún vecino de x_1 o x_k no estuviera en P , podríamos extender alguno de los extremos de P , obteniendo un camino de mayor longitud de P . Por ende todos los vecinos de x_1 , y todos los vecinos de x_k están en P .

Probemos que existen dos vértices, x_j y $x_{j+1} \in V$, tal que $\{x_j, x_k\} \in E$, y $\{x_{j+1}, x_1\} \in E$. Sea $A = \{\{x_t, x_{t+1}\} \mid \{x_{t+1}, x_1\} \in E\}$, y $B = \{\{x_t, x_{t+1}\} \mid \{x_t, x_k\} \in E\}$. Vemos que $|A| = d(x_1) \geq \frac{n}{2}$, y $|B| = d(x_k) \geq \frac{n}{2}$. Como P tiene a lo sumo $n - 1$ aristas, y $|A| + |B| \geq n$, por el principio del palomar existe alguna arista que está en ambos $|A|$ y $|B|$. Esta arista, luego, es de la forma $e = \{a, b\}$ con $\{a, x_k\} \in E$ (pues $e \in A$) y $\{x_1, b\} \in E$ (pues $e \in B$).



Sea $C = [x_1, x_2, \dots, x_i, x_k, x_{k-1}, \dots, x_{i+1}, x_1]$, de longitud $k + 1$. C es un ciclo.

Si no fuera Hamiltoniano, entonces como G es conexo, existiría un vértice $y \in V$, adjacente a algún $x_j \in C$, con $y \notin C$. Luego existe $\{x_j, y\} \in E$. Pero entonces podríamos tomar C , sacarle una arista incidente a x_j , y agregarle e , y obtendríamos un *camino* en G , de longitud $k + 1$, que no puede pasar pues P , de longitud k , era de longitud máxima.

Luego C es un ciclo Hamiltoniano en G , y luego G es Hamiltoniano. □

9 Coloreo

Definición 4.9.1

Sea $G = (V, E)$ un grafo. Un **coloreo de G con k colores** es una función $f : V \rightarrow \{1 \dots k\}$, tal que para todo $(u, v) \in E$, $f(u) \neq f(v)$.

Si existe un coloreo de G con k colores, se dice que G es **k -coloreable**.

El **número cromático** de G es el mínimo k tal que G es k -coloreable, y se denota $\chi(G)$.

Ejercicio 4.9.2

Dado un $n \in \mathbb{N}$, el grafo C_n tiene n vértices v_1, \dots, v_n , y hay una arista entre v_i y $v_{(i+1) \bmod n}$ para todo $0 \leq i \leq n$.

¿Para cuáles $n \in \mathbb{N}$, C_n es 2-coloreable?

Demostración. Sea $C_n = (V, E) = [v_1, \dots, v_n]$ el ciclo de n vértices. Supongamos que tenemos $f : V \rightarrow \{0, 1\}$ que colorea a C_n con 2 colores. Luego, claramente los valores de $f(v_1), f(v_2), f(v_3), \dots$ tienen que alternarse, pues si no no sería un colooreo válido. La única restricción va a ser qué pasa cuando el ciclo vuelve a v_1 . Supongamos sin pérdida de generalidad que $f(v_1) = 1$, entonces vamos a tener $f(v_i) = i \bmod 2$. Entonces, si tuvieramos un ciclo de longitud impar, tendríamos $f(v_n) = n \bmod 2 = 1$, pero $f(v_1) = 1$, y $\{v_n, v_1\} \in E$. Luego, si queremos que C_n sea 2-coloreable, debemos tener $n \equiv 0 \pmod{2}$. \square

Ejercicio 4.9.3

Probar que el número cromático de todo árbol es menor o igual a 2.

Demostración. Sea T un árbol con n vértices.

- Si $n = 1$, puedo colorear al único vértice de un color, y tengo un 1-coloreo. No existen 0-coloreos, luego el número cromático de T es uno.
- Si $n > 1$, entonces al ser T bipartito, podemos tomar las dos particiones U, W de sus vértices, colorear U de un color y W de otro, y obtenemos un 2-coloreo válido. Como un árbol es conexo, y hay más de un vértice, debe existir al menos una arista $e = \{u, w\}$, con $u \in U$ y $w \in W$. Esa arista previene que u y w tengan el mismo color, luego no existen 1-coloreos. Como se puede colorear con 2 colores, y no con 1 color, el número cromático de T es 2.

\square

Ejercicio 4.9.4 (Teorema de Ramsey)

Probar que toda forma de colorear las **arestas** de K_6 con dos colores, azul y rojo, tiene un triángulo rojo o un triángulo azul.

Demostración. Sea $G = (V, E) = K_6$ el grafo completo en 6 vértices, y sea $f : E \rightarrow \{\bullet, \circ\}$ una manera de colorear E con dos colores. Sea $v \in V$. Como $d_G(v) = 5$, tiene o 3 aristas incidentes \bullet , o aristas incidentes \circ . Sin pérdida de generalidad, asumamos que tiene tres aristas incidentes \bullet . Sean x, y, z los vértices que unen a estas aristas con v . Luego, $f(\{v, x\}) = f(\{v, y\}) = f(\{v, z\}) = \bullet$. Si $f(\{x, y\}) = f(\{y, z\}) = f(\{z, x\}) = \bullet$, encontramos un triángulo \bullet , y terminamos. De otra forma, existe una arista \circ entre algunos de esos tres vértices. Sin pérdida de generalidad, asumamos que es $f(\{x, y\}) = \circ$. Entonces tenemos un triángulo \bullet , $(\{v, x\}, \{x, y\}, \{y, v\})$, y terminamos. \square

Ejercicio 4.9.5

Sea G un grafo de n vértices, denotemos $\alpha(G)$ el máximo tamaño de un conjunto independiente en G .

Probar que $\frac{n}{\alpha} \leq \chi$.

Demostración. Sea $G = (V, E)$. Nombramos por comodidad $\chi = \chi(G)$, y $\alpha = \alpha(G)$, y $n = |V|$. y consideremos un colojo óptimo $f : V \rightarrow \{1, \dots, \chi\}$. Sean $S_i = f^{-1}(i) \subseteq V$ los conjuntos de cada color, para $1 \leq i \leq \chi$. Si $u, v \in S_i$, entonces $\{u, v\} \notin E$, puesto que f es un colojo válido. Luego S_i es un conjunto independiente.

Como $V = \bigsqcup_{1 \leq i \leq \chi} S_i$, entonces $n = \sum_{i=1}^{\chi} |S_i|$. Como cada S_i es un conjunto independiente, $|S_i| \leq \alpha$. Luego, $n \leq \sum_{i=1}^{\chi} \alpha = \chi\alpha$, y luego $\frac{n}{\alpha} \leq \chi$. \square

Ejercicio 4.9.6

Sea $G = (V, E)$ un grafo con $n = |V|$ vértices, y \overline{G} su complemento.

Demostrar que $\chi(G) + \chi(\overline{G}) \leq n + 1$.



Demostración. Por inducción. Si $n = 1$, $G = \overline{G}$, y $\chi(G) = \chi(\overline{G})$, luego $\chi(G) + \chi(\overline{G}) = 1 + 1 = 2 \leq 1 + 1$. En el paso inductivo, si $n > 1$, podemos tomar G y sacarle un vértice v , obteniendo $G' = G - v$. También llamamos $\overline{G}' = \overline{G} - v$. Usamos la hipótesis inductiva en $n - 1$, y obtenemos un colojo con k colores de G' , y un colojo con l colores de \overline{G}' , tal que $k + l \leq (n - 1) + 1 = n$.

1. Si $d_G(v) < k$, extendemos el colojo de G' a un colojo de G con k colores. Coloreamos \overline{G} usando el colojo de l colores de \overline{G}' , y usando un nuevo color para v que agregamos. Luego, $\chi(G) + \chi(\overline{G}) \leq (n - 1 + 1) + 1 = n + 1$, que es lo que queríamos demostrar.
2. Si $d_G(v) \geq k$, entonces $d_{\overline{G}}(v) \leq n - k = l - 1 < l$, y podemos extender el l -colojo de \overline{G}' a un l -colojo de \overline{G} . En G , usamos el k -colojo de G' , y usamos un color nuevo para v . Luego, $\chi(\overline{G}) + \chi(G) \leq l + k + 1 = n + 1$, que es lo que queríamos demostrar.



Ejercicio 4.9.7

Sea $G = (V, E)$ un grafo con $n = |V|$ vértices, y \overline{G} su complemento.

Demostrar que $\chi(G) + \chi(\overline{G}) \geq 2\sqrt{n}$.



Demostración. Consideremos $G \cup \overline{G} = K_n$, el grafo completo. Si $\chi(G)$ -coloreamos G con c , y $\chi(\overline{G})$ -coloreamos \overline{G} con c' , entonces podemos construir un $\chi(G)\chi(\overline{G})$ -colojo de K_n , coloreando a cada vértice $v \in V$ con $(c(v), c'(v))$. Como $\chi(K_n) = n$, entonces $n \leq \chi(G)\chi(\overline{G})$.

Ahora bien:

$$\begin{aligned}
& (\chi(G) - \chi(\overline{G}))^2 \geq 0 \\
& \chi(G)^2 - 2\chi(G)\chi(\overline{G}) + \chi(\overline{G})^2 \geq 0 \\
& \chi(G)^2 + 2\chi(G)\chi(\overline{G}) + \chi(\overline{G})^2 \geq 4\chi(G)\chi(\overline{G}) \\
& (\chi(G) + \chi(\overline{G}))^2 \geq 4\chi(G)\chi(\overline{G}) \geq 4n \\
& (\chi(G) + \chi(\overline{G}))^2 \geq 4n \\
& \chi(G) + \chi(\overline{G}) \geq \sqrt{4n} \\
& \chi(G) + \chi(\overline{G}) \geq 2\sqrt{n}
\end{aligned}$$

□

Ejercicio 4.9.8

Sea $G = (V, E)$ un grafo con $n = |V|$ vértices, y \overline{G} su complemento.

Demostrar que $n \leq \chi(G)\chi(\overline{G}) \leq \left(\frac{n+1}{2}\right)^2$.

Para la segunda desigualdad, puede serles útil la desigualdad aritmética-geométrica.

◆

Demostración. Consideremos el grafo $H = (V \times V, E')$, con $E' = \{(u, v), (x, y) \mid \{u, x\} \in E \vee \{v, y\} \notin E\}$. Es decir, un par (u, v) está conectado a un par (x, y) , exactamente cuando u está conectado a x en G , o v está conectado a y en \overline{G} .

Sea c un $\chi(G)$ -coloreo para G , y c' un $\chi(\overline{G})$ -coloreo para \overline{G} . Obtenemos entonces un coloreo f con $\chi(G)\chi(\overline{G})$ colores para H , donde $f((u, v)) = (c(u), c'(v))$, para cualquier $(u, v) \in V \times V$. Veamos que f es un coloreo válido. Si $\{(u, x), (v, y)\} \in E'$, entonces o bien $\{u, v\} \in E$, o bien $\{x, y\} \notin E$. En el primer caso, $c(u) \neq c(v)$, y en el segundo, $c'(x) \neq c'(y)$. Luego, el color asignado a (u, x) no puede ser igual al color asignado a (v, y) . Luego, f es un coloreo válido, y $\chi(G) \leq \chi(G)\chi(\overline{G})$.

Ahora bien, tomemos el conjunto de vértices de H , $F = \{(v, v) \mid v \in V\}$. Sea cualquier par de vértices en F , $a = (v, v)$, y $b = (w, w)$. Entonces o bien $\{v, w\} \in E$, o bien $\{v, w\} \notin E$. En el primer caso, $\{(v, v), (w, w)\} \in E'$ porque $(v, w) \in E$. En el segundo, $\{(v, v), (w, w)\} \in E'$ porque $(v, w) \notin E$. Luego, para cada par de vértices distintos en F , tenemos una arista entre ellos en H . Luego F es una clique en H , de n vértices, y luego todo coloreo de H debe asignarle n colores distintos a los vértices de F . Luego, $\chi(G) \geq n$.

Juntando ambas ecuaciones, vemos que $n \leq \chi(G)\chi(\overline{G})$.

Para ver que $\chi(G)\chi(\overline{G}) \leq \left(\frac{n+1}{2}\right)^2$, podemos usar la desigualdad aritmética-geométrica. Esta nos dice que para todo x, y reales no-negativos, tenemos $\frac{x+y}{2} \geq \sqrt{xy}$. Aplicando esto a $x = \chi(G)$, $y = \chi(\overline{G})$, sabemos que $\frac{\chi(G)+\chi(\overline{G})}{2} \geq \sqrt{\chi(G)\chi(\overline{G})}$, o lo que es lo mismo,

$\left(\frac{\chi(G)+\chi(\overline{G})}{2}\right)^2 \geq \chi(G)\chi(\overline{G})$. Por la desigualdad del Ejercicio 4.9.6, sabemos que $\chi(G) + \chi(\overline{G}) \leq n + 1$.

Luego, $\chi(G)\chi(\overline{G}) \leq \left(\frac{n+1}{2}\right)^2$, que es lo que queríamos demostrar. \square

Ejercicio 4.9.9

Sea G un grafo, denotemos $\Delta(G)$ el máximo grado de cualquier vértice en G .

Probar que $\chi \leq \Delta + 1$.



Demostración. Vamos a demostrar esto con un algoritmo, el algoritmo greedy para coloreo.

```
def color(V: list[int], E: dict[int, list[int]]) -> list[int]:
    (V, E) = G
    n = len(V)
    Delta = max(len(vs) for vs in E.values())
    all_colors = set(range(Delta + 1))
    colors = [-1 for _ in range(n)]
    for (i, v) in enumerate(V):
        used = set(colors[w] for w in E[v] if colors[w] != -1)
        colors[i] = min(all_colors - used)
    return colors
```

La variable `Delta` representa $\Delta(G)$, el máximo grado entre todos los vértices. `all_colors` es el conjunto $\{0, 1, \dots, \Delta\}$. El invariante que mantenemos es que al terminar la i -ésima iteración, le asignamos correctamente un color entre 0 y Δ a los primeros i vértices. Como hay sólo Δ vecinos, `len(used)` siempre tiene como mucho Δ elementos. Como `len(all_colors) = Delta + 1`, siempre hay algún color no usado al calcular `all_colors - used`, y por lo tanto podemos tomar el mínimo color en esa resta. Como estamos sacando `used` de `all_colors`, nunca vamos a asignarle a `colors[i]` un color idéntico a `colors[j]` con $j \in E[v]$. Luego, esta asignación de colores nos da un colooreo válido de los primeros i vértices, si ya teníamos un colooreo válido de los primeros $i - 1$ vértices. Al terminar el ciclo, coloreamos los n vértices.

Este algoritmo, entonces, colorea G usando a lo sumo $\Delta + 1$ colores, que están en `all_colors`. Como el algoritmo termina, y asigna un $(\Delta + 1)$ -colooreo válido, entonces $\Delta + 1 \geq \chi(G)$, donde $\chi(G)$ es el número cromático de G . \square

Ejercicio 4.9.10

Sea $G = (V, E)$ un grafo, y sea $v \in V$.

Probar que $\chi(G - v) \in \{\chi(G), \chi(G) - 1\}$.



Demostración. Sea $H = G - v$. Si $f : G \rightarrow \{1, \dots, \chi(G)\}$ es un colooreo óptimo de G , entonces $g : H \rightarrow \{1, \dots, \chi(G)\}$, $g(v) = f(v)$ es un $\chi(G)$ -colooreo válido de H . Luego, $\chi(H) \leq \chi(G)$.

Sólo nos queda probar que $\chi(H) \geq \chi(G) - 1$. Sea $f : V \setminus \{v\} \rightarrow \{1, \dots, \chi(H)\}$ un coloreo óptimo de H . Podemos entonces agregar v a H , con las mismas aristas que tenía v en G . Vamos a definir:

$$g : V \rightarrow \{1, \dots, \chi(H) + 1\}$$

$$g(w) = \begin{cases} f(w) & \text{si } w \neq v \\ \chi(H) + 1 & \text{si } w = v \end{cases}$$

Como estamos agregando un vértice nuevo, v , que tiene un color distinto al color de todos los otros vértices (pues $f(w) \leq \chi(H) \forall w \in V \setminus \{v\}$), entonces g no introduce conflictos de colores. Vemos que g es, entonces, un $(\chi(H) + 1)$ -coloreo válido de G . Por lo tanto, $\chi(G) \leq \chi(H) + 1$, o lo que es lo mismo, $\chi(H) \geq \chi(G) - 1$, que es lo que queríamos demostrar. \square

10 Planaridad

Definición 4.10.1

Un grafo G es **planar** cuando existe una forma de dibujarlo en el plano, de tal forma que las aristas no se crucen.



Teorema 11

En un grafo planar con $m > 1$ aristas, toda cara tiene al menos 3 aristas en su borde. Esto incluye la cara exterior.

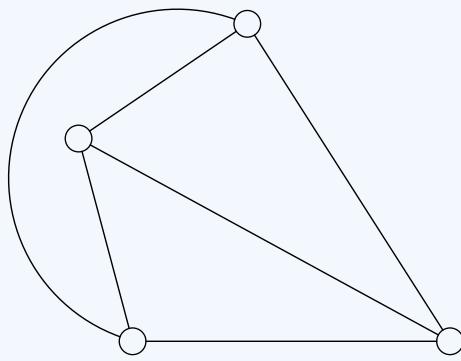


Ejercicio 4.10.2 (Formula de Euler)

Sea G un grafo conexo y planar, con n vértices, m aristas, y dibujado con r caras (incluyendo una cara exterior).

Probar que $r = m - n + 2$.

Por ejemplo, el siguiente grafo tiene $n = 4$ vértices, $m = 6$ aristas, y $r = 4$ caras:



Demostración. Vamos a probar esto por inducción en r , el número de caras. Sea $P(r)$: Todo grafo $G = (V, E)$ conexo y planar con r caras cumple que $|V| - |E| + r = 2$.

1. Caso base, $P(1)$. Sea $G = (V, E)$ un grafo conexo planar con $r = 1$ caras, y sean $n = |V|$ y $m = |E|$. Como $r = 1$, no podemos tener ciclos, pues un ciclo generaría una cara, y ya tenemos una (la externa). Luego, al ser G conexo y acíclico, G es un árbol. Luego, $m = n - 1$, y tenemos $n - m + r = n - (n - 1) + 1 = 2$, que muestra $P(1)$.
2. Paso inductivo. Sabemos $P(r)$, queremos probar $P(r + 1)$. Sea $G = (V, E)$ un grafo conexo y planar, con $r + 1 \geq 2$ caras. Si G fuera un árbol, $r = 1$, luego G no es un árbol, y al ser conexo, tiene un ciclo C . Sea $e \in C$ cualquiera arista en C . Sea ahora $G' = (V, E - \{e\})$. Como e pertenecía a un ciclo, G' sigue siendo conexo, con $|E - \{e\}| = |E| - 1$. Al haber roto un ciclo sacando e , G' tiene una cara menos que G . Luego, usando $P(r)$, vemos que $|V| - (|E| - 1) + r = 2$. Esto nos dice que $|V| - |E| + (r + 1) = 2$, que es prueba $P(r + 1)$.

Luego probamos $P(r)$ para todo $r \in \mathbb{N}$, $r \geq 1$. Como todo grafo planar tiene un número positivo de caras, esto prueba que todo grafo conexo planar cumple la ecuación. \square

Ejercicio 4.10.3

Sea G un grafo planar de n vértices y m aristas, con $n \geq 3$.

Probar que $m \leq 3n - 6$.



Demostración. Vamos a probar esto para grafos conexos. Si nos dan un grafo no conexo, podemos agregarle aristas hasta hacerlo conexo, sin perder planaridad, y vamos a ver que la desigualdad sigue valiendo. Como estamos aumentando m , sin cambiar n , valía antes de agregar las aristas.

Sea entonces $G = (V, E)$ un grafo planar conexo, con $n = |V|$, $m = |E|$, y r caras. Si $m \leq 2$, esto es cierto pues $3n - 6 \geq 3$. De otra manera, podemos usar el Teorema 11, y sabemos que toda cara tiene al menos 3 aristas en su borde. Para cada $i \in \mathbb{N}$, f_i el número de caras con exactamente i aristas en G . Luego, la siguiente suma va a contar a cada arista dos veces (una vez en cada una de las dos caras que parte, o dos veces en la misma cara si la arista está enteramente en una cara):

$$\begin{aligned} 2m &= 1f_2 + 2f_2 + 3f_3 + \dots \\ &= \sum_{i \in \mathbb{N}} if_i \\ &= \sum_{\substack{i \in \mathbb{N}, \\ i \geq 3}} if_i \\ &\geq \sum_{i \in \mathbb{N}} 3f_i \\ &= 3 \sum_{i \in \mathbb{N}} f_i \\ &= 3r \end{aligned}$$

Por el teorema de Euler, $n - m + r = 2$. Luego, $r = m - n + 2$, y luego $3r = 3m - 3n + 6$. Usando la desigualdad de arriba, $3m - 3n + 6 \leq 2m$. Simplificando, $m - 3n + 6 \leq 0$, y luego $m \leq 3n - 6$. \square

Ejercicio 4.10.4

Sea G un grafo conexo y planar, y sea $\delta(G)$ el mínimo grado entre todos los vértices de G .

Probar que $\delta(G) \leq 5$.



Demostración. Sea $G = (V, E)$ un grafo conexo y planar, con $n = |V|$, $m = |E|$, y r caras.

Si $n \leq 2$, la propiedad es obvia.

Si $n \geq 3$, contemos:

$$2m = \sum_{v \in V} d_G(v) \geq 6n$$

La primera igualdad es porque en todo grafo tenemos $\sum_{v \in V} d_G(v) = 2m$, y la segunda es porque $d_G(v) \geq 6$ para todo $v \in V$, por el enunciado.

Por el ejercicio anterior, vemos que $m \leq 3n - 6$. Luego, $2m \leq 6n - 12$. Pero entonces tenemos $6n \leq 2m \leq 6n - 12$, que no tiene sentido.

Luego no puede ser que todos los vértices tienen grado al menos 6, y tiene que haber algún vértice con grado a lo sumo 5. \square

11 Homomorfismo e isomorfismo de grafos

Definición 4.11.1

Sean $G = (V, E)$, $H = (V', E')$ grafos. Se dice que una función $f : V \rightarrow V'$ es un **homomorfismo de grafos** cuando para todo $u, v \in V$, $\{u, v\} \in E \Rightarrow \{f(u), f(v)\} \in E'$.

Un **isomorfismo de grafos** entre G y H es una función $f : V \rightarrow V'$, tal que para todo $u, v \in V$, $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$.

Una definición análoga vale para grafos dirigidos, donde las aristas son pares ordenados, en vez de conjuntos de tamaño 2.



Ejercicio 4.11.2

Sean G, H, K grafos, y $f : G \rightarrow H$, $g : H \rightarrow K$ homomorfismos. Probar que $g \circ f$ es un homomorfismo.



Demostración. Para ver que $g \circ f$ es un homomorfismo, tenemos que tomar dos vértices $u, v \in V(G)$, y ver que si $\{u, v\} \in E(G)$, entonces $\{g(f(u)), g(f(v))\} \in E(K)$.

Sean u, v tales que $\{u, v\} \in E(G)$. Entonces como f es un homomorfismo, $\{f(u), f(v)\} \in E(H)$. Como g es un homomorfismo, $\{g(f(u)), g(f(v))\} \in E(K)$, que es lo que queríamos demostrar. \square

Ejercicio 4.11.3

Sean G, H grafos, y $f : G \rightarrow H$ un isomorfismo. Probar que si G es conexo, entonces H también lo es.



Demostración. Sean $(V_H, E_H) = H$, y $(V_G, E_G) = G$. Vamos a probar que para todo par de vértices en H , hay un camino entre ellos. Sean $u, v \in V_H$ cualquier par de vértices. Como f es un isomorfismo, en particular es biyectiva, y luego existen $x = f^{-1}(u)$, $y = f^{-1}(v)$. Como G es conexo, existe un camino $P = [x = p_1, p_2, \dots, p_k = y]$ en G . Para todo $1 \leq i < k$, $\{p_i, p_{i+1}\} \in E_G$. Luego, como f es un isomorfismo, para todo $1 \leq i < k$, $e = \{f(p_i), f(p_{i+1})\} \in E_H$. Consideremos $P' = [f(x) = f(p_1), f(p_2), \dots, f(p_k) = f(y)]$. Como vimos, cada transición es una arista en E_H . Luego, P' es un camino en H , entre $f(x) = u$, y $f(y) = v$. \square

Ejercicio 4.11.4

Sean $G = (V, E)$ y $H = (V', E')$ grafos.

Notemos por $\chi(G)$ el número de coloedo de un grafo G , y $\omega(G)$ el tamaño de la clique máxima en G .

Mostrar que si hay un homomorfismo $f : G \rightarrow H$, entonces:

1. $\omega(G) \leq \omega(H)$
2. $\chi(G) \leq \chi(H)$



Demostración. Notemos $G = (V, E)$, $H = (W, F)$.

1. Sea $K \subseteq V$ una clique máxima en G . Luego $|K| = \omega(G)$. Sea $K' = \{f(w) \mid w \in K\} \subseteq W$. Sean $u, v \in K$. Entonces $\{u, v\} \in E$, y por lo tanto, $\{f(u), f(v)\} \in F$. H es un grafo, y no pseudografo, $f(u) \neq f(v)$. Luego, todos los vértices en K van a parar a vértices distintos en K' , y luego $f|_K : K \rightarrow K'$ es biyectiva. Entre cada par de vértices $x, y \in K'$, vamos a encontrar entonces una arista $\{f^{-1}(x), f^{-1}(y)\}$ en G , pues $f^{-1}(x)$ y $f^{-1}(y)$ son vértices distintos en K , una clique. Luego K' también es una clique. Como $f|_K$ es una biyección, tenemos que $|K'| = |K| = \omega(G)$. Luego, H contiene una clique de tamaño $\omega(G)$, y entonces la clique máxima de H tiene tamaño como mínimo $\omega(G)$. Luego, $\omega(G) \leq \omega(H)$.
2. Sea $g : W \rightarrow \{1, \dots, \chi(H)\}$ un coloedo óptimo de H . Vamos a ver que $g \circ f$ es un $\chi(H)$ -coloedo de G . Claramente le asigna a cada vértice de H un número entre 1 y $\chi(H)$, por cómo definimos g . Ahora tomemos cualquier par de vértices $u, v \in V(G)$, tal que $\{u, v\} \in E(G)$. Luego, como f es un homomorfismo, $\{f(u), f(v)\} \in E(H)$. Como g es un coloedo, entonces $g(f(u)) \neq g(f(v))$. Luego, el coloedo que creamos, $g \circ f$, es un coloedo válido para G , con $\chi(H)$ colores. Entonces $\chi(G) \leq \chi(H)$.

□

Ejercicio 4.11.5

Dado un grafo dirigido $D = (V, E)$, se lo llama *orientado* cuando para todo par de vértices $v, w \in V$, tenemos que $(v, w) \notin E$, o $(w, v) \notin E$. Puede no estar ninguna de las dos aristas en E . Equivalentemente, G se obtiene de tomar un grafo, y darle una dirección a cada una de sus aristas.

Demostrar que para todo $n \in \mathbb{N}, n \geq 1$, existe un único (salvo isomorfismo) grafo dirigido orientado G con n vértices tal que todos los vértices de G tienen un grado de salida distinto.

◆

Vamos a dar dos demostraciones distintas. La primera es por inducción, que es la manera más sencilla de probar esto. La segunda usa un argumento combinatorio.

Demostración. Dos observaciones:

1. Si G es un tal grafo dirigido, como todos los grados de salida están en $\{0, \dots, n - 1\}$, cuyo tamaño es n , y hay n vértices con n grados de salida distintos, entonces el conjunto de grados de salida es exactamente $\{0, \dots, n - 1\}$.
2. La suma de los grados de salida de todos los vértices es igual al número de aristas, $|E(G)|$. Por el punto anterior, llamemos v_i al único vértice con grado de salida i . Luego, si notamos por $d_G^+(v)$ el grado de salida de un vértice $v \in V(G)$, entonces $|E(G)| = \sum_{v_i \in V(G)} d_G^+(v_i) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. Pero esto es el máximo número de aristas que puede haber en un grafo, teniendo una arista entre cada par de vértices. Luego G resulta de darle una orientación a las aristas de un grafo completo, K_n .

Sea $P(n)$: Existe un único grafo dirigido orientado de n vértices tal que todos los vértices tienen grados de salida distintos. Vamos a probar $\forall n \in \mathbb{N}. (n \geq 1 \Rightarrow P(n))$ por inducción.

- Caso base, $P(1)$. Hay un sólo grafo dirigido orientado con un sólo vértice, salvo isomorfismo, y es $G = (\{0\}, \emptyset)$. El único vértice tiene grado de salida 0, y luego $P(1)$ es cierto.
- Paso inductivo. Tenemos $n \in \mathbb{N}, n \geq 1$. Asumimos $P(n)$, y queremos probar $P(n + 1)$. Para probar existencia, vamos a construir un tal grafo explícitamente. Sea $V = \{0, \dots, n\}$, y $E = \{(i, j) \mid 0 \leq i, j \leq n, i > j\}$, y $G = (V, E)$. Tenemos que G es un grafo dirigido de $|V| = n + 1$ vértices. Notemos por $d_G^+(v)$ el grado de salida de un vértice $v \in V(G)$. Entonces, $d_G^+(i) = i$, pues i es mayor que $0, 1, 2, \dots, i - 1$, y hay i de esos otros números entre 0 y n , es decir en V . Luego, si $i \neq j$, entonces $d_G^+(i) = i \neq j = d_G^+(j)$. Luego, todos los vértices tienen grados de salida distintos.

Ahora tenemos que mostrar unicidad salvo isomorfismo. Sean $G = (V, E), H = (W, F)$ dos grafos dirigidos orientados con $n + 1$ vértices, tal que en cada grafo, todos los vértices tienen grados de salida distintos. Por la primera observación al principio de la demostración, el conjunto de grados de salida de G y H es $\{0, \dots, n\}$. Nombremos entonces, para cada $0 \leq i \leq n$, $v_i \in V(G)$ al vértice en G con grado de salida $d_G^+(v_i) = i$, y $w_i \in V(H)$ al vértice en H con grado de salida $d_H^+(w_i) = i$. Ahora consideremos $G' = G - v_n$, y $H' = H - w_n$. Por la segunda observación, como v_n tiene grado de salida n , y entre todo par de vértices en G

hay exactamente una arista, entonces ningún vértice en G tiene una arista hacia v_n , y luego para todo $v \in V \setminus \{v_n\}$, $d_{G'}^+(v) = d_G^+(v)$. Luego, los grados de salida de los vértices en G' son exactamente $\{0, \dots, n-1\}$, y todos los grados de salida son distintos. Análogamente, los grados de salida de los vértices en H' son exactamente $\{0, \dots, n-1\}$, y todos los grados de salida son distintos.

Por la hipótesis inductiva, existe un isomorfismo $f : G' \rightarrow H'$. Vamos a ver que f se extiende a un isomorfismo entre G y H . Definimos $g : V \rightarrow W$, como $g(v) = f(v)$ para todo $v \in V \setminus \{v_n\}$, y $g(v_n) = w_n$. Veamos que es un isomorfismo. Es claro que g es una función biyectiva, pues f lo era, y estamos agregando un elemento al dominio y al codominio, y asignando el uno al otro. Ahora tenemos que ver que para todo par de vértices $u, v \in V$, $(u, v) \in E \Leftrightarrow (g(u), g(v)) \in F$. Los vértices de V se partitionan en $\{v_n\}$ y $V \setminus \{v_n\}$.

1. Si $u, v \in V \setminus \{v_n\}$, entonces $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in F \Leftrightarrow (g(u), g(v)) \in F$, pues f ya es un isomorfismo entre G' y H' .
2. Si $u = v_n$, y $v \in V \setminus \{v_n\}$, entonces sabemos que $(u, v) \in E$, pues u tiene grado de salida n , y luego tiene una arista hacia todos los otros vértices en G . Por otro lado, $g(u) = w_n$, y $g(v) = f(v) \in V(H')$, con $f(v) \neq w_n$. Como w_n también tiene grado de salida n , entonces $(g(u), g(v)) \in F$. Para probar la vuelta del si y sólo si, no hace falta hacer nada, pues sabemos que la conclusión, $(u, v) \in E$, es cierta.
3. Si $v = v_n$, y $u \in V \setminus \{v_n\}$, entonces sabemos que $(u, v) \notin E$, pues v no tiene aristas entrantes. Para probar la ida del si y sólo si, no hace falta hacer nada, pues falso implica todo. Para probar la vuelta, tenemos que ver que $(g(u), g(v)) \notin F$. Como $g(v) = w_n$, con w_n el vértice en H con grado de salida n , entonces w_n no tiene aristas entrantes, y luego $(g(u), g(v)) \notin F$.

Luego, g es un isomorfismo entre G y H . Esto prueba que G es único salvo isomorfismo. □

Demostración. Para mostrar que existe un único tal grafo dirigido orientado, salvo isomorfismo, debemos mostrar que existe al menos uno, y que ese uno es único. Sea entonces $n \in \mathbb{N}, n \geq 1$. Vamos a construir el grafo dirigido ordenado $G = (V, E)$, donde $V = \{0, \dots, n-1\}$, y $E = \{(i, j) \mid 0 \leq i, j \leq n-1, i > j\}$. Es decir, hay una arista dirigida desde i a j si y sólo si $i > j$.

Claramente, G es un grafo dirigido. Para cada $0 \leq i, j \leq n-1$, o bien $i \leq j$, o bien $j \leq i$, y luego al menos una de (i, j) o (j, i) no está en E , y luego G es un grafo dirigido orientado. Notando $d_G^+(v)$ como el grado de salida de cada vértice $v \in V(G)$, vemos que $d_G^+(i) = i$, pues i es mayor que $0, 1, 2, \dots, i-1$, y hay i de esos otros números entre 0 y $n-1$, es decir en V . Luego, si $i \neq j$, entonces $d_G^+(i) = i \neq j = d_G^+(j)$. Luego, todos los vértices tienen grados de salida distintos.

Para ver que G es único salvo isomorfismo, sea $H = (W, F)$ cualquier tal grafo, con $|H| = n$. Como todos los grados de salida de los vértices de H están en $\{0, \dots, n-1\}$, cuyo tamaño es n , y hay n grados de salida distintos, entonces el conjunto de grados de salida de W es exactamente $\{0, \dots, n-1\}$. Nombremos entonces w_i al vértice en H con grado de salida $d_H^+(w_i) = i$.

Como sabemos, el número de aristas de un grafo dirigido es igual a la suma de los grados de salida de todos los vértices. Luego, $|F| = \sum_{w_i \in V} d_H^+(w_i) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. La suma de los grados de salida es igual a la suma de los grados de entrada, pues toda arista suma 1 a ambas sumatorias, y luego

$$|F| = \sum_{w_i \in V} d_H^-(w_i) = \frac{n(n-1)}{2} \quad (1)$$

donde $d_H^-(w_i)$ es el grado de entrada de w_i . Por otro lado, para cualquier vértice $w \in W$, tenemos $d_H^+(w) + d_H^-(w) \leq n - 1$, pues w puede tener como mucho una arista saliente o entrante a cada uno de los otros $n - 1$ vértices. En nuestro caso, tenemos $d_H^+(w_i) = i$, luego

$$d_H^-(w_i) \leq n - 1 - i \quad (2)$$

Sumando esta desigualdad sobre todos los vértices, y usando la igualdad (1), obtenemos

$$\frac{n(n-1)}{2} = |F| = \sum_{i=0}^{n-1} d_H^-(w_i) \leq \sum_{i=0}^{n-1} n - 1 - i = \frac{n(n-1)}{2}$$

Como esto es cierto, todos los términos de la sumatoria tienen que cumplir su cota superior (2), de otra forma la suma de la izquierda sería menor la suma de la derecha. Luego, la desigualdad (2) es en realidad una igualdad, es decir $d_H^-(w_i) = n - 1 - i$ para todo i . Luego, para todo vértice $w \in W$, tenemos que el número de aristas incidentes a w es $d_H^+(w) + d_H^-(w) = n - 1$. Esto significa que H vino de orientar las aristas en un grafo completo, K_n . Luego, para cada par de vértices $w_i, w_j \in W$, o bien $(w_i, w_j) \in F$, o bien $(w_j, w_i) \in F$.

Consideremos la dirección de estas aristas. Sean w_i, w_j cualquier par de vértices en H , con $i > j$. Vamos a probar que $(w_i, w_j) \in F$. Sea $U = \{w_k \mid k \geq i\}$. Vemos que $|U| = n - i$. Tenemos también que la suma de los grados de salida de los vértices en U es $\sum_{k=i}^{n-1} d_H^+(w_k) = \sum_{k=i}^{n-1} k = (n - 1 + i) \frac{n-i}{2}$. Cada arista que sale de algún vértice en U puede ir a o a U , o a $W \setminus U$. El número de aristas salen de U y van a U es exactamente $|U| \frac{|U|-1}{2} = (n - i) \frac{n-i-1}{2}$, recordando que para cada par de vértices en W , a fortiori en U , hay exactamente una arista entre ellos. El número de aristas que salen de U y van a $W \setminus U$ es, entonces

$$\begin{aligned} & (n - 1 + i) \frac{n - i}{2} - (n - i) \frac{n - i - 1}{2} \\ &= \frac{n - i}{2} (n - 1 + i - n + i + 1) \\ &= \frac{n - i}{2} (2i) \\ &= i(n - i) \end{aligned}$$

El máximo número de aristas que pueden salir de U y entrar en $W \setminus U$ es $|U||W \setminus U| = (n - i)i = i(n - i)$, y como vimos que el número de aristas que salen de U y van a $W \setminus U$ es exactamente $i(n - i)$, entonces todas las aristas posibles entre U y $W \setminus U$ existen. En particular, $(w_i, w_j) \in F$.

Luego, para todo grafo dirigido orientado $H = (W, F)$ con todos los grados de salida distintos, para cualquier par de vértices $u, v \in W$, si $d_H^+(u) > d_H^+(v)$, entonces $(u, v) \in F$, y si $d_H^+(u) < d_H^+(v)$, entonces $(v, u) \in F$.

Luego, el isomorfismo entre G y H es $f : V \rightarrow W$, $f(i) = w_i$. Claramente es biyectiva. Además, para cualquier par de vértices $i, j \in V$, si $(i, j) \in E$, entonces $i > j$, y luego $(f(i), f(j)) = (w_i, w_j) \in F$. De la misma forma, si $(f(i), f(j)) \in F$, entonces $i = d_H^+(w_i) > d_H^+(w_j) = j$, y luego $i > j$, y luego $(i, j) \in E$. Luego, f es un isomorfismo entre G y H . \square

Parte V: Complejidad computacional

TODO: This.

Parte VI: Programación lineal y entera

TODO: This.

Parte VII: Apéndice

1 Demostración del Teorema Maestro

Teorema 12 (Teorema maestro)

Sea $T : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ una función tal que:

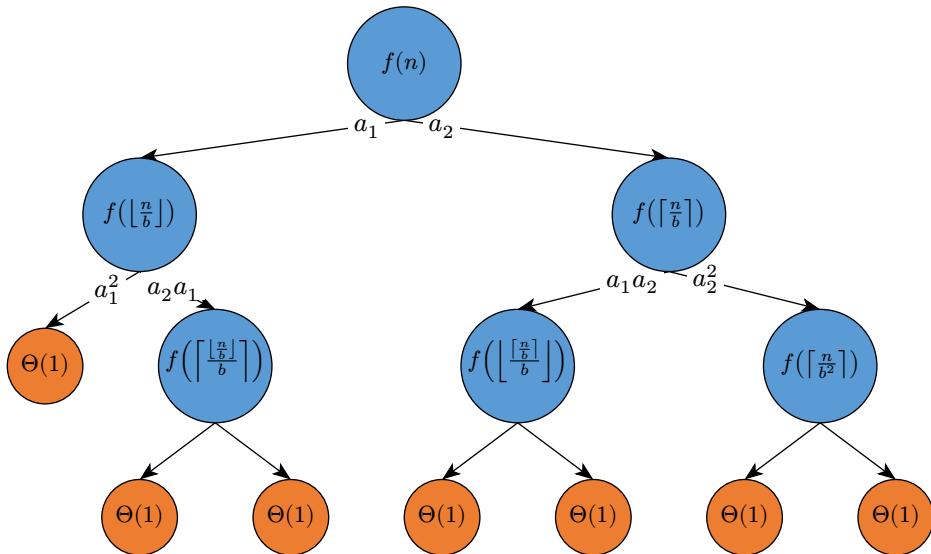
$$T(n) = \begin{cases} b_n & \text{si } 0 \leq n < n_0 \\ a_1 T(\lfloor \frac{n}{b} \rfloor) + a_2 T(\lceil \frac{n}{b} \rceil) + f(n) & \text{si } n \geq n_0 \end{cases}$$

para alguna función $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, algunas constantes $b_0, \dots, b_{n_0-1} \in \mathbb{R}_{\geq 0}$, $a_1, a_2 \in \mathbb{N}$, $b \in \mathbb{N} \geq 2$, $n_0 \in \mathbb{N}$, y asumiendo $n_0 \geq 2$ si $a_2 > 0$ para estar bien definida. Llamemos $a = a_1 + a_2 \geq 1$, y $c = \log_b a$. Entonces:

- Si $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$, entonces $T \in \Theta(n^c)$.
- Si $f \in \Theta(n^c \log^k n)$ para algún $k \in \mathbb{N}$, entonces $T \in \Theta(n^c \log^{k+1} n)$.
- Si $f \in \Omega(n^{c+\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$, y además existen $n_1 \in \mathbb{N}$ y $r < 1 \in \mathbb{R}_{\geq 0}$ tal que $a_1 f(\lfloor \frac{n}{b} \rfloor) + a_2 f(\lceil \frac{n}{b} \rceil) \leq r f(n)$ para todo $n \geq n_1$, entonces $T \in \Theta(f)$.



Demostración.



Suma de valores de vértices internos:

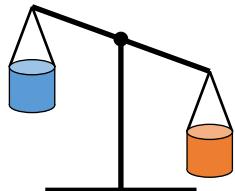
$$\sum_{j < d} a^j f\left(\frac{n}{b^j}\right)$$

con $d = \left\lfloor \log_b \left(\frac{n}{n_0}\right) \right\rfloor + 1$

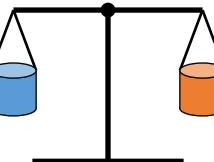
Suma de valores de hojas:

$$\Theta(a^d) = \Theta(n^c)$$

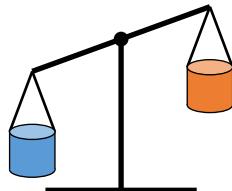
con $c = \log_b a$, y
 $d = \left\lfloor \log_b \left(\frac{n}{n_0}\right) \right\rfloor + 1$



Caso 1: Las hojas dominan.



Caso 2: Las hojas y los vértices internos están balanceados.



Caso 3: Los vértices internos dominan.

Vamos a asumir sin pérdida de generalidad que $n_0 \geq 2$. Si $n_0 = 1$, es decir hay un sólo caso base, podemos definir una función T' , idénticamente definida a T , sólo que T' contiene un caso base extra, $b_{n_0} = T(n_0)$. Como $T(n) = T'(n)$ para todo $n \in \mathbb{N}$, tendrán el mismo comportamiento asintótico, y podemos asumir que estamos analizando T' con $n_0 \geq 2$. Hacemos esto para que expresiones como $\frac{1}{\log_2 n_0}$ estén bien definidas.

Definamos $k_{\min} = \min_{0 \leq i < n_0} b_i$, y $k_{\max} = \max_{0 \leq i < n_0} b_i$.

También vamos a asumir sin pérdida de generalidad que $k_{\min} > 0$. Si todos los casos base son 0, al igual que en el párrafo anterior, tomamos el primer k tal que $T(k) > 0$, y creamos casos base b_{n_0}, \dots, b_k , tal que al menos un caso base no es cero. Si no hay tal primer k con $T(k) > 0$, entonces T es la función constantemente cero, y no hay nada que analizar.

Vamos a analizar el árbol de recursión de $T(n)$. Definimos d como el primer nivel en el que hay una hoja, $d = \left\lfloor \log_b \left(\frac{n}{n_0}\right) \right\rfloor + 1$.

Definimos el *argumento* de un vértice como el valor que se le pasa a la función T en él.

Definimos el *peso* de un vértice en el árbol de recursión de forma inductiva: la raíz tiene peso 1, y si un vértice tiene peso w , su hijo izquierdo (correspondiente a $\lfloor \frac{n}{b} \rfloor$) tiene peso $w \cdot a_1$, y su hijo derecho (correspondiente a $\lceil \frac{n}{b} \rceil$) tiene peso $w \cdot a_2$.

Lema 7.1.1

En el nivel j , los argumentos de los vértices son o bien $\lfloor \frac{n}{b^j} \rfloor$ o bien $\lceil \frac{n}{b^j} \rceil$. En particular, difieren en a lo sumo 1 entre sí. 

*Demuestra*ción. Por inducción sobre j . Para $j = 0$, el único vértice es la raíz con argumento $n = \lfloor \frac{n}{b^0} \rfloor = \lceil \frac{n}{b^0} \rceil$.

Para $j \geq 1$, por hipótesis inductiva los argumentos en el nivel $j - 1$ están en $\{\lfloor \frac{n}{b^{j-1}} \rfloor, \lceil \frac{n}{b^{j-1}} \rceil\}$. Los hijos de estos vértices tienen argumentos de la forma $\lfloor \frac{q}{b} \rfloor$ o $\lceil \frac{q}{b} \rceil$ donde $q \in \{\lfloor \frac{n}{b^{j-1}} \rfloor, \lceil \frac{n}{b^{j-1}} \rceil\}$.

Tenemos que $\left\lfloor \frac{\lfloor \frac{x}{y} \rfloor}{y} \right\rfloor = \lfloor \frac{x}{y^2} \rfloor$, y $\left\lceil \frac{\lceil \frac{x}{y} \rceil}{y} \right\rceil = \lceil \frac{x}{y^2} \rceil$ para todo x, y , con $y \neq 0$. Asimismo, $\lceil x \rceil$ y $\lfloor x \rfloor$ son enteros, y difieren en a lo sumo 1.

El máximo argumento en el nivel j -ésimo, entonces, va a ser $\lceil \frac{n}{b^j} \rceil$, y el mínimo será $\lfloor \frac{n}{b^j} \rfloor$. Como todos los otros argumentos están entre esos dos, y son enteros que difieren en a lo sumo 1, todos los argumentos del nivel j son o bien $\lfloor \frac{n}{b^j} \rfloor$ o bien $\lceil \frac{n}{b^j} \rceil$. 

Lema 7.1.2

Si hay un vértice interno en el nivel d , su argumento es exactamente n_0 . 

*Demuestra*ción. En el nivel d , los argumentos posibles son $\lfloor \frac{n}{b^d} \rfloor$ y $\lceil \frac{n}{b^d} \rceil$, que difieren en a lo sumo 1. Por definición de d , tenemos $\lfloor \frac{n}{b^d} \rfloor < n_0$ (esto es lo que hace que d sea el primer nivel con una hoja).

Para que haya un vértice interno en el nivel d , necesitamos $\lceil \frac{n}{b^d} \rceil \geq n_0$. Como $\lceil \frac{n}{b^d} \rceil - \lfloor \frac{n}{b^d} \rfloor \leq 1$ y $\lfloor \frac{n}{b^d} \rfloor < n_0$, tenemos $\lceil \frac{n}{b^d} \rceil \leq n_0$.

Combinando $\lceil \frac{n}{b^d} \rceil \geq n_0$ y $\lceil \frac{n}{b^d} \rceil \leq n_0$, concluimos que $\lceil \frac{n}{b^d} \rceil = n_0$. 

Por lo tanto, el árbol de recursión tiene la siguiente estructura:

- Niveles $j = 0, 1, \dots, d - 1$: Hay a lo sumo 2^j vértices, todos son internos (i.e. no son hojas). La suma de sus pesos es a^j .
- Nivel $j = d$: Hay a lo sumo 2^d vértices. Algunos son internos, otros hojas. Los internos tienen argumento n_0 .
- Nivel $j = d + 1$: A lo sumo 2^{d+1} vértices, todos son hojas, con valor entre k_{\min} y k_{\max} .

Lema 7.1.3

En cada nivel $j \leq d$, la suma de los pesos de todos los vértices es exactamente a^j .



Demuestra. Por inducción sobre j . Para $j = 0$, la raíz es el único vértice y tiene peso $1 = a^0$.

Para $d \geq j \geq 1$, cada vértice en el nivel $j - 1$ tiene dos hijos cuyos pesos suman $w \cdot a_1 + w \cdot a_2 = w \cdot a$, donde w es el peso del padre. Por lo tanto, la suma de pesos en el nivel j es a veces la suma de pesos en el nivel $j - 1$, que por hipótesis inductiva es a^{j-1} . Luego la suma en el nivel j es $a \cdot a^{j-1} = a^j$.

(Esto vale para $j - 1 \leq d - 1$ pues en esos niveles todos los vértices son internos, y por lo tanto todos tienen exactamente dos hijos, ambos en el nivel j .) \square

Para cada nivel $j < d$, definimos:

$$l_j^- = \min\left(f\left(\left\lfloor \frac{n}{b^j} \right\rfloor\right), f\left(\left\lceil \frac{n}{b^j} \right\rceil\right)\right), \quad l_j^+ = \max\left(f\left(\left\lfloor \frac{n}{b^j} \right\rfloor\right), f\left(\left\lceil \frac{n}{b^j} \right\rceil\right)\right)$$

Pensemos en la descomposición por nivel de nuestro árbol de recursión. En los primeros d niveles, todos los vértices son internos, y por lo tanto la suma de sus pesos es a^j . En los últimos dos niveles, hay a lo sumo a^d peso total de vértices internos, con valor $f(n_0)$, y a lo sumo a^{d+1} peso total de hojas con valor entre k_{\min} y k_{\max} , algunas estando en el nivel d y otras en el nivel $d + 1$.

$$\sum_{j=0}^{d-1} a^j l_j^- + k_{\min} a^d \leq T(n) \leq \sum_{j=0}^{d-1} a^j \cdot l_j^+ + a^d \cdot f(n_0) + k_{\max} \cdot a^{d+1}$$

- Caso 1. $f \in O(n^{c-\varepsilon})$ para algún $\varepsilon > 0$. Queremos probar que $T \in \Theta(n^c)$.

Como $f \in O(n^{c-\varepsilon})$, existen $\alpha > 0$ y $n_1 \in \mathbb{N}$ tales que $f(m) \leq \alpha m^{c-\varepsilon}$ para todo $m \geq n_1$. Definimos $\beta = \max\left(\alpha, \max_{n_0 \leq m \leq n_1} \frac{f(m)}{m^{c-\varepsilon}}\right)$. De esta forma conseguimos que $f(m) \leq \beta m^{c-\varepsilon}$ para todo $m \geq n_0$.

Como en el j -ésimo nivel, los argumentos son o bien $\left\lfloor \frac{n}{b^j} \right\rfloor$ o bien $\left\lceil \frac{n}{b^j} \right\rceil$, llamemos $l_j = \max\left(f\left(\left\lfloor \frac{n}{b^j} \right\rfloor\right), f\left(\left\lceil \frac{n}{b^j} \right\rceil\right)\right)$ al máximo valor de f en el j -ésimo nivel. Por lo tanto, $f\left(\left\lfloor \frac{n}{b^j} \right\rfloor\right) \leq \beta \left\lfloor \frac{n}{b^j} \right\rfloor^{c-\varepsilon}$ y $f\left(\left\lceil \frac{n}{b^j} \right\rceil\right) \leq \beta \left\lceil \frac{n}{b^j} \right\rceil^{c-\varepsilon}$ para todo $j < d$.

Primero probaremos que $T \in O(n^c)$. Tenemos $T(n) \leq \sum_{j=0}^{d-1} a^j \cdot l_j^+ + a^d \cdot f(n_0) + k_{\max} \cdot a^{d+1}$. Para la sumatoria, usamos $\left\lceil \frac{n}{b^j} \right\rceil \leq 2 \frac{n}{b^j}$ y $\left\lfloor \frac{n}{b^j} \right\rfloor \leq \frac{n}{b^j}$:

$$l_j^+ \leq \beta \cdot \left(2 \frac{n}{b^j}\right)^{c-\varepsilon} = \beta \cdot 2^{c-\varepsilon} \cdot \frac{n^{c-\varepsilon}}{b^{j(c-\varepsilon)}}$$

Entonces:

$$\sum_{j=0}^{d-1} a^j \cdot l_j^+ \leq \beta \cdot 2^{c-\varepsilon} \cdot n^{c-\varepsilon} \sum_{j=0}^{d-1} \frac{a^j}{b^{j(c-\varepsilon)}}$$

Como $a = b^c$, $\frac{a^j}{b^{j(c-\varepsilon)}} = \frac{b^{jc}}{b^{j(c-\varepsilon)}} = b^{j\varepsilon}$. Luego:

$$\sum_{j=0}^{d-1} a^j \cdot l_j^+ \leq \beta \cdot 2^{c-\varepsilon} \cdot n^{c-\varepsilon} \sum_{j=0}^{d-1} b^{j\varepsilon}$$

La sumatoria es geométrica con razón $b^\varepsilon > 1$:

$$\sum_{j=0}^{d-1} b^{j\varepsilon} = \frac{b^{d\varepsilon} - 1}{b^\varepsilon - 1} < \frac{b^{d\varepsilon}}{b^\varepsilon - 1}$$

Como $d = \left\lfloor \log_b \left(\frac{n}{n_0} \right) \right\rfloor + 1 \leq \log_b \left(\frac{n}{n_0} \right) + 1$:

$$b^{d\varepsilon} \leq b^{\varepsilon(\log_b(\frac{n}{n_0})+1)} = b^\varepsilon \cdot \left(\frac{n}{n_0} \right)^\varepsilon$$

Por lo tanto:

$$\begin{aligned} \sum_{j=0}^{d-1} a^j \cdot l_j^+ &\leq \beta \cdot 2^{c-\varepsilon} \cdot n^{c-\varepsilon} \cdot \frac{b^\varepsilon \cdot \left(\frac{n}{n_0} \right)^\varepsilon}{b^\varepsilon - 1} \\ &= \frac{\beta \cdot 2^{c-\varepsilon} \cdot b^\varepsilon}{(b^\varepsilon - 1) \cdot n_0^\varepsilon} \cdot n^c \end{aligned}$$

Esto está en $O(n^c)$.

Para los términos restantes:

- $a^d \cdot f(n_0)$: Como $d \leq \log_b \left(\frac{n}{n_0} \right) + 1$, tenemos $a^d \leq a \cdot \left(\frac{n}{n_0} \right)^c \in O(n^c)$.
- $k_{\max} \cdot a^{d+1} = a \cdot k_{\max} \cdot a^d \leq a \cdot k_{\max} \cdot \left(\frac{n}{n_0} \right)^c \in O(n^c)$.

Concluimos que $T \in O(n^c)$. Ahora probemos la cota inferior, $T \in \Omega(n^c)$. Por nuestra descomposición:

$$T(n) \geq \sum_{j=0}^{d-1} a^j \cdot l_j^- + k_{\min} \cdot a^d \geq k_{\min} \cdot a^d$$

ya que $f(x) \geq 0$ para todo $x \in \mathbb{N}$.

Como $d = \left\lfloor \log_b \left(\frac{n}{n_0} \right) \right\rfloor + 1 > \log_b \left(\frac{n}{n_0} \right)$:

$$a^d > a^{\log_b \left(\frac{n}{n_0} \right)} = \left(\frac{n}{n_0} \right)^c = \frac{n^c}{n_0^c}$$

Por lo tanto $T(n) > k_{\min} \cdot \frac{n^c}{n_0^c}$ para todo $n \geq n_0$, y luego $T \in \Omega(n^c)$. Concluimos que $T \in \Theta(n^c)$.

- Caso 2. Sabemos que $f \in \Theta(n^c \log^k n)$ para algún $k \in \mathbb{N}$. Queremos probar que $T \in \Theta(n^c \log^{k+1} n)$.

Como $f \in \Theta(n^c \log^k n)$, existen $\alpha, \beta' > 0$ y $N_0 \in \mathbb{N}$ tales que para todo $m \geq N_0$:

$$\alpha m^c \log^k m \leq f(m) \leq \beta' m^c \log^k m$$

Como en el caso anterior, definimos $\beta = \max(\beta', \max_{n_0 \leq m < \max(N_0, n_0+1)} \frac{f(m)}{m^c \log^k m})$, para saber que $f(m) \leq \beta m^c \log^k m$ para todo $m \geq n_0$. Definimos $N_1 = \max(n_0, N_0, 4)$. Probaremos ambas cotas para $n \geq 4N_1^2$.

Cota superior. De la descomposición del árbol:

$$T(n) \leq \sum_{j=0}^{d-1} a^j \cdot l_j^+ + a^d \cdot (ak_{\max} + f(n_0))$$

Para $j < d$, tenemos $\frac{n}{b^j} \geq n_0 \geq 2$, así que $\lceil \frac{n}{b^j} \rceil \leq \frac{n}{b^j} + 1 \leq 2 \frac{n}{b^j}$. Como ambos argumentos a f en l_j^+ son al menos n_0 , y porque β es tal que para todo $m \geq n_0$ se tiene $f(m) \leq \beta m^c \log^k m$, tenemos:

$$l_j^+ \leq \beta \cdot \left(2 \frac{n}{b^j}\right)^c \cdot \log^k \left(2 \frac{n}{b^j}\right) = \beta \cdot 2^c \cdot \frac{n^c}{b^{jc}} \cdot \log^k \left(2 \frac{n}{b^j}\right)$$

Usando $a^j = b^{jc}$:

$$a^j \cdot l_j^+ \leq \beta \cdot 2^c \cdot n^c \cdot \log^k \left(2 \frac{n}{b^j}\right)$$

Para $n \geq 4$ y $j < d$, tenemos $2 \frac{n}{b^j} \leq 2n$, así que $\log(2 \frac{n}{b^j}) \leq \log(2n) \leq 2 \log n$. Por lo tanto:

$$\sum_{j=0}^{d-1} a^j \cdot l_j^+ \leq \beta \cdot 2^c \cdot n^c \cdot d \cdot (2 \log n)^k = \beta \cdot 2^{c+k} \cdot n^c \cdot d \cdot \log^k n$$

Como $d = \left\lfloor \log_b \left(\frac{n}{n_0} \right) \right\rfloor + 1 \leq \log_b n + 1 \leq \frac{2 \log n}{\log b}$ para $n \geq b$:

$$\sum_{j=0}^{d-1} a^j \cdot l_j^+ \leq \frac{2^{c+k+1} \beta}{\log b} \cdot n^c \log^{k+1} n$$

Para los valores de los últimos dos niveles, como $d \leq \log_b \left(\frac{n}{n_0} \right) + 1$, tenemos $a^d \leq a \cdot \left(\frac{n}{n_0} \right)^c$. Sea $\delta = a \cdot n_0^{-c} \cdot (ak_{\max} + f(n_0))$. Entonces:

$$a^d \cdot (ak_{\max} + f(n_0)) \leq \delta \cdot n^c$$

Como $O(n^c) \subseteq O(n^c \log^{k+1} n)$, estos términos también están en $O(n^c \log^{k+1} n)$. Combinando ambas partes obtenemos $T \in O(n^c \log^{k+1} n)$.

Cota inferior. De la descomposición:

$$T(n) \geq \sum_{j=0}^{d-1} a^j \cdot l_j^-$$

Lema 7.1.4

Para $j \leq \frac{d-1}{2}$ y $n \geq 4N_1^2$, se tiene $\lfloor \frac{n}{b^j} \rfloor \geq \frac{n}{2b^j} \geq N_1$.

Demostración. Como $d - 1 \leq \log_b\left(\frac{n}{n_0}\right)$, para $j \leq \frac{d-1}{2}$ tenemos $b^j \leq b^{\frac{d-1}{2}} \leq \sqrt{\frac{n}{n_0}}$. Por lo tanto:

$$\frac{n}{b^j} \geq \frac{n}{\sqrt{\frac{n}{n_0}}} = \sqrt{n \cdot n_0}.$$

Como $n_0 \geq 2$, se tiene $\sqrt{n \cdot n_0} \geq \sqrt{n}$. Y como $n \geq 4N_1^2$, resulta $\sqrt{n} \geq 2N_1 \geq 8 \geq 2$. Luego $\frac{n}{b^j} \geq 2$, y por tanto:

$$\left\lfloor \frac{n}{b^j} \right\rfloor \geq \frac{n}{b^j} - 1 \geq \frac{n}{2b^j}.$$

Además:

$$\frac{n}{2b^j} \geq \frac{\sqrt{n \cdot n_0}}{2}.$$

Como $n \geq 4N_1^2$ y $n_0 \geq 2$, se cumple $n \cdot n_0 \geq 8N_1^2 \geq 4N_1^2$, y luego:

$$\frac{\sqrt{n \cdot n_0}}{2} \geq \frac{\sqrt{4N_1^2}}{2} = N_1.$$

Por lo tanto, $\left\lfloor \frac{n}{b^j} \right\rfloor \geq \frac{n}{2b^j} \geq N_1$. □

Para tales j , como $\left\lfloor \frac{n}{b^j} \right\rfloor \geq N_1 \geq N_0$:

$$l_j^- \geq \alpha \cdot \left\lfloor \frac{n}{b^j} \right\rfloor^c \cdot \log^k \left(\left\lfloor \frac{n}{b^j} \right\rfloor \right) \geq \alpha \cdot \left(\frac{n}{2b^j} \right)^c \cdot \log^k \left(\frac{n}{2b^j} \right)$$

Usando $a^j = b^{jc}$:

$$a^j \cdot l_j^- \geq \frac{\alpha}{2^c} \cdot n^c \cdot \log^k \left(\frac{n}{2b^j} \right)$$

Ahora acotemos $\log \left(\frac{n}{2b^j} \right)$ con el siguiente lemma.

Lema 7.1.5

Para $j \leq \frac{d-1}{2}$ y $n \geq 4N_1^2$, se tiene $\log \left(\frac{n}{2b^j} \right) \geq \frac{1}{4} \log n$.



Demostración. Del lema anterior, $\frac{n}{2b^j} \geq \frac{\sqrt{n \cdot n_0}}{2} \geq \frac{\sqrt{n}}{2}$.

$$\log \left(\frac{n}{2b^j} \right) \geq \log \left(\frac{\sqrt{n}}{2} \right) = \frac{1}{2} \log n - \log 2$$

Para que esto sea al menos $\frac{1}{4} \log n$, necesitamos $\frac{1}{2} \log n - \log 2 \geq \frac{1}{4} \log n$, es decir, $n \geq 16$.

Como $n \geq 4N_1^2 \geq 4 \cdot 16 = 64 \geq 16$, la desigualdad se cumple. □

Por lo tanto, para $j \leq \frac{d-1}{2}$:

$$a^j \cdot l_j^- \geq \frac{\alpha}{2^c} \cdot n^c \cdot \left(\log \frac{n}{4}\right)^k = \frac{\alpha}{2^c \cdot 4^k} \cdot n^c \log^k n$$

El número de tales términos es $\lfloor \frac{d-1}{2} \rfloor + 1 \geq \frac{d}{2}$. Como $d > \log_b \left(\frac{n}{n_0} \right) \geq \frac{\log n}{2 \log b}$ para $n \geq n_0^2$:

$$\sum_{j=0}^{\lfloor \frac{d-1}{2} \rfloor} a^j \cdot l_j^- \geq \frac{d}{2} \cdot \frac{\alpha}{2^c \cdot 4^k} \cdot n^c \log^k n \geq \frac{\alpha}{2^{c+2k+2} \log b} \cdot n^c \log^{k+1} n$$

Así $T \in \Omega(n^c \log^{k+1} n)$.

Combinando ambas cotas: $T \in \Theta(n^c \log^{k+1} n)$.

- Caso 3. Supongamos que $f \in \Omega(n^{c+\varepsilon})$ para algún $\varepsilon \in \mathbb{R}_{>0}$, y que además existen $n_1 \in \mathbb{N}$ y $r < 1 \in \mathbb{R}_{\geq 0}$ tales que para todo $n \geq n_1$ se cumple $a_1 f(\lfloor \frac{n}{b} \rfloor) + a_2 f(\lceil \frac{n}{b} \rceil) \leq r f(n)$.

Por la hipótesis $f \in \Omega(n^{c+\varepsilon})$, existen $\alpha > 0$ y $n_2 \in \mathbb{N}$ tales que para todo $n \geq n_2$ se cumple $f(n) \geq \alpha n^{c+\varepsilon} > 0$. Definimos $N = \max(n_0, n_1, n_2)$.

Vamos a probar que $T \in \Theta(f)$. Primero, la cota inferior es inmediata: para todo $n \geq n_0$, $T(n) = a_1 T(\lfloor \frac{n}{b} \rfloor) + a_2 T(\lceil \frac{n}{b} \rceil) + f(n) \geq f(n)$, luego $T \in \Omega(f)$.

Para la cota superior, vamos a definir una constante explícita. Consideremos el conjunto finito $S = \{n \in \mathbb{N} : N \leq n \leq bN\}$. Como $f(n) > 0$ para todo $n \in S$, definimos $C_0 = \max_{n \in S} \left(\frac{T(n)}{f(n)} \right)$. Definimos finalmente $C = \max(C_0, \frac{1}{1-r})$.

Vamos a probar por inducción que para todo $n \geq N$ se cumple $T(n) \leq Cf(n)$.

- Caso base. Si $N \leq n \leq bN$, entonces por definición de C_0 tenemos $\frac{T(n)}{f(n)} \leq C_0 \leq C$, y por lo tanto $T(n) \leq Cf(n)$.
- Paso inductivo. Sea $n > bN$, y supongamos que para todo m con $N \leq m < n$ se cumple $T(m) \leq Cf(m)$. Como $b \geq 2$ y $n > bN$, tenemos $\lfloor \frac{n}{b} \rfloor \geq N$ y $\lceil \frac{n}{b} \rceil \geq N$, y además $\lfloor \frac{n}{b} \rfloor < n$ y $\lceil \frac{n}{b} \rceil < n$, por lo que podemos aplicar la hipótesis inductiva.

Entonces:

$$\begin{aligned} T(n) &= a_1 T\left(\lfloor \frac{n}{b} \rfloor\right) + a_2 T\left(\lceil \frac{n}{b} \rceil\right) + f(n) \\ &\leq a_1 Cf\left(\lfloor \frac{n}{b} \rfloor\right) + a_2 Cf\left(\lceil \frac{n}{b} \rceil\right) + f(n) \\ &= C \left(a_1 f\left(\lfloor \frac{n}{b} \rfloor\right) + a_2 f\left(\lceil \frac{n}{b} \rceil\right) \right) + f(n) \\ &\leq Crf(n) + f(n) \quad (\text{por la hipótesis de regularidad}) \\ &= (Cr + 1)f(n). \end{aligned}$$

Como $C \geq \frac{1}{1-r}$, tenemos $Cr + 1 \leq C$, y por lo tanto $T(n) \leq Cf(n)$.

Por inducción, $T(n) \leq Cf(n)$ para todo $n \geq N$, y luego $T \in O(f)$. Concluimos que $T \in \Theta(f)$.

□

2 Demostración del número de comparaciones de Mergesort

Esta demostración no es particularmente larga. La pongo en el apéndice porque quiero que vean el poder del teorema maestro en el capítulo donde se ve este algoritmo por primera vez.

Notemos cómo acá derivamos la función del peor caso explícitamente, no sólo asintóticamente.

Ejercicio 7.2.1

Mergesort es un algoritmo de ordenamiento a base de comparaciones.

```
def merge(left: list[int], right: list[int]) -> list[int]:
    result = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

def mergesort(arr: list[int]) -> list[int]:
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])
    return merge(left, right)
```

Dada una lista de enteros x , sea $C(x)$ el número de comparaciones que hace `mergesort(x)`. Se define $T : \mathbb{N} \rightarrow \mathbb{N}$ como $T(n) = \max_x \{C(x) \mid \text{len}(x) = n\}$. Es decir, el máximo número de comparaciones que realiza `mergesort(x)`, entre todas las listas x tales que `len(x) = n`.

Probar que $T(n) = n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$.



Demostración. Llamemos $M(l, r)$ al número de comparaciones que hace `merge` al ser llamados con dos listas de longitud l y r respectivamente. Probemos algo sobre el comportamiento de `merge`.

Lema 7.2.2

$M(l, r) \leq l + r - 1$.



Demostración. En cada iteración del `while`, se hace una comparación entre `left[i]` y `right[j]`, y se incrementa exactamente uno de i o j . Luego, después de t iteraciones, $i + j = t$. El ciclo termina cuando $i = l$ o $j = r$. Si termina con $i = l$, entonces $j < r$

(pues la condición del `while` requiere ambas desigualdades estrictas para entrar al ciclo).
Luego $t = l + j \leq l + r - 1$. El caso $j = r$ es análogo. \square

En `mergesort`, recibimos una lista x de longitud n . La dividimos en dos partes, de tamaño $l = \lfloor \frac{n}{2} \rfloor$ y $r = \lceil \frac{n}{2} \rceil$ respectivamente. Podemos ver por inducción que la longitud de `mergesort(x)` es igual a `len(x)`. Por ende, `merge` recibe dos listas ordenadas de longitudes l y r . Por el lema, `merge` hace a lo sumo $l + r - 1 = n - 1$ comparaciones, y por lo tanto $T(n) \leq T(l) + T(r) + n - 1$.

Para ver que esta cota se alcanza, construimos inductivamente una entrada peor caso. Definimos la siguiente función, que recibe una lista ordenada y devuelve una permutación de ella:

```
def peor_caso(ordenada: list[int]) -> list[int]:
    if len(ordenada) <= 1:
        return ordenada
    izq = ordenada[1::2] # posiciones impares: l elementos
    der = ordenada[0::2] # posiciones pares: r elementos
    return peor_caso(izq) + peor_caso(der)
```

Por ejemplo, `peor_caso([1,2,3,4,5,6,7,8])` devuelve `[4,8,2,6,3,7,1,5]`. Al ejecutar `mergesort` sobre esta lista, cada llamada a `merge` en cada nivel de la recursión recibe dos listas ordenadas cuyos elementos se intercalan, y por lo tanto hace exactamente $n - 1$ comparaciones. La idea es que en cada nivel, `peor_caso` distribuye los valores pares a la mitad izquierda y los impares a la derecha (con respecto a la lista ordenada de ese nivel), de modo que al ordenar cada mitad, los resultados se intercalan.

Probemos por inducción que `mergesort(peor_caso([1, ..., n]))` hace exactamente $T(l) + T(r) + n - 1$ comparaciones. Para $n = 1$, no hay comparaciones. Para $n > 1$: la mitad izquierda es `peor_caso(izq)` donde `izq` tiene l elementos, y por hipótesis inductiva, ordenarla cuesta $T(l)$ comparaciones. Análogamente, la mitad derecha cuesta $T(r)$. Tras ordenar, las mitades producen `izq = [2, 4, ...]` y `der = [1, 3, ...]` (los elementos pares e impares de la lista original), que se intercalan. Luego, `merge` hace exactamente $n - 1$ comparaciones.

Esto nos da $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1$, con $T(1) = 0$.

Vamos a probar por inducción que $T(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$. Formalmente, sea $P(n) : T(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$. Vamos a probar $P(n)$ para todo natural $n \geq 1$.

- Para $n = 1$, $T(1) = 0$, y asimismo $1\lceil \log_2 1 \rceil - 2^{\lceil \log_2 1 \rceil} + 1 = 1\lceil 0 \rceil - 2^0 + 1 = 0$, con lo cual vale $P(1)$.
- Sea $n > 1$ un natural. Asumimos que $P(k)$ vale para todo $k < n$, queremos probar $P(n)$. Sea $k = \lceil \log_2 n \rceil$, con lo cual $2^{k-1} < n \leq 2^k$. Sea $l = \lfloor \frac{n}{2} \rfloor$, y $r = \lceil \frac{n}{2} \rceil$, con $l + r = n$. Sabemos que $T(n) = T(l) + T(r) + n - 1$.

Como $n \leq 2^k$, entonces $r \leq 2^{k-1}$. Como $n > 2^{k-1}$, entonces $r > 2^{k-2}$. Luego, r está en el intervalo $(2^{k-2}, 2^{k-1}]$, y por lo tanto, $\lceil \log_2 r \rceil = k - 1$. Usando la hipótesis inductiva, obtenemos $T(r) = r(k - 1) - 2^{k-1} + 1$.

Para encontrar $T(l)$, partimos en casos sobre n , ya que el valor de $\lceil \log_2 l \rceil$ depende de si l es exactamente una potencia de 2. Recordemos que $2^{k-1} + 1 \leq n \leq 2^k$.

- Si $n = 2^{k-1} + 1$ exactamente, entonces $l = 2^{k-2}$, y $r = 2^{k-2} + 1$. Luego, $\lceil \log_2 l \rceil = k - 2$. Aplicando la hipótesis inductiva a l , tenemos $T(l) = l(k - 2) - 2^{k-2} + 1$. Sumando lo que teníamos, sabemos que:

$$\begin{aligned}
T(n) &= T(l) + T(r) + n - 1 \\
&= l(k - 2) - 2^{k-2} + 1 + r(k - 1) - 2^{k-1} + 1 + n - 1 \\
&\quad \text{Usando que } 2^{k-2} = l, \text{ y luego } 2^{k-1} = 2l, \text{ obtenemos:} \\
&= l(k - 2) - l + 1 + r(k - 1) - 2l + 1 + (l + r) - 1 \\
&= k(l + r) - 2l - l + 1 - r - 2l + 1 + (l + r) - 1 \\
&= k(l + r) - 4l + 1 \\
&= kn - 4 \times 2^{k-2} + 1 \\
&= kn - 2^k + 1
\end{aligned}$$

que es lo que queríamos demostrar.

- Si $2^{k-1} + 1 < n \leq 2^k$, entonces $l > 2^{k-2}$. Luego, $\lceil \log_2 l \rceil = k - 1$. Aplicando la hipótesis inductiva a l , tenemos $T(l) = l(k - 1) - 2^{k-1} + 1$. Sumando lo que teníamos, sabemos que:

$$\begin{aligned}
T(n) &= T(l) + T(r) + n - 1 \\
&= l(k - 1) - 2^{k-1} + 1 + r(k - 1) - 2^{k-1} + 1 + n - 1 \\
&= (l + r)(k - 1) - 2^k + 2 + (l + r) - 1 \\
&= nk - 2^k + 1
\end{aligned}$$

que es lo que queríamos demostrar.

Luego, hemos probado por inducción que para todo $n \geq 1$, $T(n) = n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$.

□

Bibliografía

- [1] Euclides, *Elementos*. 301a. C.
- [2] Gregory H. Moore, *Zermelo's Axiom of Choice: Its Origins, Development, and Influence*, 1.^a ed. Springer, 1982.
- [3] Edsger Wybe Dijkstra, «Hierarchical Ordering of Sequential Processes», *Acta Informatica*, vol. 1, jun. 1971, [En línea]. Disponible en: <https://www.cs.utexas.edu/~EWD/ewd03xx/EWD310.PDF>
- [4] David Quarfoot y Jeffrey Mark Rabin, «Sources of Students' Difficulties with Proof By Contradiction», *International Journal of Research in Undergraduate Mathematics Education*, 2021.
- [5] Bertrand Russell, *Correspondence with Frege*. en Gottlob Frege Philosophical and Mathematical Correspondence. University of Chicago Press, 1901.
- [6] Ivan Niven, *Irrational Numbers*. The Mathematical Association of America, 1985.
- [7] Christopher Strachey, «An impossible program», *The Computer Journal*, vol. 7, p. 313-313, ene. 1965.

- [8] Richard Dedekind, «Ähnliche (deutliche) Abbildung und ähnliche Systeme», en *Gesammelte mathematische Werke*, vol. 3, R. Fricke, E. Noether, y Ö. Ore, Eds., Vieweg, 1932, pp. 447-449.
- [9] Michael Huth y Mark Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2.^a ed. Cambridge University Press, 2004.
- [10] Herbert Bruce Enderton, *A Mathematical Introduction to Logic*, 2.^a ed. Academic Press, 2001.
- [11] Charles Sanders Peirce, «On the Algebra of Logic: A Contribution to the Philosophy of Notation», *American Journal of Mathematics*, vol. 7, 1885.
- [12] Augustus De Morgan, *Formal Logic: or, The Calculus of Inference, Necessary and Probable*. Taylor and Walton, 1847.
- [13] Ronald Lewis Graham, Donald Ervin Knuth, y Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2.^a ed. Addison-Wesley, 1994.
- [14] Stephen Arthur Cook y Robert Allen Reckhow, «Time-bounded random access machines», *Journal of Computer and System Sciences*, vol. 7, n.^o 4, pp. 354-375, 1973.
- [15] Alan Mathison Turing, «On Computable Numbers, with an Application to the Entscheidungsproblem», *Proceedings of the London Mathematical Society*, vol. 2, n.^o 42, pp. 230-265, 1936.
- [16] Michael Lawrence Fredman y Dan Edward Willard, «Surpassing the information theoretic bound with fusion trees», *Journal of Computer and System Sciences*, vol. 47, n.^o 3, pp. 424-436, 1993.
- [17] Ashok Aggarwal y Jeffrey Scott Vitter, «The input/output complexity of sorting and related problems», *Communications of the ACM*, vol. 31, n.^o 9, pp. 1116-1127, 1988.
- [18] Calvin Creston Elgot y Abraham Robinson, «Random-Access Stored-Program Machines, an Approach to Programming Languages», *Journal of the ACM*, vol. 11, n.^o 4, pp. 365-399, 1964.
- [19] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, y Clifford Stein, *Introduction to Algorithms*, 3.^a ed. The MIT Press, 2009.
- [20] Robert Endre Tarjan, «Amortized Computational Complexity», *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, n.^o 2, pp. 306-318, 1985.
- [21] Franz Mertens, «Ein Beitrag zur analytischen Zahlentheorie», *Journal für die reine und angewandte Mathematik*, vol. 78, pp. 46-62, 1874.
- [22] William Kuszmaul y Charles Eric Leiserson, «Floors and Ceilings in Divide-and-Conquer Recurrences», *Symposium on Simplicity in Algorithms*, pp. 133-141, 2021, doi: 10.1137/1.9781611976496.15.
- [23] George Bernard Dantzig, «All shortest routes in a graph», Operations Research House, Stanford University, nov. 1966.
- [24] Václav Chvátal, «On Hamilton's ideals», *Journal of Combinatorial Theory, Series B*, vol. 12, pp. 163-168, 1972.