Fuentes de imágenes utilizadas: las siguientes ilustraciones fueron encontradas con Google Image Search con la opción de derechos de reuso no comercial con modificaciones. Las imágenes no mencionadas en este listado fueron creadas por mí.

Figura 1.1 https://upload.wikimedia.org/wikipedia/commons/c/cd/Venn_diagram_ABC_BW.png

Figura 1.2 https://upload.wikimedia.org/wikipedia/commons/9/91/BDD.png

Figura 1.3 https://upload.wikimedia.org/wikipedia/commons/thumb/1/14/BDD_simple.svg/378px-BDD_simple.svg.png

Figura 2.1 https://upload.wikimedia.org/wikipedia/commons/thumb/4/42/Inclusion-exclusion.svg/440px-Inclusion-exclusion.svg.png

Figura 2.4 https://upload.wikimedia.org/wikipedia/commons/0/0e/Pascals_triangle_30_lines.png

Figura 2.5 https://cdn.pixabay.com/photo/2016/08/17/18/05/fibonacci-1601158_960_720.png

Figura 2.6 https://upload.wikimedia.org/wikipedia/commons/a/a1/Alan_Turing_Aged_16.jpg

Figura 3.14 https://upload.wikimedia.org/wikipedia/commons/thumb/d/d1/Pila.svg/400px-Pila.svg.png

Figura 3.15 https://upload.wikimedia.org/wikipedia/commons/thumb/b/bb/Cola.svg/400px-Cola.svg.png

Índice general

Índice de cuadros

Índice de figuras

1 Lógica

Para comenzar, es necesario aprender operaciones básicas de **aritmética entera** en Python (es decir, operaciones que toman como entrada números enteros y producen como salida números enteros): suma, resta, producto, potencia, división entera, residuo (módulo, discutido en la sección 3.4 de Graham et al. [1994]). Prueba siempre todas las instrucciones de Python de los ejemplos hasta que estés seguro de entender exactamente qué es el efecto de cada instrucción.

```
3 + 5 # sumar
9 - 3 # restar
4 * 6 # multiplicar
2 ** 3 # dos elevado a la potencia tres
a = 1234567 # guardamos un valor en una variable
b = 17 # guardamos otro valor
a // b # cuantas veces b cabe en a
a % b # cuanto es el residuo
a % b == 0 # si a es divisible entre b
```

También existe la aritmética donde los todos operandos y/o resultados *no* son enteros.

```
3.7 + 5
9 - 3.6
4 * 2.6
2.4 ** 0.4
8 / 3
```

Además de números, en Python se puede manipular **cadenas** (de texto):

- Definición con comillas (simples o dobles, no importa).
- Concatenación con `+`.
- Repetición con `*`.
- Caracteres con acento *no* causan problemas en Python (con que la codificación del archivo sea correcta en el caso de guardarlo en un archivo para uso futuro).

```
'hola'
'hola'
'hola' + 'mundo'
'hola' * 3
```

Valores se pueden guardar en **variables**:

```
a = 3
b = 'prueba'
c = 5.434
```

Los valores guardados se pueden *imprimir*:

```
print(a)
print(a, b)
nombre = 'elisa'
print('hola, {:s}'.format(nombre))
from math import pi
pi
print('pi vale {:f}'.format(pi))
```

Aquí `math` es una librería de rutinas matemáticas adicionales, de la cual se está importando al programa una constante llamada `pi`. Lo de `:s` es un "separador de lugar" para imprimir en esa posición el valor de una cadena, mientras `:f` es para un valor con lugares decimales (no entero) y `:d` sería para un entero.

1.1. Notación matemática

Hay varios símbolos cuyo significado se debe conocer para poder interpretar adecuadamente ecuaciones y definiciones en cualquier campo de las matemáticas. Primeramente, la notación $x \Rightarrow y$ significa que si aplica x, también aplica y; a esto se le llama "implicación" y se profundiza la discusión más adelante en este curso sobre su significado formal.

Notaciones tipo $(a, b), (a, b], [a, b), [a, b]$ refieren a *intervalos* (discretos o continuos); un paréntesis implica que el valor extremo no pertenece al intervalo, mientras un corchete indica que también pertenece.

1.1.1. Fundamentos de conjuntos

Antes que nada, hay que aclarar la notación y terminología referente a *conjuntos*. Conjuntos son el tema del capítulo 3 de Grimaldi [2017]. Los conjuntos frecuentemente se representan visualmente con *diagramas de Venn*, tratados en la sección 3.3 de Grimaldi [2017].

- $A = \{a, b, c\}$ es un conjunto con tres elementos.
- Se escribe $\sum_{i \in A}$ para la **suma** sobre todos los elementos de un conjunto A.
- Se escribe $\prod_{i \in A}$ para el **producto** sobre todos los elementos de un conjunto A.
- $|A|$ es la *cardinalidad* (es decir, el tamaño).
- Se escribe $\in$ para indicar pertenencia de un elemento a un conjunto: $a \in A$.
- $a \notin A$ significa que a no pertenece a A.

Si tienes dudas sobre el comportamiento de sumas y productos, repasa leyendo la sección 1.1 de Grimaldi [2017].

```
A = {'a', 'b', 'c'} # un conjunto
len(A) # su cardinalidad
'a' in A # comprobar pertenencia
'd' in A
'b' not in A # comprobar que no pertenezca
```

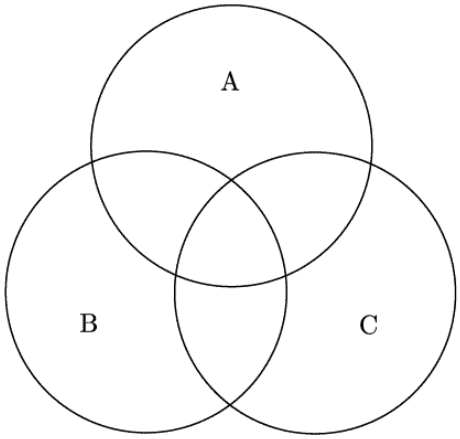


Figura 1.1: Un diagrama de Venn.

```
'e' not in A
```

Por ejemplo, $\mathbb{Z}$ (enteros) y $\mathbb{R}$ (reales) son conjuntos de cardinalidad infinita, mientras $\emptyset$ es el conjunto vacío que no contiene ningún elemento.

- $A \cup B$ es la *unión* de dos conjuntos.
- $A \cap B$ es la *intersección* de dos conjuntos.
- $A \setminus B = \{a \mid a \in A, a \notin B\}$ es la *diferencia*.

```
A = {1, 2, 3}
B = {'x', 'y', 'z'}
A.union(B)
A | B # lo mismo que union()
C = {1, 3, 5}
A.intersection(C)
A & C # lo mismo que intersection()
A.difference(C)
A - C # lo mismo que difference()
```

Nota que necesariamente aplica que $|A \cup B| \geq \max\{|A|, |B|\}$ igual como $|A \cap B| \leq \min\{|A|, |B|\}$.

La notación $B \subseteq A$ indica que B es un *subconjunto* de A, mientras se escribe $C \nsubseteq A$ para indicar que C contiene algo que no esté en A. Nota que necesariamente $B \subset A \Rightarrow |B| < |A|$.

```
A = {0, 1, 2, 3, 4, 5, 6, 7, 8}
B = set(range(9)) # un conjunto desde 0 hasta 8
A == B
C = {1, 3, 5}
C.issubset(A) # si C es un subconjunto de A
```

```
B.issuperset(C) # si C es un subconjunto de B
```

El símbolo $\exists$ refiere a la cuantificación **existencial** (es decir, algo aplica para por lo menos un elemento), mientras el símbolo $\forall$ refiere a la cuantificación **universal** (que aplica para todos).

```
A = {1, 3, 5, 7}
all(a % 2 == 1 for a in A) # si todos son impares
any(a % 5 == 0 for a in A) # si por lo menos uno es divisible
```

Por lo general se supone que existe un *universo* de posibles elementos de los cuales los conjuntos están formados. En este caso, $\bar{A} = \{a \mid a \notin A\}$ refiere al *complemento* de A.

1.2. Representación digital

El **bit** es la unidad básica de información digital: tiene dos valores posibles que se interpreta como los valores lógicos " verdad" (1) y "falso" (0). La cantidad b de bits requeridos para representar un valor $x \in \mathbb{Z}^+$ es el exponente de la mínima potencia de dos que es *mayor* a x:

$$b = \min_{k \in \mathbb{Z}} \left\{ k \mid 2^k > x \right\}. \tag{1.1}$$

Esto quiere decir que cada entero positivo es expresable como la *suma* de ciertas potencias de dos de tal manera que cada potencia aparece una vez (1) o ninguna (0); las potencias se ordenan de menor (a la derecha) a mayor (a la izquierda). A esta representación de los enteros se llama el sistema **binario**:

$$\forall i \in \mathbb{Z}, \exists \mathbf{b} \in \{0,1\}^* : i = \sum_{j=0}^{\infty} b_j \times 2^j. \tag{1.2}$$

```
for potencia in range(37):
    print(2**potencia)
```

Nota la forma en que se realiza la repetición con las instrucciones `for` y `range`.

1.2.1. Unidades de información

El **byte** es la unidad básica de *capacidad* de memoria digital: es una sucesión de ocho bits, por lo cual el número entero más grande que se puede guardar en un solo byte es $2^8 - 1 = 255$. Un *kilobyte* es 1,024 bytes, un *megabyte* es 1,024 kilobytes (1,048,576 bytes) y un *gigabyte* es 1,024 megabytes (1,073,741,824 bytes). Normalmente el prefijo "kilo" implica un mil, pero como mil no es ningún potencia de dos, eligieron la potencia más cercana, $2^{10} = 1{,}024$, para corresponder a los prefijos.

Cualquier entero positivo podría servir como *base* a una representación, no solamente el dos. La computación usa mucho el dos por la facilidad de la representación binaria con fenómenos físicos (apagado versus encendido; alto versus bajo). Dado un entero positivo k, su **conjunto residual** consiste en todos los enteros desde cero hasta $k - 1$,

$$\mathbb{Z}_k = \{0, 1, 2, \ldots, k-2, k-1\}, \tag{1.3}$$

Cuadro 1.1: Algunas potencias de dos.

j	2^j	j	2^j	j	2^j
0	1	14	16,384	28	268,435,456
1	2	15	32,768	29	536,870,912
2	4	16	65,536	30	1,073,471,824
3	8	17	131,072	31	2,147,483,648
4	16	18	262,144	32	4,294,967,296
5	32	19	524,288	33	8,589,934,592
6	64	20	1,048,576	34	17,179,869,184
7	128	21	2,097,152	35	34,359,738,368
8	256	22	4,192,304	36	68,719,476,736
9	512	23	8,388,608	⋮	⋮
10	1,024	24	16,777,216		
11	2,048	25	33,554,432		
12	4,096	26	67,108,864		
13	8,192	27	134,217,728		

es decir, todos los posibles resultados del residuo cuando se divide a cualquier entero positivo entre k. Por ejemplo, $\mathbb{Z}_2 = \{0, 1\}$.

Sistema ternario $\forall i \in \mathbb{Z}, \exists \mathbf{b} \in \mathbb{Z}_3^* : i = \sum_{j=0}^{\infty} b_j \times 3^j$

Sistema octal $\forall i \in \mathbb{Z}, \exists \mathbf{b} \in \mathbb{Z}_8^* : i = \sum_{j=0}^{\infty} b_j \times 8^j$

Corresponde a grupos de *tres* bits.

Sistema decimal $\forall i \in \mathbb{Z}, \exists \mathbf{b} \in \mathbb{Z}_{10}^* : i = \sum_{j=0}^{\infty} b_j \times 10^j$

Sistema hexadecimal $\forall i \in \mathbb{Z}, \exists \mathbf{b} \in \mathbb{Z}_{16}^* : i = \sum_{j=0}^{\infty} b_j \times 16^j$

Corresponde a grupos de *cuatro* bits; $a = 10$, $b = 11$, ..., $f = 15$.

Base arbitraria $k \in \mathbb{Z}, k > 1, \forall i \in \mathbb{Z}, \exists \mathbf{b} \in \mathbb{Z}_k^* : i = \sum_{j=0}^{\infty} b_j \times k^j$

```
x = 250
bin(x) # binario
oct(x) # octal
hex(x) # hexadecimal
```

Para representar números reales por computadora, hay que definir hasta que exactitud se guarda los decimales del número. *Punto flotante* es la representación se adapta al orden de magnitud del valor $x \in \mathbb{R}$ por trasladar la coma decimal hacia la posición de la primera cifra significativa de x mediante un exponente γ, $x = m \cdot b^\gamma$, donde m se llama la *mantisa* y contiene los dígitos significativos de x. El parámetro b es la *base* del sistema de represen-

tación, mientras $\gamma \in \mathbb{Z}$ determina el rango de valores posibles (por la cantidad de memoria que tiene reservada).

Hay que tener mucho cuidado con la operación de parte entera en programación, como implica pérdida de datos. Por ejemplo, considera lo siguiente:

```
a = 0.6 / 0.2
b = int(a)
print(b)
```

Esto suele resultar en `b` asignada al valor 2 y no 3, porque por la representación binaria de punto flotante de 0.6 y 0.2, su división resulta en 2.9999999999999996.

1.3. Lógica booleana

La lógica es el tema del capítulo 2 de Grimaldi [2017]. El momento que necesites ejemplos detallados, checa el libro.

Los **valores de verdad** son dos: verdad $\top$ y falso $\bot$, frecuentemente representados como uno y cero, respectivamente (como en el sistema binario). Variables booleanas pueden tomar estos valores. Por ejemplo, comparaciones de números producen valores de verdad:

```
3 < 5
8 / 2 == 4
7 <= 7
a = (8 > 2)
9 != 6
```

Se tiene un conjunto $X = \{x_1, x_2, \ldots\}$ de *variables* (también: llamados átomos). Cada variable puede tomar el valor "verdad" $\top$ o "falso" $\bot$. La *negación* de una variable x_i. $\neg x_i = \top$ si $x_i = \bot$, mientras $\neg x_i = \bot$ si $x_i = \top$.

Se forman *proposiciones* combinando los valores de verdad y/o variables booleanas con conectivos. Expresiones básicas son los *literales* x_i y $\neg x_i$. Se forman más expresiones utilizando los *conectivos* $\vee$ ("o"), $\wedge$ ("y") — también $\neg$ se considera un conectivo. Si ϕ_1 y ϕ_2 son expresiones booleanas, también $(\phi_1 \vee \phi_2)$, $(\phi_1 \wedge \phi_2)$ y $\neg\phi_1$ lo son.

```
b = not (9 == 6)
c = a and b
d = a or b
```

Los **operadores lógicos** básicos son "no" con "y" o "o". Con la negación unaria y uno de los dos operadores binarios, se pueden definir el resto. Los operadores lógicos derivados incluyen los siguientes:

- $\oplus$ = XOR (o exclusivo),
- $\rightarrow$ = implicación y
- $\leftrightarrow$ = equivalencia.
- $\varphi_1 \oplus \varphi_s$ significa $(\varphi_1 \vee \varphi_2) \wedge \neg(\varphi_1 \wedge \varphi_2)$
- $\bigvee_{i=1}^{n} \varphi_i$ significa $\varphi_1 \vee \cdots \vee \varphi_n$

Cuadro 1.2: Algunas tablas de verdad.

a	b	$a \wedge b$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

a	b	$a \vee b$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$

a	b	$a \oplus b$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$

- $\bigwedge_{i=1}^{n} \varphi_i$ significa $\varphi_1 \wedge \cdots \wedge \varphi_n$¡/tr¿$\phi_1 \to \phi_2$ significa $\neg\phi_1 \vee \phi_2$
- $\phi_1 \leftrightarrow \phi_2$ significa $(\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1)$

Se pueden definir subrutinas propias en Python para estos tres:

```
def xor(a, b):
    return (a or b) and not (a and b)

xor(True, False)
xor(True, True)

def impl(a, b):
    return (not a) or b

impl(True, False)
impl(False, False)
impl(True, True)

def equi(a, b):
    return impl(a, b) and impl(b, a)

equi(True, False)
equi(False, False)
```

Para evaluar si una expresión es verdadera o falsa, se procede a evaluar todas las posibles combinaciones de valores a sus variables de forma sistemática en un cuadro. A este cuadro se le dice "tabla de verdad". Si la expresión con tiene n variables, esta tabla tiene 2^n renglones. Conviene crear columnas auxiliares para subexpresiones cuando la expresión contiene múltiples conectivos.

Un posible orden de *precedencia* de operadores lógicos es el siguiente (aunque no viene escrita en piedra más allá de que la negación sea la más fuerte — en electrónica es común que $\wedge$ sea más fuerte que $\vee$ y no hay ningún acuerdo global sobre la posición del $\oplus$) de la más fuerte al más débil: $\neg$, $\vee$, $\wedge$, $\to$, $\leftrightarrow$. Por ejemplo

$$\neg x_1 \vee x_2 \to x_3 \leftrightarrow \neg x_4 \wedge x_1 \vee x_3 \tag{1.4}$$

debería ser interpretada bajo este orden de precedencia como

$$((((\neg x_1) \vee x_2) \to x_3) \leftrightarrow ((\neg x_4) \wedge (x_1 \vee x_3))). \tag{1.5}$$

Se recomienda siempre utilizar paréntesis para evitar confusión, y al no ser posible, especificar de manera explícita la precedencia que no utiliza.

Inferencia refiere a derivar expresiones verdaderas a partir de un conjunto de expresiones que se conocen ser verdaderas. Las reglas básicas de inferencia incluyen la *modus ponens*

$$(a \wedge (a \to b)) \Rightarrow b, \tag{1.6}$$

modus tollens

$$((a \to b) \wedge \neg b) \Rightarrow \neg a \tag{1.7}$$

y el *silogismo*

$$((a \to b) \wedge (b \to c)) \Rightarrow (a \to c). \tag{1.8}$$

Denotemos por $X(\phi)$ el conjunto de variables booleanas que aparezcan en una expresión ϕ. Una **asignación de valores de verdad** $T : X' \to \{\top, \bot\}$ se dice *adecuada* para ϕ si $X(\phi) \subseteq X'$ (es decir, cada variable recibe un valor). Escribamos $x_i \in T$ si $T(x_i) = \top$ y $x_i \notin T$ si $T(x_i) = \bot$.

Si T *satisface* a ϕ, se escribe $T \models \phi$.

- Si $\phi \in X'$, $T \models \phi$ si y sólo si $T(\phi) = \top$.
- Si $\phi = \neg\phi'$, $T \models \phi$ si y sólo si $T \not\models \phi'$.
- Si $\phi = \phi_1 \wedge \phi_2$, $T \models \phi$ si y sólo si $T \models \phi_1$ y $T \models \phi_2$.
- Si $\phi = \phi_1 \vee \phi_2$, $T \models \phi$ si y sólo si $T \models \phi_1$ o $T \models \phi_2$.

A una expresión booleana ϕ se dice *satisfactible* si existe una T adecuada para ϕ tal que $T \models \phi$. A una expresión booleana ϕ se dice *válida* si $\forall T : T \models \phi$; al ser así, la expresión es una *tautología* y se escribe simplemente $\models \phi$. En general aplica que $\models \phi$ si y sólo si $\neg\phi$ es *no satisfactible*.

Dos expresiones ϕ_1 and ϕ_2 son *lógicamente equivalentes* si para toda asignación T que es adecuada para las dos expresiones aplica que $T \models \phi_1$ si y sólo si $T \models \phi_2$. La equivalencia lógica se denota por $\phi_1 \equiv \phi_2$.

La **forma normal conjuntiva** CNF utiliza puramente el conectivo $\wedge$ y literales. La **forma normal disyunctiva** DNF utiliza puramente el conectivo $\vee$ y literales. Una conjunción de literales se llama un *implicante* y una disyunción de literales se llama una *cláusula*. Supongamos que ninguna cláusula ni implicante sea repetido en una forma normal, y tampoco se repiten literales dentro de las cláusulas o los implicantes. Las expresiones en forma normal pueden en el peor caso tener un *largo exponencial* en comparación con el largo de la expresión original.

Transformaciones de expresiones lógicas se realizan aplicando reglas de equivalencia:

- Reemplazar $\phi_1 \leftrightarrow \phi_2$ con $(\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_1 \vee \phi_2)$.
- Reemplazar $\phi_1 \to \phi_2$ con $\neg\phi_1 \vee \phi_2$.
- Mover los $\neg$ a las variables para formar literales:
  - $\neg\neg\phi$ se reemplaza con ϕ,
  - $\neg(\phi_1 \vee \phi_2)$ se reemplaza con $\neg\phi_1 \wedge \neg\phi_2$ y
  - $\neg(\phi_1 \wedge \phi_2)$ se reemplaza con $\neg\phi_1 \vee \neg\phi_2$.

Transformaciones al CNF se logran moviendo los $\wedge$ afuera de los disyunciones:

- $\phi_1 \vee (\phi_2 \wedge \phi_3)$ se reemplaza con $(\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$,
- $(\phi_1 \wedge \phi_2) \vee \phi_3$ se reemplaza con $(\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3)$.

Para transformaciones al DNF, se mueven los $\vee$ afuera de los conjunciones:

- $\phi_1 \wedge (\phi_2 \vee \phi_3)$ se reemplaza con $(\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$,
- $(\phi_1 \vee \phi_2) \wedge \phi_3$ se reemplaza con $(\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3)$.

Una **función booleana** f de n-dimensiones es un mapeo de $\{\top, \bot\}^n$ al conjunto $\{\top, \bot\}$. La negación $\neg$ corresponde a una función unaria $f^\neg : \{\top, \bot\} \to \{\top, \bot\}$, mientras los otros dos conectivos básicos $\vee$ y $\wedge$ (igual como los derivados $\oplus$, $\to$ y $\leftrightarrow$) definen cada uno una función binaria $f : \{\top, \bot\}^2 \to \{\top, \bot\}$.

Cada expresión booleana se puede interpretar como una función booleana con la dimensión $n = |X(\phi)|$. La expresión ϕ expresa una función f si cada n-eada de valores de verdad $\tau = (t_1, \ldots, t_n)$ aplica que

$$f(\tau) = \begin{cases} \top, & \text{si } T \models \phi, \\ \bot, & \text{si } T \not\models \phi, \end{cases} \tag{1.9}$$

donde T es tal que $T(x_i) = t_i$ para todo $i = 1, \ldots, n$.

Definición 1. Problema de satisfiabilidad SAT
Entrada*: una expresión booleana ϕ en CNF.*
Pregunta*: ¿es ϕ satisfactible?*

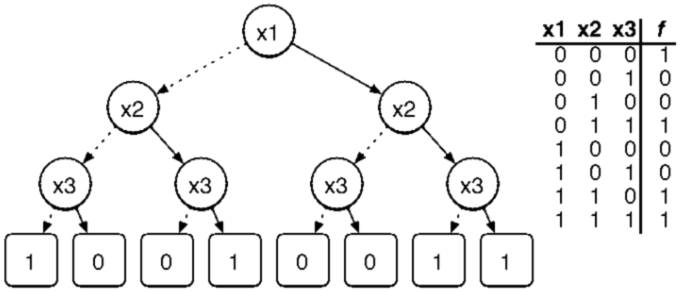
Al SAT se puede resolver utilizando tablas de asignaciones en tiempo $\mathcal{O}(n^2 \cdot 2^n)$. (Más adelante veremos qué quiere decir esa cosa de $\mathcal{O}$.)

Lógica proposicional, llamado "lógica de primer orden", es una lógica donde se cuantifica a variables individuales y las proposiciones tienen que ver con si existe ($\exists$) una variable para la cual una condición estée válida o si para todas ($\forall$) las aplique alguna condición. Los cuantificadores se discuten en la sección 2.4 de Grimaldi [2017]. Nota que $\neg\forall P$ significa "no para todos aplica P" es lo mismo que $\exists\neg P$, es decir, "existe algo para el cual no aplica P".

Diagramas de decisión binarios (inglés: binary decision diagram, BDD) son representaciones de funciones booleanas de tal forma que cada nivel corresponde a una variable y sus posibles valores (verdad y falso) producen dos ramos. Obviamente diferentes ordenamientos de variables producen diferentes árboles.

Se puede simplificar a un BDD eliminando recursivamente estructuras repetidas hasta que ya no quede ninguna.

Operaciones con valores de verdad, cuando se expresan en términos de cero (falso) y uno (verdad), son en realidad *álgebra booleana* (checa el capítulo 15 de Grimaldi [2017] por más detalles). Por lo general $\wedge$ es una multiplicación, $\vee$ es redondear hacía arriba el promedio y $\oplus$ es como sumar en módulo dos.



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figura 1.2: Ejemplo de un BDD.

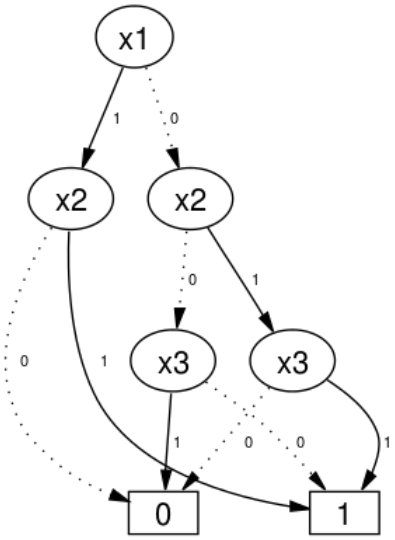


Figura 1.3: El BDD de la figura 1.2 simplificado.

1.3.1. Aritmética binaria

La *aritmética binaria* opera con los bits con operaciones de lógica booleana. Las operaciones son no, y, o, o exclusivo, desplazamiento izquierdo y derecho:

```
a = 1000
b = 7
~a # los ceros a unos y vice versa
a & b # y binario
a | b # o binario
a ^ b # o-exclusivo binario
a << b # agregar b ceros a la derecha
a >> b # quitar b bits a la izquierda
```

1.4. Tarea 1

Las tareas se responden en línea. Hay que usar el mismo usuario para cada tarea y registrar el usuario con la matrícula correspondiente en la página de resultados del semestre actual para recibir puntos por ellas.

1. El resultado de $A \cup (B \setminus (C \cap D))$ con los conjuntos proporcionados en la tarea personalizada.
2. El valor decimal del número binario proporcionado en la tarea personalizada.
3. La representación en una base proporcionada de un valor decimal proporcionado en la tarea personalizada.
4. El resultado de (a & (b << c)) con enteros positivos propocionados en la tarea personalizada.
5. El valor *binario* de la expresión $(x \vee z) \wedge \neg(\neg y \vee \neg z)$ con los valores de verdad proporcionados en la tarea personalizada.

1.4.1. Preguntas de verificación

Discute lo siguiente con los compañeros y con la profesora hasta que esté todo claro. Conviene consultar el libro de texto (capítulos y secciones indicados en el material de la unidad en esta página) y hasta buscar por videos en la web. Cuando ya no cabe duda, procede a la segunda unidad temática.

¿Para qué sirven los conjuntos? ¿Qué relación tienen con consultas en bases de datos? ¿Por qué no importa en que orden están los elementos? ¿Por qué no puede entrar un mismo elemento muchas veces en un conjunto? (Pregunta 1)

¿Para qué sirve la base dos? ¿Por qué se llama binario? ¿Para qué existen otras bases como 16 y 20? ¿Qué relaciones tienen entre ellos las bases 2, 8 y 16? (Preguntas 2 y 3)

¿Por qué se hace lógica con enteros y no solamente con valores de verdad sencillas de un sólo bit? ¿Qué se gana con esto? ¿Para qué se puede aplicar? (Pregunta 4)

¿Qué aplicaciones tiene en la ingeniería la lógica booleana? ¿Para qué lo necesita un mecatrónico? ¿Para qué lo necesita un administrador de sistemas? ¿Para qué lo necesita un ingeniero de software? ¿Qué relación tiene con circuitos digitales? ¿Qué relación tiene con la programación? (Pregunta 5)

2 Combinatoria

La combinatoria estudia maneras de seleccionar y ordenar elementos de conjuntos. Una *permutación* es un ordenamiento de los elementos de un conjunto. Si $|A| = k$, existen $k! = 1 \times 2 \times \ldots \times (k-1) \times k$ permutaciones de los elementos de A. Las permutaciones son el tema de la sección 1.2 de Grimaldi [2017].

La función $k!$ es el factorial:

$$n! = \prod_{k=1}^{n} k \tag{2.1}$$

para enteros positivos (factorial de cero se puede definir por cuestiones de compatibilidad como, aunque en algunos textos prefieren el cero, según para qué propósito se define). La función factorial es el tema de la sección 4.4 de Graham et al. [1994].

```
from math import factorial
factorial(5) # subrutina existente
f = 1
for i in range(1, 20):
    f *= i
print('{:d}! = {:d}'.format(i, f))
```

Por ejemplo, si $A = \{a, b, c\}$, se tiene $3! = 6$ posibles permutaciones: $abc, acb, bac, bca, cab, cba$. Esto es porque hay k maneras de seleccionar el primer elemento, $k-1$ opciones para el segundo, etcétera, hasta que en el último elemento ya no queda más que una sola opción.

Para todo este tipo de cosa, en Python conviene utilizar la librería itertools[1].

```
from itertools import permutations
p = list(permutations(['a','b','c','d']))
len(p)
```

Se puede pensar en una permutación como un mapeo de posiciones originales a las nuevas posiciones. Una *composición* de permutaciones es la aplicación de una después de otra. Estos mapeos se pueden representar con matrices de adyacencia que indiquen cuál elemento va en lugar de cuál y la multiplicación de estas matrices da el efecto de composición de las permutaciones. Permutaciones de n elementos y su composición forman un grupo.

Para seleccionar desde un conjunto A, $|A| = n$, un subconjunto de cardinalidad $0 \leq k \leq n$, hay $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$ maneras de hacer esto: el primer elemento se selecciona libremente entre los n elementos, mientras el segundo tiene $n-1$ opciones, etc., pero al hacer esto hay $k!$ formas de llegar a seleccionar el mismo subconjunto pero en un orden distinto.

A este tipo de conjuntos seleccionados se les llaman *combinaciones* y se discuten en la sección 1.3 de Grimaldi [2017]. Al valor $\binom{n}{k}$ se llama el *coeficiente binomial*. Las coeficientes

[1] https://docs.python.org/dev/library/itertools.html

binomiales son el tema del capítulo 5 de Concrete Mathematics.

```
from math import factorial
def binom(n, k):
    value = 1
    for factor in range(max(k, n - k) + 1, n + 1):
        value *= factor
    return value // factorial(min(k, n - k))

binom(10, 2) == factorial(10) // (factorial(10 - 2) * factorial(2))
from itertools import combinations
A = ['a', 'b', 'c', 'd']
for subconjunto in combinations(A, 3):
    print(subconjunto)
```

Para crear combinaciones de elementos desde un conjunto, pero permitiendo *repeticiones*, la fórmula cambia ya que el elemento seleccionado sigue siendo un candidato para selecciones futuras: $\binom{n+k-1}{k}$.

El *conjunto potencia* 2^A está formado por todos los posibles subconjuntos de A (incluyendo al A mismo y a $\emptyset$). Aplica que $|2^A| = 2^{|A|}$.

```
from itertools import combinations, chain
k = 7
A = range(7)
contador = 0
c = chain.from_iterable(combinations(A, r) for r in range(k + 1))
for subconjunto in c:
    print(subconjunto)
    contador += 1

contador == 2**k
```

2.1. Demostraciones

En las matemáticas, a los hechos universales se les dice *axiomas*. Una *definición* es una manera de fijar el sentido de algún formalismo, notación o terminología. La meta de las demostraciones es derivar de los axiomas y las definiciones, algunos **teoremas**. (Teoremas auxiliares que se obtienen como resultados intermedios al buscar comprobar alguna cosa mayor se llaman *lemas*.) La demostración en sí es una cadena de pasos que establecen que un teorema sea verdad.

Una técnica básica de demostración es la *inducción*, donde primero se establece que una *condición inicial* c_1 es válida y verdadera (el paso base). Luego, el **paso inductivo** consiste en comprobar que si c_k es válida y verdadera, también c_{k+1} lo debe ser. La inducción matemática es el tema del capítulo 4 de Grimaldi [2017].

Por ejemplo, para resolver a

$$T(n) \leq \begin{cases} c, & \text{si } n = 1 \\ g\Big(T(n/2), n\Big), & \text{si } n > 1 \end{cases} \tag{2.2}$$

primero *adivinamos* que la solución sea, en forma general, $T(n) \leq f(a_1, \ldots, a_j, n)$, donde $a_1, \ldots, a_j$ son parámetros de la función f. Para mostrar que para algunos valores de los parámetros $a_1, \ldots, a_j$ aplica $\forall n$ que la solución sea la adivinada, tenemos que demostrar que $c \leq f(a_1, \ldots, a_j, 1)$ y también que si $n > 1$,

$$g\left(f\left(a_1, \ldots, a_j, \frac{n}{2}\right), n\right) \leq f(a_1, \ldots, a_j, n). \tag{2.3}$$

Aplica que $T(k) \leq f(a_1, \ldots, a_j, k)$ para $1 \leq k < n$ por inducción, obteniendo

$$T(n) \leq g\left(T\left(\tfrac{n}{2}\right), n\right) \leq g\left(f\left(a_1, \ldots, a_j, \tfrac{n}{2}\right), n\right) \leq f(a_1, \ldots, a_j, n) \tag{2.4}$$

si $n > 1$.

Ecuaciones de recurrencia como ésta y su solución son el tema de la sección 7.3 de Graham et al. [1994] y del capítulo 10 de Grimaldi [2017].

2.2. Principios y teoremas útiles

El **principio de palomar** (veáse la sección 5.5 de Grimaldi [2017]) establece que si m palomas estén ocupando un total de n nidos y $m > n$, entonces por lo menos un nido contiene dos o más palomas. (Yo sé, suena a sentido común.)

El **principio de inclusión-exclusión** (tema del capítulo 8 de Grimaldi [2017]) es un poco más complejo, pero tampoco es posible de entender: dado un conjunto C de cardinalidad n y un total de t condiciones c_i, el número de elementos en C que no satisfacen *ninguna* condición es

$$\bar{n} = n - \sum_{1 \leq i \leq t} n_{c_i} + \sum_{1 \leq i < j \leq t} n_{c_i c_j} - \sum_{1 \leq i < j < k \leq t} n_{c_i c_j c_k} \ldots + (-1)^t n_{c_1 c_2 \ldots c_t}, \tag{2.5}$$

donde n_S es el número de elementos que satisfacen todas las condiciones incluidos en S. Nota los signos alternantes.

El **teorema de Burnside** (que mucha gente argumenta no ser de Burnside, pero así se conoce) sirve para problemas de conteo de configuraciones posibles. Se explica en términos de ejemplos paso a paso en la sección 16.9 de Grimaldi [2017]. Sea C un conjunto de configuraciones sobre las cuales se opera con un grupo finito de permutaciones $\mathbb{G}$; habrá

$$\frac{1}{|\mathbb{G}|} \sum_{\pi \in \mathbb{G}} \psi(\pi) \tag{2.6}$$

clases de equivalencia a las cuales C se divide por ser actuado con $\mathbb{G}$, donde $\psi(\pi)$ es la cantidad de configuraciones en C que son fijas bajo la permutación π.

Esto es un caso especial del **método de enumeración de Polya** (tratado en la sección 16.11 de Grimaldi [2017]) que se expresa en términos de polinomios que representan posibles

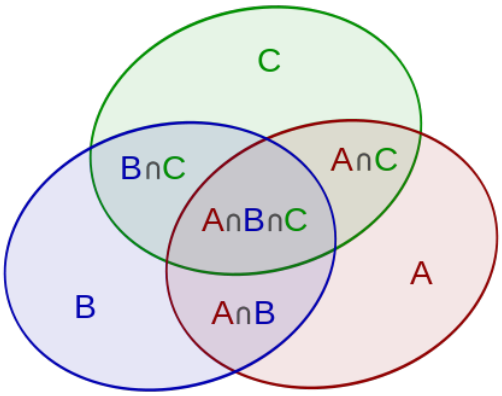


Figura 2.1: Un diagrama de Venn para ilustrar el principio I-E.

patrones. También el capítulo 17 de Grimaldi [2017] sigue en este tipo de temas. Para personas que contemplan un posgrado como una opción futura, se recomienda estudiar estos capítulos también. Para inges comunes y corrientes no es tan crítico saber manipular funciones generadoras y anillos de polinomios, pero por ejemplo en mi campo de estudio son bastante relevantes.

2.3. Sucesiones, secuencias y series

La **sucesión de Fibonacci** se define como

$$\mathcal{F}_k = \begin{cases} 0, & \text{si } k = 0 \\ 1, & \text{si } k = 1 \\ \mathcal{F}_{k-1} + \mathcal{F}_{k-2}, & \text{para } k > 1. \end{cases} \tag{2.7}$$

lo que es una definición *recursiva*: algo se define en términos de si mismo. Para más información sobre definiciones recursivas, checa la sección 4.2 de Grimaldi [2017].

A los elementos de la sucesión $\mathcal{F}$ se les dice los los números Fibonacci; son el tema de la sección 6.6 de Concrete Mathematics.

A los números de Fibonacci aplica que $\mathcal{F}(a) > \frac{\phi^a}{\sqrt{5}} - 1$ donde $\phi = \frac{1+\sqrt{5}}{2}$ es la *tasa dorada* $\approx 1{,}618$.

```
from math import sqrt # subrutina
(1 + sqrt(5)) / 2 # valor de la tasa dorada

def fibo(cuantos):
    listado = [0, 1] # fragmento inicial
    while len(listado) < cuantos: # extendemos al largo deseado
        listado.append(listado[-1] + listado[-2])
     return listado # regresamos resultado

fibo(20) # consultamos la secuencia de veinte elementos
```

En Python, `range(n)` va desde cero hasta $n-1$. Nota que `listado[-1]` accede al último elemento de una secuencia y `listado[-2]` al penúltimo.

Dada una secuencia, es común querer realizar operaciones con ella, como por ejemplo sumas o productos totales o parciales de sus elementos. Se escribe $\sum_{i=a}^{b}$ para la **suma** donde un *índice* i toma valores enteros desde a hasta b en pasos de uno, mientras para la suma parcial hasta el $n$ésimo elemento se escribe

$$S = x_1 + x_2 + x_3 + \ldots + x_n = \sum_{i=1}^{n} x_i. \tag{2.8}$$

Para el producto (multiplicación), la notación es

$$P = x_1 \cdot x_2 \cdot x_3 \cdot \ldots \cdot x_n = \prod_{i=1}^{n} x_i. \tag{2.9}$$

Sumas son el tema del capítulo 2 de Graham et al. [1994]. Les aplica la *ley de distribución*

$$\sum_{k \in K} c \times a_k = c \sum_{k \in K} a_k \tag{2.10}$$

y la *ley de asociación*

$$\sum_{k \in K} (a_k + b_k) = \sum_{k \in K} a_k + \sum_{k \in K} b_k. \tag{2.11}$$

Una suma infinita es una suma sobre una secuencia infinita o un conjunto infinito de elementos. Si el valor de una sumatoria es finita, se dice que esa suma *converge* (si no convegre, diverge). Recuerda que sumas son efectivamente "integrales discretas".

La diferencia entre conjuntos y secuencias es que una secuencia tiene un orden mientras un conjunto no lo tiene. Por ende, en Python las secuencias no son lo mismo que conjuntos, aunque también de puede sumar y multiplicar sobre conjuntos. En secuencias se puede tener elementos repetidos, mientras en conjuntos esto no sucede.

```
secuencia = [1, 4, 4, 7, 8]
conjunto = {1, 4, 7, 8}
sum(secuencia) != sum(conjunto)

def producto(entrada):
    resultado = 1
    for elemento in entrada:
        resultado *= elemento
    return resultado

producto(secuencia)
producto(conjunto)
```

Dada dos secuencias $F = f_0, f_1, f_2, \ldots$ y $G = g_0 g_1 g_2 \ldots$, su **convolución** (ver la sección 7.5 de Graham et al. [1994]) es una secuencia $f_0 g_0, f_0 g_1 + g_0 f_1, \ldots$ donde los elementos valen

$$\sum_{k} f_k g_{n-k}. \tag{2.12}$$

Nota que no importa cuál es f y cuál es g ya que su intercambio no afecta el resultado.

Para entender qué sucede, intenta imaginar las secuencias como cintas. Voltea la cinta de g y ponla encima de la cinta de f así que los elementos cero estén uno encima de otro; el primer elemento viene su multiplicación. Ahora, desliza la cinta de g una posición para que coincidan dos elementos. Multiplica juntos los que estén uno encima de otro y suma sobre las multiplicaciones. Luego que coindican tres elementos, etc.

```
# calculamos con entradas f = [1, 2, 3] y g = [0, 1, 5]
#
# 5 1 0
#  1 2 3
# ---------- * por elemento
#  0   + = 0 de (0, 0)
#
# 5 1 0
#   1 2 3
# ---------- * por elemento
#   1 0   + = 1 de (0, 1) y (1, 0)
#
# 5 1 0
# 1 2 3
# ---------- * por elemento
# 5 2 0   + = 7 de (0, 1, 2) y (2, 1, 0)
#
#   5 1 0
# 1 2 3
# ---------- * por elemento
#  10 3  + = 13 de (1, 2) y (2, 1)
#
#  5 1 0
# 1 2 3
# ---------- * por elemento de (2, 2)
# 15  + = 15
#
# el resultado debe ser [0, 1, 7, 13, 15]

def convo(f, g):
 assert len(f) == len(g) # ocupan tener el mismo largo
 n = len(f)
 desde = 0 # inicio del pedazo a multiplicar
 hasta = 0 # final del pedazo a multiplicar
 c = [] # resultado va en esta lista
 for k in range(2 * n - 1): # para cada elemento
  s = 0 # inicializar la suma
  pos = list(range(desde, hasta + 1)) # pedazo
  l = len(pos) # largo del pedazo
  for i in range(l): # para cada uno
   j = l - (i + 1) # desde el final para g
```

```
    s += f[pos[i]] * g[pos[j]]
   c.append(s) # agregar al resultado
   if k < n - 1: # incrementar pezado en su final
    hasta += 1
   else:
    desde += 1 # disminuir el pedazo al inicio
 return c

f = [1, 2, 3] # datos de prueba
g = [0, 1, 5]
fog = convo(f, g)
print(fog, fog == convo(g, f))
```

También existen versiones donde se toman en cuenta los elementos que no coinciden con otros; son muy importantes en el procesamiento de señales y su versión en dos dimensiones (con matríces en vez de vectores) se necesita en la visión computacional (es decir, procesamento de imágenes).

2.3.1. Aritméticas

Serie: $x_{i+1} - x_i = d$, donde d es constante: $S_n = \sum_{i=0}^{n-1}(x_1 + i \cdot d) = \frac{n(x_1 + x_n)}{2}$.

```
def siguiente(anterior, constante):
    return anterior + constante

n = 10
serie = [7]
d = 6
for pos in range(n - 1):
    serie.append(siguiente(serie[-1], d))

print(serie)
sum(serie) == n * (serie[0] + serie[-1]) / 2
```

Progresión: $\sum_{k=0}^{n}(a + bk) = (a + bk/2)(n + 1)$.

2.3.2. Geométricas

Serie: $x_{i+1} = x_i \cdot d$: $S = \sum_{i=0}^{\infty} d^i x_i = \frac{x_1}{1-d}$ y $S_n = \sum_{i=0}^{n} d^i x_i = \frac{x_1(1-d^{n+1})}{1-d}$.

```
def sig_geom(anterior, constante):
    return anterior * constante

n = 12
geom_serie = [13]
d = 3
```

```
for pos in range(n - 1):
    geom_serie.append(sig_geom(geom_serie[-1], d))

print(geom_serie)
sum(geom_serie) == geom_serie[0] * (1 - d**n) / (1 - d)
```

Progresión: $\sum_{k=0}^{n} a \times x^k = \frac{a - ax^{n+1}}{1-x}$.

2.4. Relaciones, mapeos y funciones

Este tema se trata en el capítulo 5 de Grimaldi [2017] y nuevamente en el capítulo 7. Cualquier cosa que no entiendes viendo esta página tendrás que consultar en el libro. Hay muchos ejemplos.

Relación $\mathcal{R}$

- $A \times B = \{(a, b) \mid a \in A, b \in B\}$.
- $\mathcal{R} \subseteq A \times B$.
- $(a, b) \in \mathcal{R}$ o alternativamente $a\mathcal{R}b$.
- Se pueden representar con una matriz de adyacencia: $a_{ij} = 1$ si $(a_i, a_j) \in \mathcal{R}$ mientras $a_{ij} = 0$ si $(a_i, a_j) \notin \mathcal{R}$.
- Transitiva: $\forall a, b, c \in A$ tales que $a\mathcal{R}b$ y $b\mathcal{R}c$, $a\mathcal{R}c$.
- Reflexiva: $\forall a \in A$, $a\mathcal{R}a$.
- Simétrica: $(a_i, a_j) \in \mathcal{R} \Leftrightarrow (a_j, a_i) \in \mathcal{R}$.

```
R = {(2, 4), (5, 10), (3, 6), (4, 7)}
(3, 6) in R
[b for (a, b) in R if a < 5]
[(a, b) for (a, b) in R if 2*a == b]
```

Mapeo g

- $g : A \to B$, $g(a) = b$ para significar que $(a, b) \in \mathcal{R}_g$.
- A es el *dominio* y B es el *rango*.
- Los $b_i \in B$ para las cuales aplica $(a, b_i) \in \mathcal{R}_g$ forman la *imagen* de a en B.
- Los elementos de A que corresponden a un $b \in B$ así que $(a, b) \in \mathcal{R}_g$ son la *imagen inversa* de b, $g^{-1}(b)$.
- Epiyectivo: $\forall b \in B \exists a \in A$ tal que $g(a) = b$.
- Inyectivo: $\forall a_1, a_2 \in A$ tales que $g(a_1) = g(a_2) \Rightarrow a_1 = a_2$.
- Biyectivo: inyectivo y epiyectivo.

```
M = {'a': 5, 'b': 13, 'c': 27}

for a in M.keys():
    print(a)
```

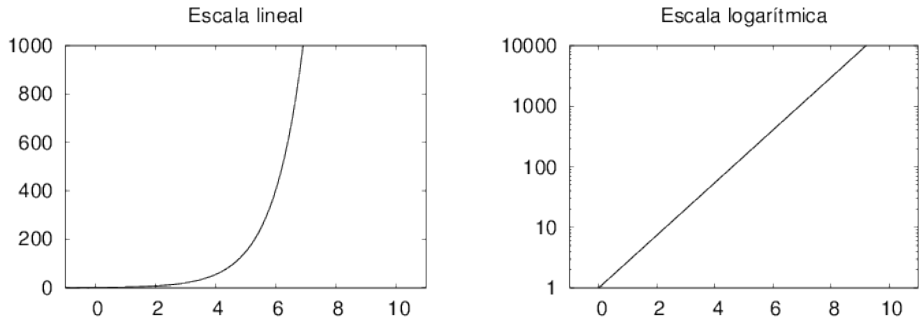


Figura 2.2: La función exponencial en dos escalas.

```
for b in M.values():
    print(b)

M['c'] - M['a']
```

Una **función** $f : A \to B$ es un mapeo que asigna a cada elemento de A un único elemento de B. Por ejemplo,

- $|x| = \begin{cases} x, & \text{si } x \geq 0 \\ -x, & \text{si } x < 0. \end{cases}$
- $\exp(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots.$

Python tiene una librería `math`[2] de funciones comunes.

```
abs(-5)
from math import exp
exp(1.0)
```

$f(x) = \exp(x)$ es la función *exponencial*, que también se define como un límite

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n; \tag{2.13}$$

su derivada es la misma $D(e^x) = e^x$. Además tiene las mismas propiedades que cualquier exponenciación a base b; simplemente en este caso la base $b = e \approx 2,718281828$, lo que es la *constante de Napier*.

- $b^0 = 1$.
- $b^1 = b$.
- $b^{a+c} = b^a b^c$.
- $b^{ac} = (b^a)^c$.
- $b^{-a} = \left(\frac{1}{b}\right)^a = \frac{1}{b^a}$.

Su función *inversa* es el **logaritmo**:

[2] https://docs.python.org/dev/library/math.html

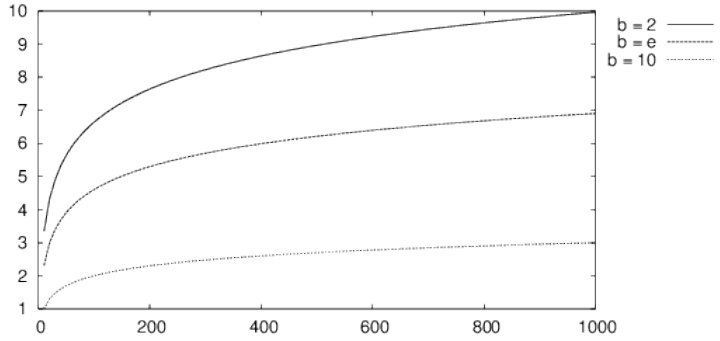


Figura 2.3: Tres funciones logarítmicas.

- La definición es $b^{\log_b x} = x$ donde $x > 0$.
- Si $\log_b x = y$, $b^y = x$ por definición.
- Se define que $\log_b 1 = 0$.
- Cambios de base: $\log_{b'}(x) = \dfrac{\log_b(x)}{\log_b(b')}$.
- Logaritmo es una función inyectiva: $\log_b x = \log_b y \implies x = y$.
- También es creciente: $x > y \implies \log_b x > \log_b y$.
- Aplica para la multiplicación que $\log_b(x \cdot y) = \log_b x + \log_b y$.
- División es un caso especial de la multiplicación: $\log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y$.
- Potencia también es simplemente multiplicación repetida: $\log_b x^c = c \log_b x$.
- Factorial también es una multiplicación repetida: $\log_b(n!) = \sum_{i=1}^{n} \log_b i$.
- Logaritmo en el exponente permite un cambio de variables: $x^{\log_b y} = y^{\log_b x}$.
- El logaritmo *natural* usa la base e: $\ln(x) = \log_e(x)$.

```
from math import log, exp
log(10)
2.302585092994046
log(100, 10)
log(16, 2)
log(exp(1.0))
log(1000, 10) == log(1000)/log(10)
log(8, 2) == log(8) / log(2)
```

2.4.1. Redondeo de decimal a entero

Funciones de redondeo son el tema del capítulo 3 de Graham et al. [1994].

Definición 2. Piso $\lfloor x \rfloor = \max_{y \in \mathbb{Z}} \{y \mid x \geq y\}$.

Necesariamente $\forall x \in \mathbb{R}$, $\lfloor x \rfloor \leq x < \lfloor x+1 \rfloor$, $x - 1 < \lfloor x \rfloor \leq x$ y $\forall k \in \mathbb{Z}$ y $x \in \mathbb{R}$ aplica $\lfloor k + x \rfloor = k + \lfloor x \rfloor$.

```
from math import floor, pi
floor(pi)
```

Definición 3. Techo $\lceil x \rceil = \min_{y \in \mathbb{Z}} \{y \mid x \leq y\}$.

Necesariamente $x \leq \lceil x \rceil < x + 1$.

```
from math import ceil, pi
ceil(pi)
```

Consideremos el siguiente código:

```
a = 3.76 # sin probar
b = int(a)
print(a, b)
```

donde la regla de asignar un valor a `b` es la función de **parte entera**, $\lfloor x \rfloor$. Denote el valor de la variable `a` por a. Si $a \geq 0$, se asigna a `b` el valor $\lfloor a \rfloor$, y cuando $a < 0$, se asigna a `b` el valor $\lceil a \rceil$. El redondeo típico "de mitad para arriba" de $x \in \mathbb{R}$ al entero más próximo es equivalente a $\lfloor x + 0{,}5 \rfloor$.

```
from math import pi
round(pi)
round(3.5000000001) != round(3.4999999999999999)
```

2.5. álgebra abstracta

La teoría de números y la álgebra abstracta en general son importantes para la criptografía que es un campo de la seguridad informática. Para más sobre álgebra abstracta, véase el capítulos 14 y 15 de Grimaldi [2017]. Además de criptografía, es necesario en la codificación de la información (compresión, detección de errores, etc.).

2.5.1. Divisibilidad

Un entero que no tiene divisores aparte de si mismo y el uno es un *número primo*. Los números primos se tratan en la sección 4.3 de Grimaldi [2017].

Para checar si un número dado es primo o no, se puede intentar las divisiones y ver si alguna da un residuo cero (no vamos a preocuparnos por los negativos por simplicidad):

```
def primo(n):
 for i in range(2, n):
  if n % i == 0:
   return False
  return True

primo(176527)
```

```
primo(98217)
primo(97)
```

Pensándolo bien, es estúpido dividir entre 4 y 6 y 8 si ya se sabe que la división entre dos no fue exacta, por lo cual conviene manejar los números pares como un caso especial para hacer menos intentos de división y así ahorrale esfuerzo a la computadora:

```
def primo_v2(n):
 if n < 4:
  return True # 1, 2 y 3 son primos todos
 for i in range(3, n, 2): # en pasos de dos
  if n % i == 0:
   return False
 return True
```

Se puede ahorrar aún más. Supongamos que n no sea primo, por lo cual tendrá que tener por lo menos dos divisores $n = a \times b$. Para saber que n no es primo, basta con encontrar el menor de los divisores. ¿Qué tan grande puede ser el menor de ellos? Cuando a crece, b disminuye ya que $n = ab$. Entonces el menor es a su más grande cuando $a \approx b$, lo que significa que si $a = b$ y $n = ab = a^2$, tenemos que $a = \sqrt{(n)}$, por lo cual basta con probar divisores hasta alcanzar la raíz de n:

```
from math import sqrt
def primo_v3(n):
    if n < 4:
        return True # 1, 2 y 3 son primos todos
    tope = int(ceil(sqrt(n))) # el siguiente entero
    for i in range(3, tope, 2): # en pasos de dos
        if n \% i == 0:
            return False
    return True
```

Para más detalle sobre números primos, checa las secciones 4.2 & 4.3 de Graham et al. [1994]

Se dice que dos enteros son **relativamente primos** si no tienen divisores en común aparte del uno, es decir, su *mayor divisor común* (GCD por sus siglas en inglés) es uno. Nota que $c = \text{GCD}(a, b) \Rightarrow \exists x, y \in \mathbb{Z}$ así que $c = (x \times a) + (y \times b)$. La GDC es el tema de la sección 4.1 de Concrete Mathematics. Nota que dos números primos siempre son relativamente primos entre ellos y un número primo es relativamente primo con cualquier otro entero. Se escribe $a \perp b$ para indicar que a y b son relativamente primos. Los primos relativos se discuten en la sección 4.5 de Concrete Mathematics.

Aplica que el GCD entre a y b es el mismo que el CGD entre b y a mod b, lo que nos permite definir un *algoritmo recursivo* para calcularlo:

```
def gcd(a, b):
 if b == 0:
  return a
 return gcd(b, a % b)

gcd(123456, 123)
```

```
def relativamente_primos(x, y):
 return gcd(x, y) == 1

a = 87665
b = 16731
relativamente_primos(a, b)
```

Este algoritmo para el GCD es el tema de la sección 4.4 de Grimaldi [2017].

2.5.2. Grupos

Sea G un conjunto y $\circ$ un operador binario. $\mathbb{G} = (G, \circ)$ es un *grupo* si aplican las siguientes cuatro propiedades:

1. Clausura: $\forall g, h \in G, i = g \circ h \in G$.
2. Asociatividad: $\forall g, h, i \in G : (g \circ h) \circ i = g \circ (h \circ i)$.
3. Identidad: $\exists e \in G$ tal que $\forall g \in G : e \circ g = g \circ e = g$.
4. Inversión: $\forall g \in G \exists h \in G$ tal que $h \circ g = g \circ h = e$.

La cardinalidad del grupo es $|G|$ y se le dice el *orden* de $\mathbb{G}$. Un grupo es finito si su orden lo es. Por ejemplo el o-exclusivo con los valores de verdad es un grupo: $(\{\top, \bot\}, \oplus)$. Para grupos, aplica lo siguiente:

- Cancelación izquierda: $g \circ h = g \circ i \Rightarrow h = i$.
- Cancelación derecha: $h \circ g = i \circ g \Rightarrow h = i$.
- Repetición: $g \circ g \in G$, $(g \circ g) \circ g = g \circ (g \circ g) \in G$.
- Se escribe $g^2 = g \circ g$, $g^3 = g \circ g^2$, etc.
- Entonces g^i refiere a una aplicación de $\circ$ repetida i veces a g.

Un grupo es *cíclico* si $\exists g \in G$ tal que $\forall h \in G \exists i \in \mathbb{Z}, i \geq 0$ tal que $g^i = h$. Esto quiere decir que todos los elementos de un grupo cíclico son potencias de algún elemento de ese grupo. A ese elemento se le dice el *generador* del grupo.

Sea i el menor entero positivo tal que $g^i = e$. Si existe tal i, su valor es el orden del elemento h. Si no existe, se dice que g es de orden infinito. Se escribe $i = \text{ord}(g)$.

$\mathbb{H}$ es un *subgrupo* de $\mathbb{G}$ si ambos son grupos, tienen el mismo operador y los elementos del primero forman un subconjunto del segundo. Se escribe $\mathbb{H} \sqsubseteq \mathbb{G}$. El subgrupo generado por un elemento g contiene los elementos en $H_g = \{h \mid \exists i \text{ tal que } h = g^i\}$, donde $|H_g| = \text{ord}(g)$ ya que g es el generador de $\mathbb{H}_g$. En grupos finitos, cualquier elemento genera un subgrupo, por lo cual no existen elementos de orden infinito en grupos finitos.

Un *morfismo* $f()$ es una función que mapea los elementos de un grupo $(G, \circ)$ a los de otro grupo $(H, \bullet)$ sin afectar la estructura:

$$f : G \to H \wedge f(g \circ h) = f(g) \bullet f(h). \tag{2.14}$$

En un grupo aplica que $\forall g \in G$ que $\phi : n \to g^n$ es un morfismo de los enteros al subgrupo H_g. Su demostración requiere que se defina $g^0 = e$ y $g^{-n} = (g^n) - 1$.

La relación de **congruencia** se define a los números naturales $\mathbb{N}$ en términos de conjuntos $\{x \mid \exists y \in \mathbb{N}$ así que $((y \times a) + b = x)\}$; se dice que x es congruente con b en módulo a si x pertenece al mismo conjunto con b, en cual caso se escribe $x \equiv b \mod a$. Nota que a es congruente a b en módulo n si y sólo si $(a - b)$ es divisible entre n: $a \equiv b \mod n \Leftrightarrow n \mid (a - b)$. La relación de congruencia se discute en las secciones 4.6–9 de Concrete Mathematics.

Aplica que $x \mod n + y \mod n = (x + y) \mod n$; la sumación modular forma un grupo. También aplica que $x \mod n \times y \mod n = (x \times y) \mod n$; la multiplicación modular forma un grupo en aquellos elementos del conjunto residual de n que son relativamente primos con n, $(i \perp n) \wedge (j \perp n) \Rightarrow ((i \times j) \mod n) \perp n$.

Sea $\mathbb{G} = (G, \circ)$ un grupo y $\mathbb{H} = (H, \circ) \sqsubseteq \mathbb{G}$. Para $g \in G$, se define el co-conjunto (izquierdo) de $\mathbb{H}$ como $g \circ H = \{g \circ h \mid h \in H\}$. Aplica $\forall g, H : |H| = |g \circ H| \wedge H \cap g \circ H = \emptyset$. Además,

$$\forall a, b \in G, H \sqsubseteq G \Rightarrow ((a \circ H = b \circ H) \vee ((a \circ H) \cap (b \circ H) = \emptyset)). \tag{2.15}$$

El teorema de Lagrange establece que $(H \sqsubseteq G \wedge |G| = a < \infty \wedge |H| = b) \Rightarrow b|a$. Un corollario (es decir, consecuencia) de esto es que si $|G| = n$, $\forall g \in G : \mathrm{ord}(g)|n$.

La función de Euler se define como $\phi(n) = |\{p \in \mathbb{Z}$ tal que $1 < p \leq n \wedge p \perp n\}|$. Para n que son primos (o sea, no tienen divisores aparte de uno y si mismo), aplica trivialmente que $\phi(n) = n - 1$.

El teorema de producto indica que $p \perp q \Rightarrow \phi(p \times q) = \phi(p) \times \phi(q)$, mientras el teorema de Euler establece que $a \perp n \Rightarrow a^{\phi(n)} \equiv 1 \mod n$. El teorema chica de Fermat da para un primo p que $a^{p-1} \equiv 1 \mod p$. Además aplica que $n \perp (p \times q) \Leftrightarrow (n \perp p \wedge n \perp q)$.

2.6. Lenguajes y autómatas

Lenguajes y autómatas son el tema del capítulo 6 de Grimaldi [2017].

Un *alfabeto* refiere a un conjunto de símbolos (las *letras*), mientras una *palabra* es una sucesión de letras. Por ejemplo, con el alfabeto $\{a, b, c\}$, se puede formar la palabra "cab". La **gramática** son las reglas que definen un *lenguaje*. Por ejemplo, un lenguaje es aquellas palabras del alfabeto mencionado que contienen por lo menos una letra "a", su largo no excede a cuatro letras y terminan en "c", en cual caso el lenguje corresponde al conjunto finito $\mathcal{L} = \{ac, aac, abc, bac, acc, cac, aaac, aabc, baac, abac, abbc, babc, bbac, accc, cacc, ccac\}$.

Un *autómata* es un conjunto de *estados* junto con una función de transiciones; cuenta con un estado inicial definido y posiblemente uno o más estados finales. Las transiciones pueden activarse según una *entrada* o un *eventos*. Técnicamente los símbolos de la entrada se consideran una secuencia de eventos individuales. Un *autómata determinista* tiene una sola opción por entrada. Uno *no determinista* tiene por lo menos en un estado hay dos o más posibles transiciones para por lo menos un evento.

Una *máquina Turing* (TM) es un **modelo formal de computación** en términos de un autómata determinista $M = (K, \Sigma, \delta, s)$ que utiliza una sola estructura de datos: una sucesión de símbolos escrita en una **cinta** (infinita) que permite borrar e imprimir símbolos. K es su conjunto finito de **estados**, donde el estado inicial es $s \in K$. Además cuenta con

estados finales "alto", "sí" y "no". Opera con un alfabeto finito de **símbolos** Σ tal que $\sqcup, \triangleright \in \Sigma$. Su **función de transición**

$$\delta : K \times \Sigma \to (K \cup \{\text{alto}, \text{sí}, \text{no}\}) \times \Sigma \times \{\to, \leftarrow, -\} \tag{2.16}$$

La cinta está siendo accedida por una cabeza lector al cual le llamamos "puntero"; este tiene tres posibles acciones: $\to$ (desplazar una posición hacia la derecha), $\leftarrow$ (lo mismo pero hacia la izquierda) y $-$ (no moverse).

La función de transición δ captura el "programa" de la TM. Si el **estado actual** es $q \in K$ y el *símbolo actualmente bajo del puntero* es $\sigma \in \Sigma$, tenemos $\delta(q, \sigma) = (p, \rho, D)$, donde

- p es el **estado nuevo**,
- ρ es el *símbolo que será escrito* en el lugar de σ,
- $D \in \{\to, \leftarrow, -\}$ indica como mueve del puntero.
- Si el puntero mueve *afuera* de la sucesión de entrada a la derecha, el símbolo que es leído es siempre $\sqcup$ (un símbolo blanco).

Cada programa comienza con la siguiente configuración:

- la TM en el estado inicial $s \in K$,
- con la cinta inicializada a contener $\triangleright x$, donde x es una sucesión finita de símbolos en $(\Sigma - \{\sqcup\})^*$,
- el puntero puntando a $\triangleright$ en la cinta.

La secuencia x es la entrada de la máquina. Una TM se ha **detenido** al haber llegado a un estado de alto $\{\text{alto}, \text{sí}, \text{no}\}$. Si la máquina se detuvo en sí, la máquina **acepta** la entrada. Si la máquina se detuvo en no, la máquina **rechaza** su entrada. La **salida** $M(x)$ de la máquina M con la entrada x se define como

- $M(x) =$ "sí" si la máquina acepta x.
- $M(x) =$ "no" si la máquina rechaza x.
- $M(x) = y$ si la máquina llega a "alto" y $\triangleright y \sqcup \sqcup \ldots$ es la sucesión escrita en la cinta de M en el momento de detenerse..
- $M(x) = \nearrow$ si M nunca se detiene con la entrada x.

Estudiamos como ejemplo el cómputo de $n+1$ dado $n \in \mathbb{Z}$ para $n > 0$ en representación binaria con por lo menos un cero inicial: $K = \{s, q\}$; $\Sigma = \{0, 1, \sqcup, \triangleright\}$ y la δ viene en el cuadro 2.1.

Las máquinas Turing son una representación bastante natural para resolver muchos problemas sobre sucesiones. Por ejemplo, realizan **reconocimiento de lenguajes**. Un **lenguaje** se define como $L \subset (\Sigma - \{\sqcup\})^*$, donde el símbolo $*$ indica que se crea una secuencia con cero o más símbolos que provienen de ese conjunto, permitiendo repeticiones. En el contexto de las máquinas Turing, debemos suponer que el alfabeto del lenguaje **no** contiene el símbolo "vacío" que llena la cinta al inicio.

Una máquina Turing M **decide** el lenguaje L si y sólo si para toda sucesión $x \in (\Sigma \backslash \{\sqcup\})^*$ aplica que si $x \in L$, $M(x) =$ sí y si $x \notin L$, $M(x) =$ no.

$$L = a^* = \{\epsilon, a, aa, aaa, aaaa, \ldots\}. \tag{2.17}$$

Cuadro 2.1: δ para la Máquina Turing ejemplo.

$p \in K$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s	0	$(s, 0, \rightarrow)$
s	1	$(s, 1, \rightarrow)$
s	$\sqcup$	$(q, \sqcup, \leftarrow)$
s	$\triangleright$	$(s, \triangleright, \rightarrow)$
q	0	$(\text{alto}, 1, -)$
q	1	$(q, 0, \leftarrow)$
q	$\triangleright$	$(\text{alto}, \triangleright, \rightarrow)$

La clase de lenguajes decididos por alguna máquina Turing son los lenguajes **recursivos**.

Una máquina Turing **acepta** un lenguaje L si para toda sucesión $x \in (\Sigma - \{\sqcup\})^*$ aplica que si $x \in L$, $M(x) =$ sí, pero si $x \notin L$, $M(x) = \nearrow$. Los lenguajes aceptados por algún máquina Turing son **recursivamente numerables**. Note que si L es recursivo, también es recursivamente numerable.

2.7. Tarea 2

Las tareas se responden en línea. Hay que usar el mismo usuario para cada tarea y registrar el usuario con la matrícula correspondiente en la página de resultados del semestre actual para recibir puntos por ellas.

1. La cantidad de permutaciones del conjunto proporcionado en la tarea personalizada, tomando en cuenta las restricciones mencionadas.
2. La cantidad de subconjuntos del conjunto proporcionado en la tarea personalizada, tomando en cuenta la restricción mencionada.
3. El último elemento del fragmento inicial de la sucesión de Fibonacci que contiene el elementos especificado en la tarea personalizada.
4. La salida de la siguiente máquina Turing con la entrada proporcionada en la tarea personalizada (nota que $\triangleright$ no forma parte ni de la salida ni de la entrada, sino es fijo en la cinta):

$p \in K$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s	$\triangleright$	$(s, \triangleright, \rightarrow)$
s	0	$(s, 0, \rightarrow)$
s	1	$(s, 1, \rightarrow)$
s	$\sqcup$	$(q, \sqcup, \leftarrow)$
q	0	$(t, \sqcup, \leftarrow)$
q	1	$(t, \sqcup, \leftarrow)$
q	$\triangleright$	$(\mathsf{alto}, \triangleright, \rightarrow)$
t	$\triangleright$	$(\mathsf{alto}, \triangleright, -)$
t	0	$(\mathsf{alto}, 0, -)$
t	1	$(\mathsf{alto}, 1, -)$

5. La cantidad de veces que se evalúa δ en la ejecución de la TM con la entrada de la tarea personalizada.

2.7.1. Preguntas de verificación

Discute lo siguiente con los compañeros y con la profesora hasta que esté todo claro. Conviene consultar el libro de texto (capítulos y secciones indicados en el material de la unidad en esta página) y hasta buscar por videos en la web. Cuando ya no cabe duda, procede al repaso con los viejos exámenes de medio curso en la página web de la unidad de aprendizaje, comenzando desde el más antiguo ya que es el más fácil.

¿Qué significa permutar? ¿Por qué son $n!$ permutaciones para n elementos? ¿Qué pasaría a la cantidad de permutaciones si algún elemento se repitiera dos o más veces? ¿Para qué sirven las permutaciones en la ingeniería? (Pregunta 1)

¿Para que se les llama *combinaciones* a los subconjuntos? ¿Qué explicación tiene la fórmula del coeficiente binomial? ¿Por qué se llama así, de hecho? ¿Qué relación tiene el coeficiente binomial con el triángulo de Pascal (ver figura 3.3)?

¿Por qué suman a una potencia de dos todos los renglones del triángulo? ¿Qué pasa si se permiten repeticiones de los elementos a la cantidad de posibles combinaciones? ¿En qué situaciones se aplicaría esto en la ingeniería? (Pregunta 2)

Figura 2.4: Un mayor fragmento del triángulo de Pascal.

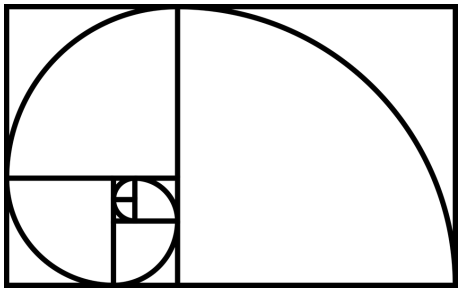


Figura 2.5: El espiral de Fibonacci.

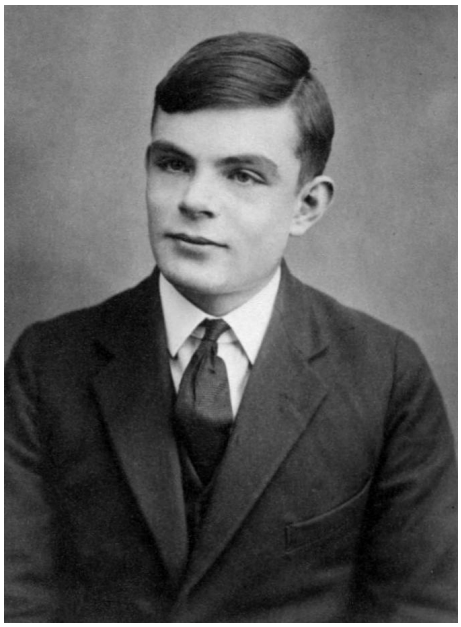


Figura 2.6: Alan Turing a los 16 años de edad.

¿Por qué se le dice *recursiva* a la definición de la sucesión? ¿Qué tiene que ver la suceción de Fibonacci con la tasa dorada? ¿Cómo se relacionan estos números al famoso espiral de Fibonacci (ver la figura 2.5?

¿Que otras secuencias interesantes existen? ¿ Checa por ejemplo `https://oeis.org/`.

Se podría variar la definición usando restas o multiplicaciones en lugar de la suma? (Pregunta 3)

¿Quién fue Alan Turing (figura 2.6? ¿Qué hizo de importancia? ¿Tendría sentido físicamente construir una máquina Turing? ¿Qué relación tienen las TM con las computadoras? ¿Qué hace la TM del ejemplo en el sentido matemático? ¿Que hace la TM de la tarea en el sentido matemático? ¿Cómo se vería la tabla de la función de transición para una TM que reste uno a un número binario o que divida entre dos, cuatro, etc.? ¿Por qué es importante el cuántos pasos ocupa una TM con una entrada? ¿En términos de qué se pudiera medir el consumo de memoria de una ejecución de una TM ya que el número de pasos sirve para medir el tiempo de ejecución? (Preguntas 4 y 5)

3 Grafos y árboles

3.1. Grafos

Un *grafo* $\mathcal{G}$ es un par de conjuntos $\mathcal{G} = (V, E)$. V es un conjunto de n **vértices** $u, v, w \in V$ mientras E es un conjunto de m **aristas**; $|V| = n$ es el *orden* del grafo y $|E| = m$ se llama el *tamaño* del grafo. Las aristas son típicamente *pares de vértices*, $\{u, v\} \in E$, en cual caso $E \subseteq V \times V$ — se escribe (u, v) como notación alternativa en vez de representar la arista como un subconjunto tal cual.

Grafos se empiezan a ver desde la sección 7.2 en Grimaldi [2017] y son el tema principal de toda la tercera parte del libro, desde capítulo 11.

Los vértices se suelen dibujar como circulos y las aristas como líneas que les conectan uno al otro.

También se puede definir grafos donde el producto es entre más de dos "copias" del conjunto V, el cual caso se habla de *hípergrafos*. El **complemento** de un grafo $\mathcal{G} = (V, E)$ es un grafo con los mismos vértices pero solamente con aquellas aristas que no están en E.

Un grafo es **plano** si se puede *dibujar en dos dimensiones* así que ninguna arista cruza a otra arista. En un grafo *no dirigido*, los vértices v y w tienen un papel igual en la arista $\{v, u\}$. Si las aristas tienen *dirección*, $\mathcal{G}$ es **dirigido** (también *digrafo*). En una arista dirigida $\langle v, w \rangle$:

- v es el *origen* (o inicio) de la arista y
- w es el *destino* (o fin) de la arista.

Al visualizar grafos dirigidos, las aristas se suelen dibujar como flechas de su origen a su destino.

Un **bucle** es una arista *reflexiva*, donde coinciden el vértice de origen y el vértice de destino: $\{v, v\}$ o $\langle v, v \rangle$. Si un grafo $\mathcal{G}$ no cuente con *ningún* bucle, el grafo es no reflexivo.

Además, E podría ser un *multiconjunto*: más de una arista entre un par de vértices. Si no se permiten aristas múltiples, el grafo es *simple*. Si se asignan *pesos* $\omega(v, w)$ a las aristas, el grafo es **ponderado**. Si se asigna *identidad* a los vértices o las aristas, el grafo es **etiquetado**.

Dos aristas $\{v_1, v_2\}$ y $\{w_1, w_2\}$ son **adyacentes** si tienen un vértice en común:

$$|\{v_1, v_2\} \cap \{w_1, w_2\}| \geq 1. \tag{3.1}$$

Una arista es **incidente** a un vértice si ésta lo une a otro vértice. Vértices v y w son **adyacentes** si una arista los une: $\{v, w\} \in E$. Vértices adyacentes son llamados **vecinos**. El conjunto de vecinos de v es su *vecindad*, $\Gamma(v)$.

La matriz que corresponde a la relación E se llama la **matriz de adyacencia** del grafo, A. Es necesario etiquetar los vértices para que sean identificados como $v_1, v_2, \ldots, v_n$.

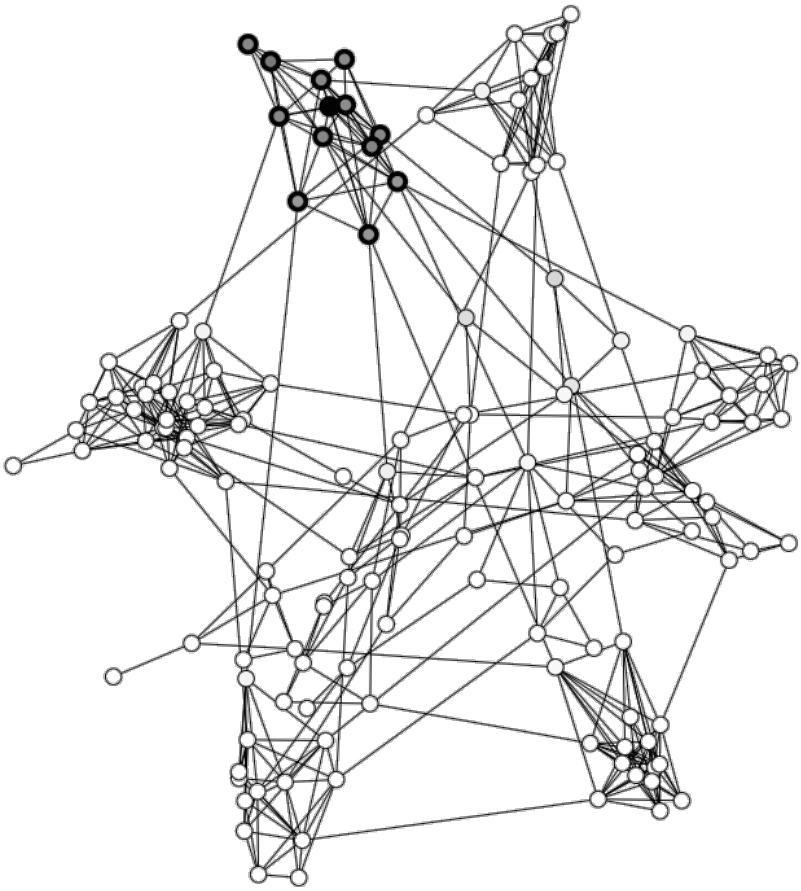


Figura 3.1: Los vértices del grafo se representan con círculos, posiblemente rellenos de colores, y las aristas se representan con líneas.

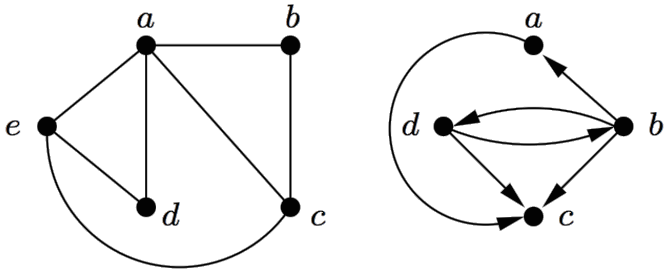


Figura 3.2: Un grafo no dirigido y uno dirigido.

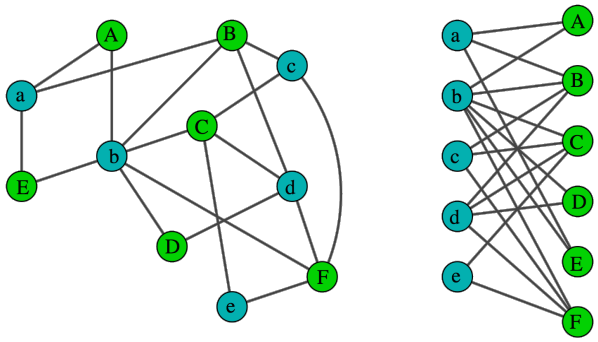


Figura 3.3: Un mismo grafo bipartito, dibujado dos veces.

- Para un grafo *no dirigido* A es *simétrica*.
- **Multigrafos**: una matriz de enteros A' donde $a'_{i,j} \geq 0$ es el número de aristas entre v_i y v_j.
- Grafos **ponderados**: una matriz (real) A donde $a_{i,j}$ es el peso de la arista $\{v_i, v_j\}$ o cero si no hay tal arista.

Se puede visualizar una matriz (de adyacencia o lo que sea) asignando colores a pixeles según el valor de los elementos; por ejemplo para una matriz de adyacencia de un grafo simple no dirigido negro puede significar uno y blanco cero. Si hay pesos, se pueden usar escalas de grises para rango de pesos.

El **grado** $\deg(v)$ es el número de aristas incidentes a v. Para grafos dirigidos,

- el *grado de salida* $\overrightarrow{\deg}(v)$ es el número de aristas que tienen su origen en v y
- el *grado de entrada* $\overleftarrow{\deg}(v)$ es el número de aristas que tienen su destino en v.

El grado total de un vértice de un grafo dirigido es

$$\deg(v) = \overleftarrow{\deg}(v) + \overrightarrow{\deg}(v). \tag{3.2}$$

En un grafo *simple no dirigido*, el grado $\deg(v_i)$ del vértice v_i es la suma de la $i$ésima fila de A,

$$\sum_{v \in V} \deg(v) = 2m. \tag{3.3}$$

En un grafo *simple no reflexivo* aplica que $\deg(v) = |\Gamma(v)|$. Si *todos* los grados son k, el grafo es k-**regular**; un grafo $(n-1)$-regular se llama un grafo *completo* K_n.

Un grafo *bipartito* es un grafo $\mathcal{G} = (V, E)$ cuyos vértices se pueden separar en dos conjuntos U y Q de tal forma que $U \cap W = \emptyset$ y $U \cup W = V$ y que además

$$\{u, w\} \in E \Rightarrow (u \in U \wedge w \in W) \vee (u \in W \wedge w \in U). \tag{3.4}$$

En un grafo bipartito completo están presentes todas las aristas permitidas, $K_{|U|,|W|}$.

El número máximo posible de aristas en un grafo *simple* es

$$m_{\text{máx}} = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{3.5}$$

Para K_n, tenemos $m = m_{\text{máx}}$.

La *densidad* es la proporción de aristas presentes:

$$\delta(\mathcal{G}) = \frac{m}{m_{\text{máx}}} = \frac{m}{\binom{n}{2}}. \tag{3.6}$$

Un grafo *denso* tiene $\delta(\mathcal{G}) \approx 1$ y un grafo *escaso* tiene $\delta(\mathcal{G}) \ll 1$ (lo de $\ll$ significa "mucho menor"; de la misma manera, $\gg$ significa "mucho mayor").

Una sucesión de aristas adyacentes que empieza en v y termina en w es un **camino** de v a w. El **largo** de un camino es el *número de aristas* que contiene La **distancia** $\text{dist}(v, w)$ entre v y w es el largo mínimo de todos los caminos de v a w. La distancia de un vértice a si mismo es cero. El **diámetro** $\text{diam}(\mathcal{G})$ de $\mathcal{G}$ es la distancia máxima $\text{diam}(\mathcal{G}) = \text{máx}_{v\in V, w\in V} \text{dist}(v, w)$.

Un camino **simple** solamente recorre cada arista una sola vez o ninguna. Un **ciclo** es un camino que regresa a su vértice inicial. Un grafo que no cuente con ningún ciclo es *acíclico*. Entonces, un ciclo simple empieza y regresa del mismo vértice, pero no visita a ningún otro vértice dos veces. Sin embargo, la elección del punto de inicio de un ciclo es arbitrario.

Un grafo $\mathcal{G}$ es **conexo** si *cada* par de vértices está conectado por un camino. Si por algunos v y w no existe ningún camino, grafo es *no conexo*. $\mathcal{G}$ es *fuertemente conexo* si cada par de vértices está conectado por *al menos dos* caminos disjuntos. Un grafo no conexo se puede dividir en dos o más **componentes conexos** que son formados por tales conjuntos de vértices de distancia definida.

$\mathcal{G}' = (V', E')$ es un **subgrafo** de $\mathcal{G} = (V, E)$ si $V' \subseteq V$ y $E' \subseteq E$ tal que

$$\{v, w\} \in E' \Rightarrow \big((v \in V') \wedge (w \in V')\big). \tag{3.7}$$

Si el subgrafo contiene todas las aristas posibles, es un **subgrafo inducido** por el conjunto V'. A un subgrafo que completo se dice una **camarilla** (inglés: clique).

Un **árbol** es un grafo conexo acíclico. Un **árbol de expansión** de $\mathcal{G} = (V, E)$ es un subgrafo que es un árbol y contiene todos los vértices de $\mathcal{G}$; también se conoce como árbol cubriente. Si el grafo es *ponderado*, el árbol de expansión *mínimo* es cualquier árbol donde la suma de los pesos de sus aristas es mínima. Los árboles son el tema del capítulo 12 de Grimaldi [2017].

Un grafo $\mathcal{G}$ no conexo es un **bosque** si cada componente conexo de $\mathcal{G}$ es un árbol.

Grafos son *isomorfos* si existe una bijección de los vértices de uno a los del otro que preserve las aristas.

Los tres grafos en la figura 3.4 tienen la misma estructura, llamada el grafo de Petersen.

3.1.1. Grafos en Python

Podemos definir con programación orientada a objetos una clase para representar un grafo; guarda esto en un **archivo** grafo.py (copiar y pegar no va a funcionar con los espacios entre las rutinas):

```
class Grafo:

 def __init__(self):
```

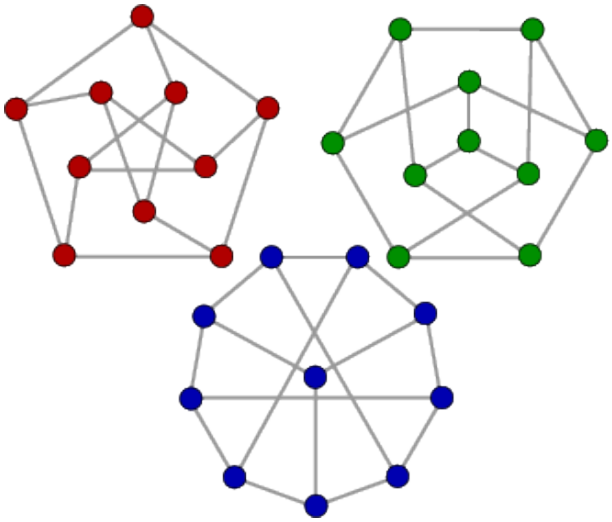


Figura 3.4: El grafo de Petersen, dibujado de tres formas.

```
  self.V = set() # un conjunto
  self.E = dict() # un mapeo de pesos de aristas
  self.vecinos = dict() # un mapeo

 def agrega(self, v):
  self.V.add(v)
  if not v in self.vecinos: # vecindad de v
   self.vecinos[v] = set() # inicialmente no tiene nada

 def conecta(self, v, u, peso=1):
  self.agrega(v)
  self.agrega(u)
  self.E[(v, u)] = self.E[(u, v)] = peso # en ambos sentidos
  self.vecinos[v].add(u)
  self.vecinos[u].add(v)

 def complemento(self):
  comp= Grafo()
  for v in self.V:
   for w in self.V:
 if v != w and (v, w) not in self.E:
  comp.conecta(v, w, 1)
  return comp
```

Ejecutando Python en la carpeta en la cual guardaste el archivo, puedes ahora usar esa clase:

```
from grafo import Grafo
G = Grafo()
G.conecta('a', 'b', 5)
```

```
G.conecta('a', 'c', 7)
G.conecta('b', 'c', 2)
G.conecta('c', 'd', 4)
print(G.vecinos['a'])
print(G.V)
print(G.E)
G2 = G.complemento()
print(G2.E)
```

Definición 4. REACHABILITY *(problema de alcance)*
Entrada*: un grafo* $\mathcal{G} = (V, E)$ *y dos vértices* $v, u \in V$.
Pregunta*: ¿existe un camino de* v *a* u*?*

Tiene un *problema complemento* (cuya respuesta es "falso" si y sólo si el problema original tiene respuesta "verdad": ¿es verdad que no existe ningún camino de v a u? El algoritmo **Floyd-Warshall** es un algoritmo básico para *Reachability* que en realidad determina los largos de los *caminos más cortos* hasta para grafos ponderados. Los pesos tienen que ser *no negativos* para que funcione. El algoritmo construye de una manera incremental estimaciones a los caminos más cortos entre dos vértices hasta llegar a la solución óptima. Etiquetemos los vértices de $\mathcal{G} = (V, E)$ así que $V = \{1, 2, \ldots, n\}$.

Supongamos que $C(i, j, k)$ construye el camino más corto entre los vértices i y j pasando *solamente* por vértices con etiqueta $\leq k$. Para un camino de i a j con vértices intermedios con menores o iguales a $k + 1$, hay dos opciones:

- O el camino más corto con etiquetas $\leq k+1$ utiliza *solamente* vértices con etiquetas $\leq k$.
- O existe algún camino que primero va de i a $k + 1$ y después de $k + 1$ a j así que la *combinación* de estos dos caminos es más corto que cualquier camino que solamente utiliza vértices con etiquetas menores a $k + 1$.

Aplica que

$$C(i, j, k) = \text{mín}\{C(i, j, k - 1), C(i, k, k - 1) + C(k, j, k - 1)\} \tag{3.8}$$

con la condición inicial: $C(i, j, 0) = w(i, j)$, donde $w(i, j)$ es el peso de $(i, j) \in E$.

Para grafos no ponderados se utiliza $C(i, j, 0) = 1$ para cada arista. Donde no hay arista, se asigna $C(i, j, 0) = \infty$.

La computación de C procede de la siguiente manera: iteremos primero con $k = 1$, después con $k = 2$, continuando hasta $k = n$ la formulación recursiva para cada par $\{i, j\}$. La información de la iteración k se puede sobre-escribir con la de la iteración $k + 1$, para ahorrar espacio. La complejidad asintótica del algoritmo es $\mathcal{O}(n^3)$ y el uso de memoria es *cuadrático* (sigue pendiente ver qué significa esto — viene más adelante en el curso).

```
def floyd_warshall(G):
 d = {}
 for v in G.V:
  d[(v, v)] = 0 # distancia reflexiva es cero
  for u in G.vecinos[v]: # para vecinos, la distancia es el peso
  d[(v, u)] = G.E[(v, u)]
```

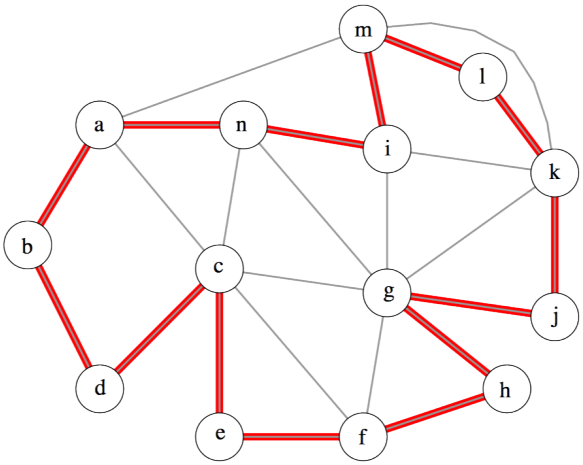


Figura 3.5: Un ciclo de Hamilton.

```
for intermedio in G.V:
 for desde in G.V:
  for hasta in G.V:
di = None
if (desde, intermedio) in d:
 di = d[(desde, intermedio)]
ih = None
if (intermedio, hasta) in d:
 ih = d[(intermedio, hasta)]
if di is not None and ih is not None:
 c = di + ih # largo del camino via ``i''
 if (desde, hasta) not in d or c < d[(desde, hasta)]:
  d[(desde, hasta)] = c # mejora al camino actual
return d
```

Otros problemas famosos de grafos involucran la existencia y la construcción de **ciclos y caminos de Hamilton**, donde la entrada es un grafo $\mathcal{G} = (V, E)$. La figura 3.5 tiene un ejemplo.

Definición 5. Hamiltonian Path
Entrada: un grafo G.
Pregunta: ¿existe un camino C en $\mathcal{G}$ tal que C visite cada vértice exactamente una vez?

Definición 6. Hamiltonian Cycle
Entrada: un grafo G Pregunta: ¿existe un ciclo C en $\mathcal{G}$ tal que C visite cada vértice exactamente una vez?

Caminos y ciclos Hamiltonianos son el tema de la sección 11.5 de Grimaldi [2017].

3.1.2. Circuitos booleanos

Circuitos booleanos son el tema de la sección 15.2 de Grimaldi [2017] — los llama redes de puertas.

- Esencialmente grafos dirigidos no ciclicos.
- Los vértices son "puertas" $V = \{1, 2, \ldots, n\}$.
- Las etiquetas están asignadas así que para cada arista $\langle i, j \rangle \in E$ aplica que $i < j$.
- Una puerta corresponde o a una x_i o $\top$ o $\bot$ o un conectivo.
- Las que corresponden a variables o los valores de verdad tienen grado de entrada cero.
- Las de tipo negación tienen grado de entrada uno.
- Las de $\wedge$ o $\vee$ tienen grado de entrada dos.
- La última puerta n es la salida del circuito.

Los valores de verdad de las distintas puertas se determina con un procedimiento inductivo así que se define el valor para cada puerta todas las entradas de la cual ya están definidos. Los circuitos pueden ser representaciones *más compactas* que las expresiones: en un circuito se puede compartir subcircuitos. El *teorema de Cook* establece que un circuito $R(x)$ está satisfactible si y sólo si existe una secuencia de elecciones tal que la tabla de computación es aceptante si y sólo si $x \in L$.

3.2. Tarea 3

Las tareas se responden en línea. Hay que usar el mismo usuario para cada tarea y registrar el usuario con la matrícula correspondiente en la página de resultados del semestre actual para recibir puntos por ellas.

Determina lo solicitado abajo para el grafo no dirigido con $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ y $E = \{(1, 2), (1, 3), (2, 4), (u, v), (3, 5), (3, 6), (4, 6), (4, 7), (5, 8), (6, 9), (7, 9)\}$, con los valores proporcionados para u y v en la tarea personalizada:

1. El grado máximo.
2. La densidad.
3. La distancia entre 1 y v.
4. El diámetro.
5. La cantidad de aristas en el subgrafo inducido con $V' = V \setminus \{u, v\}$.

3.2.1. Preguntas de verificación

Discute lo siguiente con los compañeros y con la profesora hasta que esté todo claro. Conviene consultar el libro de texto (capítulos y secciones indicados en el material de la unidad en esta página) y hasta buscar por videos en la web. Cuando ya no cabe duda, procede a la siguiente parte del material de estudio.

¿Qué tipo de sistemas se puede modelar con grafos? Inventa e investiga ejemplos en tu propia área de ingeniería.

¿En cuáles de esos ejemplos se necesita grafos dirigidos? ¿En cuáles se ocupa ponderación a las aristas? ¿Hay alguno que ocupe multigrafos? ¿Alguno que ocupe ser un hipergrafo?

¿En qué modelos se necesita incluir ciclos? ¿En qué caso, por ejemplo, se ocuparía bucles? ¿Cuáles son conexos y cuáles pueden ser desconexos? (Debería ser posible inventar ejemplos para cada caso. Si no se te ocurre nada, investiga en línea o en el libro de texto.)

¿Para qué sirve dibujar los grafos? ¿Qué es una buena forma de decidir las posiciones de los vértices para que el dibujo salga entendible?

¿Para qué sirve representar grafos en Python? ¿Por qué no basta con dibujarlos?

3.3. Problemas y algoritmos

Un **problema** es un *conjunto* (posiblemente infinito) de *instancias* junto con una *pregunta* sobre alguna propiedad de las instancias. Formalmente dicho, es un conjunto de instancias al cual corresponde un conjunto de soluciones, junto con una relación que asocia para cada instancia del problema un subconjunto de soluciones (posiblemente vacío). Se clasifican en dos grupos:

- Problemas de **decisión** donde la respuesta es "sí" o "no".
- Problemas de **optimización** donde la pregunta es del tipo
  - "cuál es el mejor valor posible" o
  - "con qué configuración se obtiene el mejor valor posible".

Con problemas de decisión, la tarea es decidir **sí o no** la relación entre instancias y soluciones asigna un *subconjunto vacío* a una dada instancia. Si existen soluciones, la respuesta a la pregunta del problema es "sí", y si el subconjunto es vacío, la respuesta es "no".

Resolver un **problema de decisión** por una TM significa *decidir un lenguaje* que consiste de representaciones de las instancias del problema que corresponden a la respuesta "sí". En *problemas de optimización* la TM hace el cómputo de una función apropiada de sucesiones a sucesiones, representando tanto la entrada como la salida en formato de sucesiones con un alfabeto adecuado.

Dado Σ y $L \subseteq \Sigma^*$, el **complemento** de L es $\bar{L} = \Sigma^* \setminus L$. Aplica que $\bar{A}(x) =$ "sí" si y sólo si $A(x) =$ "no".

La clase TIME$(f(n))$ es el conjunto de lenguajes L tales que una máquina Turing **determinista** decide L en tiempo $f(n)$. La clase de complejidad NTIME$(f(n))$ es el conjunto de lenguajes L tales que una máquina Turing **no determinista** decide L en tiempo $f(n)$.

El conjunto P contiene todos los lenguajes decididos por las TM *deterministas* en tiempo *polinomial*, P $= \bigcup_{k>0}$ TIME(n^k). El conjunto NP contiene todos los lenguajes decididos por máquinas Turing *no deterministas* en tiempo *polinomial*, NP $= \bigcup_{k>0}$ NTIME(n^k).

Se dice que un lenguaje $\mathcal{L}$ es *completo* en su clase si una TM que decide a $\mathcal{L}$ es capaz de decidir a todos los demás lenguajes de esa clase, utilizando un TM auxiliar que realiza una *reducción* eficiente.

3.3.1. Algoritmos

Un **algoritmo** es un *proceso formal* para *encontrar la respuesta* correcta a la pregunta de un problema *para una instancia dada* de un problema específico.

- ¿Cómo encontrar un nombre en la guía telefónica?
- ¿Cómo llegar de mi casa a mi oficina?
- ¿Cómo determinar si un dado número es un número primo?

Para un problema, por lo general existen varios algoritmos con diferente nivel de *eficiencia*. Es decir, diferentes algoritmos pueden tener diferentes **tiempos de ejecución** con la *misma* instancia del problema.

- Un conjunto $\mathcal{E}$ de las **entradas** del algoritmo, que representan las instancias del problema.
- Un conjunto $\mathcal{S}$ de las **salidas**, que son los posibles resultados de la ejecución del algoritmo.
- La salida del un **algoritmo determinista** depende únicamente de la entrada: $f : \mathcal{E} \to \mathcal{S}$.
- Existen también algoritmos **probabilistas** o *aleatorizados* donde **no** es así.

Los algoritmos se escribe como sucesiones de **instrucciones** que *procesan* la entrada $\rho \in \mathcal{E}$ para producir el resultado $\xi \in \mathcal{S}$. Cada instrucción es una operación simple que es posible ejecutar con eficiencia y produce un resultado intermedio único. La sucesión S de instrucciones tiene que ser **finita** y $\forall \rho \in \mathcal{E}$, si P está ejecutada con la entrada ρ, el resultado de la computación será $f(\rho) \in \mathcal{S}$. Sería altamente deseable que para todo $\rho \in \mathcal{E}$, la ejecución de S terminará después de un **tiempo finito**.

Los algoritmos se implementan como programas de cómputo en diferentes lenguajes de programación. Un mismo algoritmo se puede implementar en diferentes lenguajes y para diferentes plataformas computacionales. Frecuentemente se expresan en pseudocódigo. En este curso se usa Python, lo que es tan conciso que no hace falta usar pseudocódigo.

Un algoritmo **recursivo** es un algoritmo donde una parte del algoritmo o el algoritmo completo utiliza *a si mismo* como subrutina. En muchos casos es más fácil entender la función de un algoritmo recursivo y también demostrar que funcione correctamente. A un algoritmo que en vez de llamarse a si mismo repite en una manera cíclica el mismo código se le dice *iterativo*. En muchos casos, el pseudocódigo de un algoritmo recursivo resulta más corto que el pseudocógido de un algoritmo parecido pero iterativo para el mismo problema. Cada algoritmo recursivo puede ser convertido a un algoritmo iterativo (aunque no viceversa), aunque típicamente hace daño a la eficiencia del algoritmo hacer tal conversión. Depende del problema cuál manera es más eficiente: recursiva o iterativa.

Como ejemplo, estudiamos la detección de un *palíndromo* que es una cadena de letras (en español, típicamente se ignora los acentos) que se lee igual hacia adelante que hacia atrás, como por ejemplo: “reconocer”.

```
def recursivo(palabra):
 if len(palabra) < 1:
  return True
 elif palabra[0] == palabra[-1]:
  return recursivo(palabra[1:-1])
 else:
  return False

def iterativo(palabra):
 i = 0
 j = len(palabra) - 1
 while i < j:
  if palabra[i] != palabra[j]:
   return False
  i += 1
  j -= 1
```

```
return True
```

Las dos medidas más importantes de la **calidad** de un algoritmo son

- el *tiempo total de computación*, medido por el número de operaciones de cómputo realizadas durante la ejecución del algoritmo, y
- la *cantidad de memoria* utilizada, medida por la cantidad y tipo de las variables requeridas.

La notación para capturar tal información es a través de **funciones de complejidad**. Para un cierto problema computacional, existen típicamente varias si no una cantidad infinita de instancias. Para definir el *tamaño* de una dicha instancia, hay que fijar cuál será la *unidad básica*de tal cálculo. Típicamente se utiliza la cantidad de bits, bytes, variables enteras, etcétera que se necesita ocupar para representar el problema en su totalidad en la memoria de una computadora.

La **función del peor caso** es una función de complejidad $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ tal que para un valor n, el valor $f(n)$ representa el número de operaciones básicas para *el más difícil* de todas las instancias de tamaño n. La única dificultad es **identificar o construir** esa peor instancia.

La meta del análisis de algoritmos es evaluar la calidad de un algoritmo en comparación con otros algoritmos o en comparación a la complejidad del problema o alguna cota de complejidad conocida. Típicamente el conteo de "pasos de computación" falta precisión en el sentido que no es claro que cosas se considera operaciones básicas. Por eso normalmente se caracteriza la calidad de un algoritmo por la **clase de magnitud** de la función de complejidad y no la función exacta misma.

No son interesantes los tiempos de computación para instancias pequeñas, sino instancias grandes. Con una instancia pequeña, normalmente todos los algoritmos producen resultados rápidamente. Por ende se estudia el **crecimiento asintótico**. Para funciones $f : \mathbb{Z}^+ \to \mathbb{R}$ y $g : \mathbb{Z}^+ \to \mathbb{R}$, escribamos

- $f(n) \in \mathcal{O}(g(n))$ si $\exists c > 0$ tal que $|f(n)| \leq c|g(n)|$ para suficientemente grandes valores de n,
- $f(n) \in \Omega(g(n))$ si $\exists c > 0$ tal que $|f(n)| \geq c|g(n)|$ para suficientemente grandes valores de n,
- $f(n) \in \Theta(g(n))$ si $\exists c, c' > 0$ tales que

$$c \cdot |g(n)| \leq |f(n)| \leq c' \cdot |g(n)| \tag{3.9}$$

  para suficientemente grandes valores de n,
- $f(n) \in o(g(n))$ si $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$.
- $\mathcal{O}(f(n))$ es una **cota superior asintótica** al tiempo de ejecución.
- $\Omega(f(n))$ es una **cota inferior asintótica**.
- $\Theta(f(n))$ dice que las dos funciones crecen asintóticamente iguales.

La notación $\mathcal{O}$ y la complejidad asintótica son el tema del capítulo 9 de Graham et al. [1994] La complejidad computacional se discute también en la sección 5.7 de Grimaldi [2017] y el análisis de algoritmos en la sección 5.8.

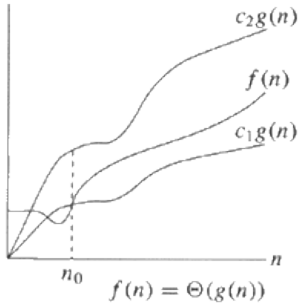


Figura 3.6: La curva de arriba indica que $f(n) \in \mathcal{O}(g(n))$ y la de abajo que también $f(n) \in \Omega(g(n))$, por lo cual se tiene que $f(n) \in \Theta O(g(n))$.

El símbolo $\in$ se reemplaza frecuentemente con una igualdad. Las definiciones de crecimiento asintótica se generalizan para funciones de argumentos múltiples y son *transitivas*:

$$(f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(h(n))) \Rightarrow f(n) \in \mathcal{O}(h(n)) \tag{3.10}$$

y

$$(f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n))) \Rightarrow f(n) \in \Omega(h(n)) \tag{3.11}$$

Como éste aplica para $\Omega(f(n))$ y $\mathcal{O}(f(n))$ los dos, aplica por definición también para $\Theta(f(n))$.

Es fácil formar $\mathcal{O}(f(n))$ de **polinomios** y muchas otras expresiones por observar que en una suma, el término mayor domina el crecimiento:

$$(f(n) \in \mathcal{O}(h(n)) \wedge g(n) \in \mathcal{O}(h(n))) \Rightarrow f(n) + g(n) \in \mathcal{O}(h(n)) \tag{3.12}$$

y

$$g(n) \in \mathcal{O}(f(n)) \Rightarrow f(n) + g(n) \in \mathcal{O}(f(n)). \tag{3.13}$$

Con logaritmos, conviene notar que para cualquier base $b > 0$ y cada $x > 0$ tal que $x \in \mathbb{R}$, aplica que $\log_b(n) \in \mathcal{O}(n^x)$. Cambiando la base de un logaritmo, llegamos a tener

$$\log_a(n) = \frac{1}{\log_b(a)} \log_b(n) \in \Theta(\log_b n), \tag{3.14}$$

porque $\log_b(a)$ es una constante. Entonces no hay necesidad de marcar la base en una expresión de complejidad asintótica con logaritmos. Con funciones exponenciales, tenemos que $\forall x > 1$ y $\forall k > 0$, $n^k \in \mathcal{O}(x^n)$. Es decir, cada polinomial crece asintóticamente más lentamente que cualquiera expresión exponencial.

Para analizar desde un pseudocódigo la complejidad, típicamente se aplica las reglas siguientes:

- Asignación de variables simples toman tiempo $\mathcal{O}(1)$.
- Escribir una salida simple toma tiempo $\mathcal{O}(1)$.
- Leer una entrada simple toma tiempo $\mathcal{O}(1)$.

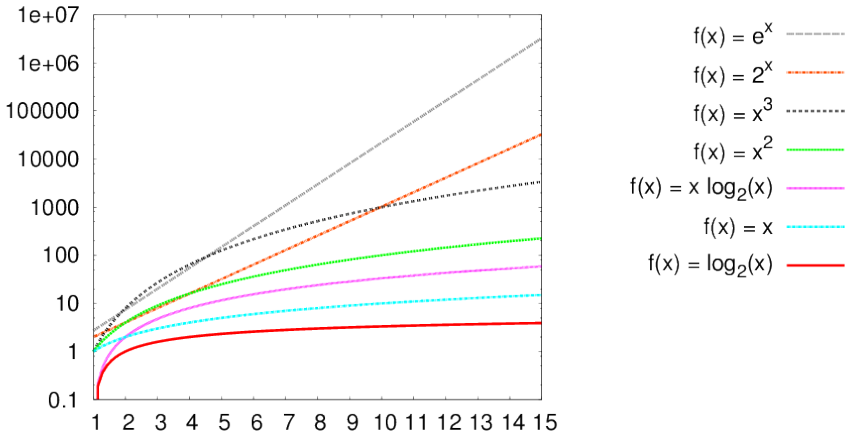


Figura 3.7: Crecimiento de algunas funciones.

Cuadro 3.1: Tiempos de ejecución en función de complejidad asintótica donde mayor a 10^{25} años se denota como $\approx \infty$ y menor a un segundo se denota como ≈ 0.

n	n	$n \log_2 n$	n^2	n^3	$1,5^n$	2^n	$n!$
10	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	4 s
30	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	18 min	10^{25} a
50	≈ 0	≈ 0	≈ 0	≈ 0	11 min	36 a	$\approx \infty$
100	≈ 0	≈ 0	≈ 0	1 s	12,892 a	10^{17} a	$\approx \infty$
1,000	≈ 0	≈ 0	1 s	18 min	$\approx \infty$	$\approx \infty$	$\approx \infty$
10,000	≈ 0	≈ 0	2 min	12 d	$\approx \infty$	$\approx \infty$	$\approx \infty$
100,000	≈ 0	2 s	3 h	32 a	$\approx \infty$	$\approx \infty$	$\approx \infty$
1,000,000	1 s	20 s	12 d	31,710 a	$\approx \infty$	$\approx \infty$	$\approx \infty$

- Si las complejidades de una sucesión de instrucciones $I_1, I_2, \ldots, I_k$ son respectivamente $f_1, f_2, \ldots, f_k$, la complejidad total de la sucesión es

$$\mathcal{O}(f_1 + f_2 + \ldots + f_k) = \mathcal{O}(\text{máx}\{f_1, \ldots, f_k\}) \tag{3.15}$$

  siempre y cuando k **no dependa** del tamaño de la instancia.
- La complejidad de una cláusula de condición (if) es la suma del tiempo de evaluar la condición y la complejidad de la alternativa ejecutada.
- La complejidad de una repetición (`while`, `for`, etc.) es $\mathcal{O}(k(f_t + f_o))$, donde k es el número de veces que se repite, f_t es la complejidad de evaluar la condición de terminar y f_o la complejidad de la sucesión de operaciones de las cuales consiste una repetición.
- La complejidad de tiempo de una **llamada de subrutina** es la suma del tiempo de calcular sus parámetros, el tiempo de asignación de los parámetros y el tiempo de ejecución de las instrucciónes.
- Operaciones aritméticas y asignaciones que procesan listados o conjuntos tienen complejidad lineal en el tamaño su entrada.

La complejidad de programas recursivos típicamente involucra la solución de una **ecuación diferencial**. El método más simple es adivinar una solución y verificar si está bien la adivinanza.

3.4. Optimización combinatoria

Para **problemas de optimización**, una instancia está compuesta por

- un conjunto de *configuraciones*,
- un conjunto de *restricciones*, y
- una *función objetivo* que asigna un valor (real) a cada instancia.

Optimización se discute en el capítulo 13 de Grimaldi [2017].

Si las configuraciones son discretas, el problema es **combinatorial**.

La tarea en problemas de optimización es identificar cuál de las configuraciones **factibles** (o sea, las que cumplen con todas las restricciones) tiene el **mejor** valor de la función objetivo. Depende del problema si el mejor valor es el **mayor** (problema de *maximización* o el **menor** (problema de *minimización*). A la configuración factible con el mejor valor se llama la **solución óptima** de la instancia.

El caso ponderado del ciclo de Hamilton de costo mínimo tiene otro nombre:

Definición 7. *Problema del viajante (TSP)*
Entrada*: un grafo ponderado $\mathcal{G} = (V, E)$ con pesos en las aristas y una constante c.*
Pregunta*: ¿existe un ciclo C en $\mathcal{G}$ tal que C visite cada vértice exactamente una vez y que la suma de los pesos de las aristas de C sea $\leq c$?*

Definición 8. k-Coloreo
Entrada*: un grafo no dirigido $\mathcal{G} = (V, E)$ y un entero $k > 0$.*
Pregunta*: ¿existe una asignación de colores a los vértices de V así que ningún par de vértices $v, u \in V$ tal que $\{v, u\} \in E$ tenga el mismo color?*

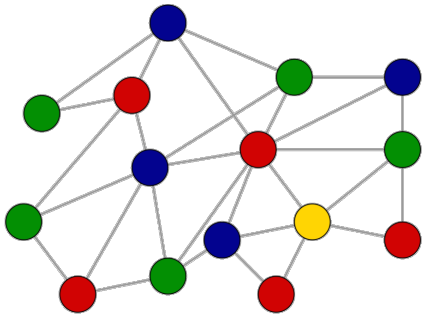


Figura 3.8: Un ejemplo de un coloreo.

Coloreo se discute en la sección 10.6 de Grimaldi [2017]; hay un ejemplo en la figura 3.8.

Definición 9. Isomorfismo de subgrafos
Entrada*: un grafo no dirigido $\mathcal{G} = (V, E)$ y otro grafo $\mathcal{G}'$.*
Pregunta*: ¿si $\mathcal{G}$ contiene un subgrafo isomórfico con $\mathcal{G}'$?*

Definición 10. Problema de la mochila (Knapsack)
Entrada*: una lista de N diferentes artículos $\varphi_i \in \Phi$ y cada objeto tiene una* utilidad $\nu(\varphi_i)$ *y un* peso $\omega(\varphi_i)$.
Pregunta*: ¿Qué conjunto $M \subseteq \Phi$ de artículo debería uno elegir para tener un valor total por lo menos k si tiene una mochila que solamente soporta peso hasta un cierto límite superior Ψ. Es decir, con la restricción*

$$\Psi \geq \sum_{\varphi \in M} \omega(\varphi), \tag{3.16}$$

se aspira maximizar la utilidad total

$$\sum_{\varphi \in M} \nu(\varphi) \geq k. \tag{3.17}$$

El problema de la mochila es **NP-completo**, igual como el TSP (más adelante veremos qué quiere decir eso), lo que se demuestra por un problema de conjuntos (cubierto exacto, inglés: exact cover). Sin embargo, cada instancia del problema de la mochila se puede resolver en tiempo $\mathcal{O}(N \cdot \Psi)$.

Definamos variables auxiliares $V(w, i)$ que es el valor total máximo posible seleccionando algunos entre los primeros i artículos así que su peso total es exactamente w. Cada uno de los $V(w, i)$ con $w = 1, \ldots, \Psi$ y $i = 1, \ldots, N$ se puede calcular a través de la ecuación recursiva siguiente:

$$V(w, i+1) = \text{máx}\{V(w, i), v_{i+1} + V(w - w_{i+1}, i)\} \tag{3.18}$$

donde $V(w, 0) = 0$ para todo w y $V(w, i) = -\infty$ si $w \leq 0$.

Podemos calcular en tiempo constante un valor de $V(w, i)$ conociendo algunos otros y en total son $N\Psi$ elementos, por lo cual su tiempo de ejecución es $\mathcal{O}(N \cdot \Psi)$. La respuesta de la problema de decisión es "sí" únicamente en el caso que algún valor $V(w, i)$ en el cuadro sea mayor o igual a k.

```
peso_permitido = 25
objetos = ((5, 10), (8, 12), (4, 24), (12, 30), \
           (5, 7), (2, 8), (1, 3))
peso_total = sum([objeto[0] for objeto in objetos])
valor_total = sum([objeto[1] for objeto in objetos])
if peso_total < peso_permitido: # cabe todo
    print('mejor valor es', valor_total, 'con un peso', peso_total)
else:
    cantidad = len(objetos)
    V = dict()
    for w in range(peso_permitido + 1):
        V[(w, 0)] = 0
    for i in range(0, cantidad):
        (peso, valor) = objetos[i - 1]
        for w in range(peso_permitido + 1):
            cand = V.get((w - peso, i), -float('inf')) + valor
            V[(w, i + 1)] = max(V[(w, i)], cand)
    mejor_valor = max(V.values())
    peso_de_mejor = max(V.keys(), key = (lambda k: V[k]))[0]
    print('mejor valor es', mejor_valor, 'con un peso', peso_de_mejor)
```

A un algoritmo donde la cota de tiempo de ejecución es polinomial en los enteros de la entrada y no sus logaritmos se llama un algoritmo **pseudo-polinomial**.

3.4.1. Camarilla y conjunto independiente

Definición 11. Clique
Entrada*: un grafo no dirigido $\mathcal{G} = (V, E)$ y un entero $k > 0$.*
¿existe un subgrafo completo inducido por el conjunto $C \subseteq V$ tal que $|C| = k$?

Definición 12. Independent Set (IdSet
Entrada*: un grafo no dirigido $\mathcal{G} = (V, E)$ y un entero $k > 0$.*
¿existe un subgrafo inducido por el conjunto $I \subseteq V$ tal que $|I| = k$ y que no contenga arista ninguna?

Nota que si C es una camarilla (ver la figura 3.9) en $\mathcal{G} = (V, E)$, C es un conjunto independiente en $\bar{\mathcal{G}}$.

3.4.2. Acoplamientos y cubiertas

Un *acoplamiento* $\mathcal{M} \subseteq E$ es un conjunto de *aristas no adyacentes*. Se dice que un vértice v está *acoplado* si hay una arista incidente a v in $\mathcal{M}$; si no, está *libre*. Hay un ejemplo en la figura 3.10 — los acoplamientos (también llamados emparejamientos) se ven en la sección 13.4 de Grimaldi [2017].

Un acoplamiento *máximo* $\mathcal{M}_{\text{máx}}$ contiene el número máximo posible de aristas (no es necesariamente único), mientras un acoplamiento *maximal* es una donde las aristas $\notin \mathcal{M}$ están adyacentes a por lo menos una arista $\in \mathcal{M}$. Note que máximo $\Rightarrow$ maximal (pero no vice versa).

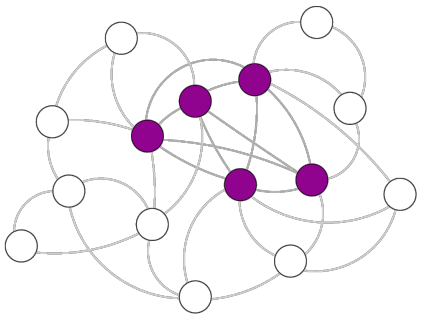


Figura 3.9: Un ejemplo de una camarilla.

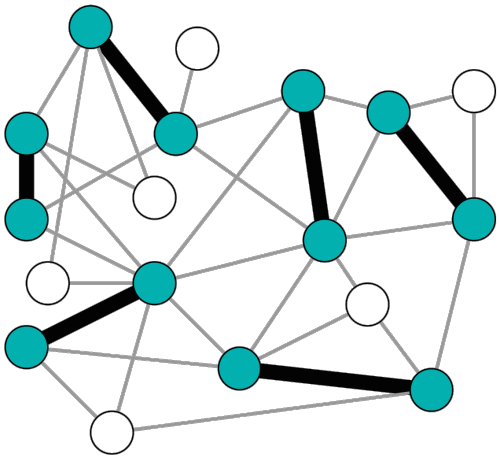


Figura 3.10: Las aristas gruesas forman un acoplamiento, en el cual los vértices coloreados fueron acoplados.

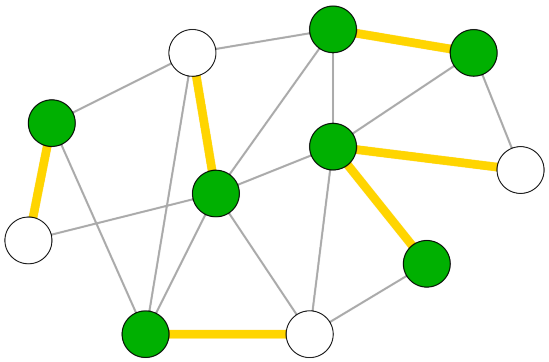


Figura 3.11: Los vértices verdes cubren todas las aristas. Las aristas amarillas cubren todos los vértices.

El *número de acoplamiento* de un grafo es la cardinalidad de su acoplamiento máximo, $|\mathcal{M}_{\text{máx}}|$.¡Al número de vértices libres en este se llama el *déficit*. Un acoplamiento *perfecto* tiene cero deficit y su número de acoplamiento es $\frac{n}{2}$. Cada acoplamiento perfecto es máximo y maximal.

Un *camino alternante* es uno cuyos aristas alternativamente pertenecen y no pertenecen a $\mathcal{M}$. Un *camino aumentante* $\mathcal{A}$ es un camino alternante de un vértice libre v a otro vértice libre u. Un acoplamiento $\mathcal{M}$ es máximo si y sólo si no contiene *ningún* camino aumentante.

Dado un $\mathcal{A}$, podemos intercambiar las aristas en $\mathcal{M}$ para las no en $\mathcal{M}$ para construir $\mathcal{M}'$. Aplica que $|\mathcal{M}'| = |\mathcal{M}| + 1$ y $\mathcal{M}' = \big(\mathcal{M} \setminus (\mathcal{M} \cap \mathcal{A})\big) \cup \big(\mathcal{A} \setminus (\mathcal{M} \cap \mathcal{A})\big)$.

Una *cubierta de aristas* es un conjunto $\mathcal{C}_E$ de aristas así que para cada vértice $v \in V$, $\mathcal{C}_E$ contiene una arista incidente a v. Una *cubierta de vértices* es un conjunto $\mathcal{C}_V$ de vértices así que para cada arista $\{v, w\}$, por lo menos uno de los vértices incidentes está incluido en $\mathcal{C}_V$. La figura 3.11 tiene ejemplos de los dos.

La meta de suele ser encontrar un conjunto de cardinalidad *mínima*. Nota que I es un conjunto independiente en $\mathcal{G}$ si y sólo si $V \setminus I$ es una cubierta de vértices $\mathcal{G}$.

Definición 13. Vertex Cover
Entrada*: un grafo $\mathcal{G} = (V, E)$ no dirigido y un entero $k > 0$.*
Pregunta*: ¿existe un conjunto de vértices $C \subseteq V$ con $|C| \leq k$ tal que $\forall\{v, u\} \in E$, o $v \in C$ o $u \in C$?*

3.4.3. Flujos y cortes

Flujos y cortes son el tema se la sección 13.3 de Grimaldi [2017]. Forman una clase de problemas muy comunes con grafos ponderados (y posiblemente dirigidos) con dos vértices especiales: **fuente** s y **sumidero** t. Se ocupa que el grafo esté conexo en un sentido especial: $\forall v \in V$, existe un camino — dirigido en en caso de grafos dirigidos — del fuente s al sumidero t que pasa por el vértice v. A los pesos de las aristas se les dice **capacidades** $c(v, w) \geq 0$.

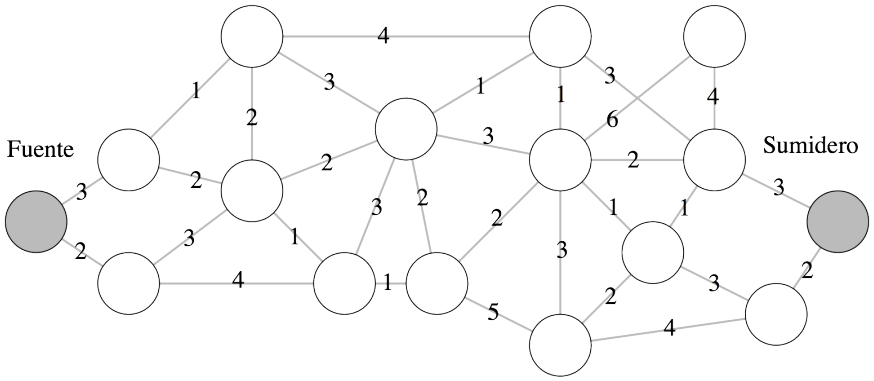


Figura 3.12: Un grafo con capacidades.

Un **flujo positivo** es una función $f : V \times V \to \mathbb{R}$ así que $\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$ y

$$\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v). \tag{3.19}$$

Un **corte** $C \subseteq V$ de $\mathcal{G}$ es una *partición* del conjunto de vértices V en dos conjuntos: C y $V \setminus C$. Al cortar un grafo de flujo, se exige que $s \in C$ y $t \notin C$. La **capacidad** del corte es

$$\sum_{v \in C, w \notin C} c(v, w). \tag{3.20}$$

El corte **mínimo** es un corte cuya capacidad es mínima; el problema de corte mínimo es **polinomial**. Además, la capacidad del corte mínimo entre dos vértices s y t es **igual** al flujo máximo entre s y t. Cuando establecido un flujo en el grafo, la cantidad de flujo que cruza un corte es *igual para cada corte* del grafo. En la mayoría de las aplicaciones de cortes de grafos, los tamaños de los dos "lados" del corte, $|C|$ y $|V \setminus C|$ no suelen ser arbitrarios.

Definición 14. Máxima Bisección
Entrada*: un grafo $\mathcal{G} = (V, E)$ (donde n es par) y un entero $k > 0$.*
Pregunta*: ¿existe un corte C en $\mathcal{G}$ con capacidad $\geq k$ tal que $|C| = |V \setminus C|$?*

Dado un grafo dirigido con capacidades en las aristas y un flujo no-óptimo, se puede aumentar el flujo que cruza un corte desde el lado de s al lado de t o alternativamente por disminuir el flujo desde el lado de t al lado de s. Para empezar, podemos elegir el *flujo cero*, donde el flujo por cada arista es cero — no rompe con ninguna restricción, por lo cual es un flujo factible, aunque no óptimo.

Para aumentar el flujo, buscamos un **camino aumentante** C de s a t en el cual de puede viajar por las aristas según su dirección o **en contra**. Las aristas $\langle v, w \rangle$ incluidas serán tales que si se viaja en la dirección original, aplica que $f(v, w) < c(v, w)$, pero si se viaja en contra, $f(v, w) > 0$. Definamos una función auxiliar

$$\delta(v, w) = \begin{cases} c(v, w) - f(v, w) & \text{si } \langle v, w \rangle \in E, \\ f(v, w), & \text{si } \langle w, v \rangle \in E, \end{cases} \tag{3.21}$$

Sea $\delta = \text{mín}_C\{\delta(v,w)\}$. El flujo se aumenta por añadir δ en todos los flujos que van según la dirección de las aristas en el camino C y **restar** δ de todos los flujos que van en contra en C. Este procedimiento se itera hasta que ya no existan caminos aumentantes. Cuando ya no existe camino aumentante ninguno, el flujo es **maximal**. La eficiencia del método presentado depende de cómo se construye los caminos aumentantes. La mayor eficiencia se logra por elegir siempre el camino aumentante de **largo mínimo**; el algoritmo que resulta es polinomial, $\mathcal{O}(nm^2) = \mathcal{O}(n^5)$. Es posible que hayan más que un camino de largo mínimo — aplicándolos todos al mismo paso resulta en un algoritmo de complejidad asintótica $\mathcal{O}(n^3)$.

Construyamos un *grafo residual* que captura las posibilidades de mejoramiento: $\mathcal{G}_f = (V, E_f)$ del grafo $\mathcal{G} = (V, E)$ con respeto a f tiene aquellas aristas $\{v,w\} \in E$ para las cuales $f(v,w) < c(v,w)$ y aquellas donde $f(v,w) > 0$. La *capacidad de aumento* $c'(v,w)$ vale $c(v,w) - f(v,w)$ si $\{v,w\} \in E$ y $f(w,v)$ si $\{w,v\} \in E$. Nota que cada camino simple entre s y t en el grafo residual $\mathcal{G}_f$ es un camino aumentante de $\mathcal{G}$. El valor de δ es igual al capacidad de aumento mínimo del camino.

Para elegir los caminos aumentantes más cortos en el grafo residual, utilizamos BFS desde s. En subgrafo formado por los caminos cortos en $\mathcal{G}_f$ se llama la **red de capas** (inglés: layered network) $\mathcal{G}'_f$.

Se asigna a cada vértice un valor de "capa" que es su distancia desde s. Solamente vértices con distancias finitas están incluidas: $\{v,w\}$ de $\mathcal{G}_f$ se incluye en $\mathcal{G}'_f$ solamente si el valor de capa de w es el valor de capa de v más uno. En el grafo $\mathcal{G}'_f$, cada camino de s a t tiene el mismo largo.

El mejor aumento sería igual al flujo máximo en $\mathcal{G}'_f$, pero en el peor caso es igual en complejidad al problema original. Entonces construyamos una aproximación: definimos el **flujo mayor** en $\mathcal{G}'_f$ como un flujo que ya no se puede aumentar con caminos que solamente utilizan aristas que "avanzan" hacía t. Definamos también como el **flujo posible** de un vértice es el mínimo de la suma de las capacidades de las aristas que entran y de la suma de las capacidades de las aristas que salen:

$$v_f = \text{mín}\left\{ \sum_{\{u,v\}\in\mathcal{G}'_f} c'(u,v), \sum_{\{v,w\}\in\mathcal{G}'_f} c'(v,w) \right\}. \tag{3.22}$$

El algoritmo es entonces el siguiente:

1. Quitar de $\mathcal{G}'_f$ los vértices con flujo posible cero y sus aristas adyacentes.
2. Identificar el vértice v con flujo posible mínimo.
3. Empujar una cantidad de flujo igual al flujo posible de v de v a t.
4. Retirar flujo a v de sus aristas entrantes por construir caminos de s a v.
5. Repetir el paso anterior hasta que se satisfaga la demanda de flujo de v a t.
6. Actualizar las capacidades de las aristas afectadas.
7. Memorizar el flujo generado y el camino que toma.
8. Actualizar los flujos posibles.
9. Volver a eliminar vértices con flujo posible cero & sus aristas adyacentes.
10. Si s y t quedaron fuera, el flujo construido es el flujo mayor en $\mathcal{G}'_f$.

11. Si todavía están, repetimos el proceso.

```
def camino(s, t, c, f): # construir un camino aumentante
    cola = [s]
    usados = set()
    camino = dict()
    while len(cola) > 0:
        u = cola.pop(0)
        usados.add(u)
        for (w, v) in c:
   if w == u and v not in cola and v not in usados:
    actual = f.get((u, v), 0)
    dif = c[(u, v)] - actual
    if dif > 0:
        cola.append(v)
        camino[v] = (u, dif)
    if t in usados:
        return camino
    else: # no se alcanza
        return None

def ford_fulkerson(c, s, t): # algoritmo de Ford y Fulkerson
    if s == t:
        return 0
    maximo = 0
    f = dict()
    while True:
        aum = camino(s, t, c, f)
        if aum is None:
            break # ya no hay
        incr = min(aum.values(), key = (lambda k: k[1]))[1]
        u = t
        while u in aum:
            v = aum[u][0]
            actual = f.get((v, u), 0) # cero si no hay
            inverso = f.get((u, v), 0)
            f[(v, u)] = actual + incr
            f[(u, v)] = inverso - incr
            u = v
        maximo += incr
    return maximo

# datos tomados de:
# http://www.aduni.org/courses/ (URL cont.)
# algorithms/courseware/handouts/Reciation_09.html
c = {(0, 1): 16, (0, 2): 13, (1, 2): 10, (2, 1): 4, (3, 2): 9, \
(1, 3): 12, (2, 4): 14, (4, 3): 7, (3, 5): 20, (4, 5): 4}
print(ford_fulkerson(c, 0, 5))
```

La complejidad asintótica se determina de la siguiente manera: la construcción de $\mathcal{G}'_f$ toma tiempo $\mathcal{O}(n^2)$, mientras la distancia entre s y t está en el peor caso $\mathcal{O}(n)$. Cada iteración de construcción de una red de capas $\mathcal{G}'_f$ utiliza caminos más largos que el anterior, por lo cual la construcción se repite $\mathcal{O}(n)$ veces. Las operaciones de empujar y retirar flujo son ambas $\mathcal{O}(n)$ y se ejecutan en total $\mathcal{O}(n)$ veces. Entonces, el algoritmo del flujo mayor tiene complejidad asintótica $\mathcal{O}(n^3)$.

El problema de *corte mínimo* (MinCut) es igual al problema del flujo máximo: se resuelve por fijar un vértice s cualquiera y después resolver el flujo máximo entre s y todos los otros vértices. El valor mínimo de los flujos máximos corresponde al corte mínimo del grafo entero. Existen algoritmos polinomiales para el problema de flujo máximo, y solamente repetimos $n-1$ veces su ejecución; entonces el problema corte mínimo pertenece a P.

Definición 15. MaxCut
Entrada*: un grafo $\mathcal{G} = (V, E)$ no dirigido y no ponderado y un entero k.*
Pregunta*: ¿existe un corte en $\mathcal{G}$ con capacidad $\geq k$?*

MaxCut resulta ser NP-completo.

3.5. Tarea 4

Las tareas se responden en línea. Hay que usar el mismo usuario para cada tarea y registrar el usuario con la matrícula correspondiente en la página de resultados del semestre actual para recibir puntos por ellas.

1. Simplifica $\mathcal{O}(\ldots)$ para el polinomio proporcionado en la tarea personalizada; escribe tu respuesta en el formato $\mathrm{O}(x)$ con el exponente adecuado.
2. ¿Cuál de las tres funciones proporcionadas en la tarea personalizada es la más lenta en términos de crecimiento asintótico?
3. Agregando un objeto con el peso y el valor indicados en la tarea personalizada a la instancia del problema de la mochila usado en el código ejemplo, ¿cuánto vale el óptimo si usando límite de peso proporcionada en la tarea personalidaza?
4. Asignando la capacidad indicada a la tarea especificada en la tarea personalizada en la instancia del problema de flujo máximo, ¿cuánto vale el óptimo para $s = 1$ y $t = 4$?
5. ¿Cuánto es el mayor flujo posible dentro de esa instancia modificada del problema de flujo máximo?

3.5.1. Preguntas de verificación

Discute lo siguiente con los compañeros y con la profesora hasta que esté todo claro. Conviene consultar el libro de texto (capítulos y secciones indicados en el material de la unidad en esta página) y hasta buscar por videos en la web. Cuando ya no cabe duda, procede a la última parte del material de estudio.

¿Qué es la diferencia fundamental entre problemas de decisión y las de optimización? Piensa en ejemplos prácticos de los dos tipos de problemas en tu área de ingeniería. ¿Qué es la diferencia entre los conceptos "problema" e "instancia"? ¿Por qué pueden existir muchos algoritmos para un mismo problema?

¿Qué es la diferencia principal entre algoritmos recursivos y algoritmos iterativos? ¿Por qué algunos problemas tienen ambos tipos de algoritmos? ¿Para qué se estudia la complejidad asintótica de algoritmos?

¿Qué significa que una solución es factible? ¿Qué significa que una solución es óptima? ¿Pueden existir múltiples soluciones factibles? ¿Tiene que ser factible una solución para que pueda ser óptima? ¿Aplica también al revés? ¿Pueden existir múltiples soluciones óptimas?

Sin consultar alguna fuente externa, explica en palabras propias y dibujos de apoyo en qué consisten los siguientes problemas: coloreo de grafos, problema de viajante, problema de la mochila, camarilla máxima, conjunto independiente máximo, acoplamiento máximo, cubierta máxima de vértices, cubierta máxima de aristas, flujo máximo, corte mínimo, corte máximo.

3.6. Estructuras de datos

Un paso típico en el diseño de un algoritmo es la elección de una **estructura de datos** apropiada para el problema. Revisemos algunas estructuras que se utiliza en construir algoritmos eficientes.

Un *arreglo* es una estructura capaz de guardar en un orden fijo n elementos. Los índices de las posiciones pueden empezar de cero $a[] = [a_0, a_1, a_2, \ldots, a_{n-1}]$ o uno $b[] = [b_1, b_2, \ldots, b_{n-1}, b_n]$; en Python comienzan desde cero. Se refiere al elemento con índice k como $a[k]$. El *tiempo de acceso* del elemento en posición k en un arreglo es $\mathcal{O}(1)$. Se asume con los elementos guardados en un arreglo no están ordenados por ningún criterio, si no se indica lo contrario de forma explcícita. La complejidad de identificar si un arreglo no ordenado $a[]$ contiene un cierto elemento x es $\mathcal{O}(n)$:

- Habrá que comparar cada elemento $a[k]$ con x.
- Terminar al encontrar igualdad o al llegar al final.
- Si el elemento no está en el arreglo, se necesita n comparaciones.

```
arreglo = (1, 3, 5, 7) # una tupla es como un arreglo ''fijo''
arreglo[1] # se puede consultar, pero no modificar
otro = (''hola'', ``mundo'')
print(otro)
arreglo = 1, 3, 5, 7 # parentesis en realidad son opcionales
arreglo[1] # todo funciona igual
otro = ``hola'', ``mundo'' # lo mismo en este caso
print(otro)
```

Arreglos sirven bien para situaciones donde el número de elementos que se necesita es fijo y conocido o por lo menos que no varie mucho de repente. Si el tamaño no está fijo ni conocido, comúnmente hay que ajustar el tamaño por reservar en la memoria otro arreglo del tamaño deseado y copiar los contenidos del arreglo actual al nuevo. Si los ajustes de la capacidad ocurren con mucha frecuencia, el arreglo no es la estructura adecuada.

Si el arreglo está ordenado en un orden conocido, un algoritmo mejor para encontrar un elemento igual a x es la **búsqueda binaria**: comparar x con el elemento $a[k]$ donde $k = \lfloor \frac{n}{2} \rfloor$ (o $k = \lceil \frac{n}{2} \rceil$, depende de si los índices comienzan de cero o uno). Al $a[k]$ se llama el *elemento pivote*. Si $a[k] = x$, tuvimos suerte y la búsqueda ya terminó. Las otras opciones son:

- Si $a[k] < x$ y el arreglo está en orden creciente, habrá que buscar entre los elementos $a[k+1]$ y el último elemento.
- Si $a[k] > x$ y el arreglo está en orden creciente, habrá que buscar entre el primer elemento y el elemento $a[k-1]$.
- Si $a[k] < x$ y el arreglo está en orden decreciente, habrá que buscar entre el primer elemento y el elemento $a[k-1]$.
- Si $a[k] > x$ y el arreglo está en orden decreciente, habrá que buscar entre los elementos $a[k+1]$ y el último elemento.

Entonces, si i es el primer índice del área donde buscar y j es el último índice del área, **repetimos** el mismo procedimiento de elección de k y comparación con un arreglo

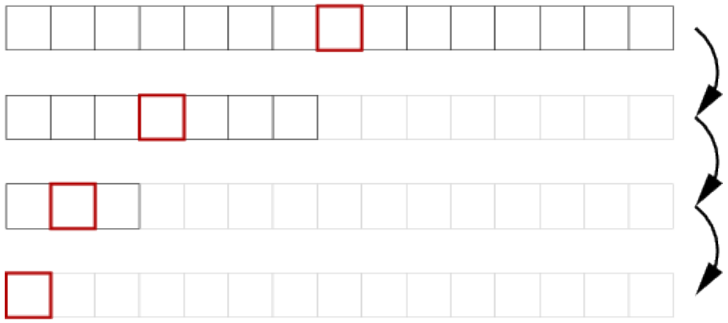


Figura 3.13: Divisiones en la búsqueda binaria (en el peor caso).

$a^{(1)} = [a_i, a_{i+1}, \ldots, a_{j-1}, a_j]$. El resto del arreglo nunca será procesado, que nos ofrece un ahorro. De esta manera, si el contenido siempre se divide en dos partes de aproximadamente el mismo tamaño, la iteración termina cuando la parte consiste de un sólo elemento.

```
def bbinaria(ordenados, buscado):
    n = len(ordenados)
    if n == 0: # no hay nada
        return False
    pos = n // 2 # div. entera
    pivote = ordenados[pos]
    if pivote == buscado:
        return True # encontrado
    elif buscado < pivote: # viene antes del pivote
        return bbinaria(ordenados[: pos], buscado)
    else: # pivote > buscado # viene desp. del pivote
        return bbinaria(ordenados[pos + 1 :], buscado)
```

El peor caso es que el elemento esté en la primera o la última posición del arreglo o que no esté incluido. ¿Cuántas divisiones tiene el peor caso?

El tamaño de la parte que queda para buscar tiene al máximo $\lceil \frac{n}{2} \rceil$ elementos. Al último nivel el tamaño de la parte es uno. Entonces, habrá $\log_2(n)$ divisiones. Cada división contiene una comparación de x con un $a[k]$ y la asignación del nuevo índice inferior y el nuevo índice superior. Por ende son $3\log_2(n)$ operaciones y la complejidad asintótica es $\mathcal{O}(\log n)$.

Listas son estructuras un poco más avanzadas que puros arreglos, como típicamente permiten ajustes naturales de su capacidad. Una lista enlazada (inglés: linked list) consiste de elementos que todos contengan además de su dato, un puntero al elemento siguiente. Si el orden de los elementos no está activamente mantenido, es fácil agregar un elemento en la lista:

- Crear el elemento nuevo.
- Inicializar su puntero del siguiente elemento a nulo.
- Hacer que el puntero del siguiente del último elemento actualmente en la lista punte al elemento nuevo.

Para acceder una lista, hay que mantener un puntero al primer elemento. Si también se mantiene un puntero al último elemento, añadir elementos cuesta $\mathcal{O}(1)$ unidades de tiempo,

mientras solamente utilizando un puntero al comienzo, se necesita tiempo $\mathcal{O}(n)$, donde n es el número de elementos en la lista.

Si uno quiere mantener el orden mientras realizando inserciones y eliminaciones, hay que primero ubicar el elemento anterior al punto de operación en la lista:

- para **insertar** un elemento nuevo v inmediatamente *después* del elemento u actualmente en la lista, hay que ajustar los punteros tal que el puntero del siguiente v.sig de v tenga el valor de u.sig, después de que se cambia el valor de u.sig a puntar a v;
- para **eliminar** un elemento v, el elemento anterior siendo u, primero hay que asignar u.sig := v.sig y después simplemente eliminar v, a que ya no hay referencia de la lista.

Una lista **doblemente enlazada** tiene además en cada elemento un enlace al elemento anterior. Su mantenimiento es un poco más laborioso por tener que actualizar más punteros por operación, pero hay aplicaciones en las cuales su eficacia es mejor. En Python todo lo de "puntero al siguiente" funciona de forma implícita, mientras en lenguajes de más bajo nivel de abstracción como es el ANSI C se manipulan de forma directa explícitamente.

```
datos = [1, 2, 4, 8, 16]
datos[1]
datos[-1] # final de la lista
datos[3] = 4
datos += [30]
nada = [] # lista sin elementos
len(datos)
datos[1:-1] # sublista sin el inicio y el final
```

Con listas, ganamos tamaño dinámico, pero búsquedas y consultas de elementos ahora tienen costo $\mathcal{O}(n)$ mientras con arreglos tenemos acceso en tiempo $\mathcal{O}(1)$ y tamaño "rígido".

Una **pila** (inglés: stack) es una lista especial donde todas las operaciones manipulan el primer elemento de la lista; se añade al frente y remueve del frente y se implementa como una lista enlazada manteniendo un puntero p al primer elemento. (Conviene visualizar una pila de libros en un escritorio para entender la operación: conviene poner libros nuevos encima de los que ya están en la pila y el único libro fácil de quitar es el que esté encima de los demás.)

```
pila = []
inicio = 0
pila.insert(inicio, 'a') # para que agregue al _inicio_ de la lista
pila.insert(inicio, 'b')
pila.insert(inicio, 'c')
print(pila) # viejos al final, nuevos al inicio
sale = pila.pop(inicio) # eliminar del inicio
print(sale) # el elemento eliminado fue c
sale = pila.pop(inicio) # ahora al inicio queda b; lo quitamos
print(pila) # queda puro a en la pila
```

Una **cola** (inglés: queue) es una estructura donde los elementos nuevos llegan al final, pero el procesamiento se hace desde el primer elemento. También colas están fácilmente implementadas como listas enlazadas, manteniendo un puntero al comienzo de la cola y

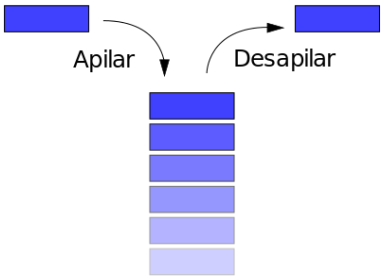


Figura 3.14: La operación de una pila.

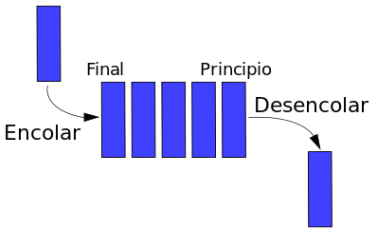


Figura 3.15: La operación de una cola.

otro al final. (Piensa en la forma de hacer fila en bancos para entender la operación de una cola — la figura 3.15 ilustra esto.)

```
cola = []
inicio = 0
cola.append('a') # al final
cola.append('b')
cola.append('c')
cola.append('d')
print(cola)
sale = cola.pop(inicio) # eliminar del inicio
print(sale)
sale = cola.pop(inicio)
print(cola)
```

3.7. Algoritmos de ordenamiento

Para **ordenar** una lista $L = [\ell_1, \ell_2, \ldots, \ell_n]$ en orden creciente, se puede definir un mecanismo `insertar`(L, i, x) que busca desde el comienzo la posición i en la lista L por un elemento $\ell_j \leq x$ hacía la primera posición, tal que $j \leq i$ (puedes probar si te sale implementar algo asín en Python). Al encontrar tal elemento, el elemento x estará insertada en la posición justo después del elemento ℓ_j. Si no se encuentra un elemento así, se inserta x al comienzo de la lista.

Cuadro 3.2: Un ejemplo de ordenamiento de burbuja.

1	2	3	4	5
2	**1**	3	4	5
2	3	**1**	4	5
2	3	4	**1**	5
2	3	4	5	**1**
2	3	4	5	1
3	**2**	4	5	1
3	4	**2**	5	1
3	4	5	**2**	1
3	4	5	1	**2**
3	4	5	2	1
4	**3**	5	2	1
4	5	**3**	2	1
4	5	3	2	1
5	**4**	3	2	1
5	4	3	2	1

3.7.1. Ordinamiento por inserción

El procedimiento de la ordenación empieza con el primer elemento de la lista y progresa con una variable indicadora de posición i hasta el último elemento. Para cada elemento, quitamos ℓ_i de la lista y se utiliza a `insertar`(L, i, x) para volver a guardarlo.

3.7.2. Ordenamiento de burbuja

Hay varios algoritmos para ordenar un arreglo, esto siendo uno de los más básicos y menos eficientes. En inglés se conoce como *bubble sort*.

- Inicia una variable contadora a cero: $c := 0$.
- Comenzando desde el primer elemento, compáralo con el siguiente.
- Si su orden está correcto con respeto a la ordenación deseada, déjalos así.
- Si no están en orden, con una variable auxiliar t, intercambia sus valores y incrementa a la contadora, $c := c + 1$.
- Avanza a comparar el segundo con el tercero, repitiendo el mismo procesamiento, hasta llegar al final del arreglo.
- Si al final, $c \neq 0$, asigna $c := 0$ y comienza de nuevo.
- Si al final $c = 0$, el arreglo está en la orden deseada.

3.7.3. Ordenamiento por selección

Dado un arreglo de n elementos, podemos ordenar sus elementos en el orden creciente con el siguiente procedimiento:

1. Asigna $i :=$ primer índice del arreglo $a[]$.
2. Busca entre i y el fin del arreglo el elemento menor.
3. Denote el índice del mejor elemento por k y guarda su valor en una variable auxiliar $t := a[k]$.
4. Intercambia los valores de $a[i]$ y $a[k]$: $a[k] := a[i]$, $a[i] := t$.
5. Incrementa el índice de posición actual: $i := i + 1$.
6. Itera hasta que i esté en el fin del arreglo.

Es una operación $\mathcal{O}(n)$ combinar dos partes ordenadas en un solo arreglo ordenada bajo el mismo criterio. Por ejemplo, si la entrada son dos partes A y B de números enteros ordenaros del menor a mayor, el arreglo combinado se crea por

- leer el primer elemento de A,
- leer el primer elemento de B,
- añadir el **mínimo** de estos dos en el arreglo nuevo C,
- re-emplazar la variable auxiliar utilizada por leer el siguiente elemento de su arreglo de origen.

3.7.4. Ordinamiento por fusión

Ordenamiento **por fusión** (inglés: mergesort) funciona por divisiones parecidas a las de la búsqueda binaria. El contenido está dividido a dos partes del mismo tamaño (más o menos un elemento): la primera parte tiene largo $\lfloor \frac{n}{2} \rfloor$ y la segunda tiene largo $n - \lfloor \frac{n}{2} \rfloor$. Ambas partes están divididos de nuevo hasta que contengan k elementos, $k \geq 1 \ll n$. Al llegar al nivel donde el contenido por procesar tiene k elementos, se utiliza otro algoritmo para ordenarlo; opcionalmente se podría fijar $k = 1$. Dos subarreglos $b_\ell[]$ y $b_r[]$ ordenados están combinados con uso de memoria auxiliar a un arreglo $b[]$:

1. $i_\ell := 0$, $i_r := 0$ y $i := 0$.
2. Si $b_\ell[i_\ell] < b_r[i_r]$, $b[i] := b_\ell[i_\ell]$ y después $i_\ell := i_\ell + 1$ y $i := i + 1$.
3. Si $b_\ell[i_\ell] \geq b_r[i_r]$, $b[i] := b_r[i_r]$ y después $i_r := i_r + 1$ y $i := i + 1$.
4. Cuando i_ℓ o i_r pasa afuera del subarreglo que corresponde, copia lo que queda del otro al final.
5. Mientras todavía quedan elementos en los dos, repite la elección del elemento menor a guardar en $b[]$.

3.7.5. Ordenamiento rápido

La idea del ordenamiento rápido (inglés: quicksort) es también dividir el contenido, pero no necesariamente en partes de tamaño igual. Se elige un **elemento pivote** $a[k]$ según algún criterio (existen varias opciones como elegirlo), y divide el contenido de entrada $a[]$ en dos partes: una parte donde todos los elementos son menores a $a[k]$ y otra parte donde son mayores o iguales a $a[k]$ por escanear todo el contenido una vez. La eficiencia del algoritmo depende de la calidad del pivote.

El escaneo se puede implementar con **dos índices** moviendo el el arreglo, uno del co-

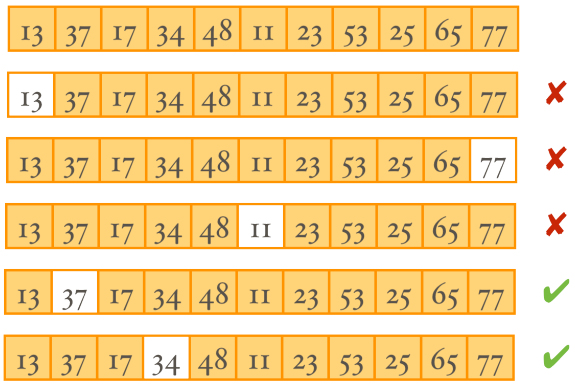


Figura 3.16: Ejemplos de malos y buenos pivotes

mienzo y otro del final. El índice del comienzo busca por *el primer elemento con un valor mayor o igual al pivote*, mientras el índice de atrás mueve en contra buscando por *el primer elemento menor al pivote*. Al encontrar elementos tales antes de cruzar un índice contra el otro, se *intercambia* los dos valores y continua avanzando de las mismas posiciones de los dos índices. Al *cruzar*, se ha llegado a la posición donde cortar el arreglo a las dos partes.

También se puede realizar de forma recursiva: se repite el mismo procedimiento con cada uno de las dos partes.Así nivel por nivel resultan ordenadas los subarreglos, y por el procesamiento hecha ya en los niveles anteriores, todo el arreglo resulta ordenado.

No todas las herramientas computacionales vienen con mecanismos de ordenamiento ya implementados, pero Python sí ya los tiene.

```
a = (2, 6, 9, 12, 10, 20, 120, 391, 10, 209, 1730, 284, 1, 42)
print(a)
print(a[::-1]) # el orden contrario
print(a[::2]) # cada dos pasos desde el inicio
print(a[::-3]) # cada tres pasos desde el final
b = [2, 6, 9, 12, 10, 20, 120, 391, 10, 209, 1730, 284, 1, 42]
b.sort() # esto funciona igual con una lista en lugar de arreglo
b.sort() # ordenar
print(b)
b[2] = 12
print(b[::-1])
print(b[::2])
print(b[::-3])
```

3.8. Estructuras ramificadas

Un **árbol** es un grafo conexo simple no cíclico — de n vértices etiquetados $1, 2, \ldots, n$ se puede guardar en un arreglo $a[]$ de n posiciones, donde el valor de $a[i]$ es la etiqueta del vértice padre del vértice i. Otra opción es guardar en cada elemento un puntero al

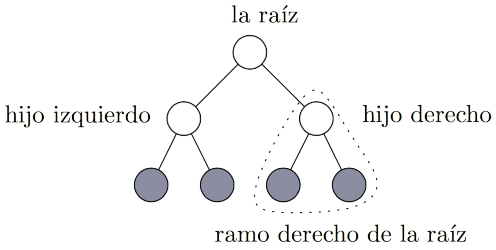


Figura 3.17: Un ejemplo de un arbol.

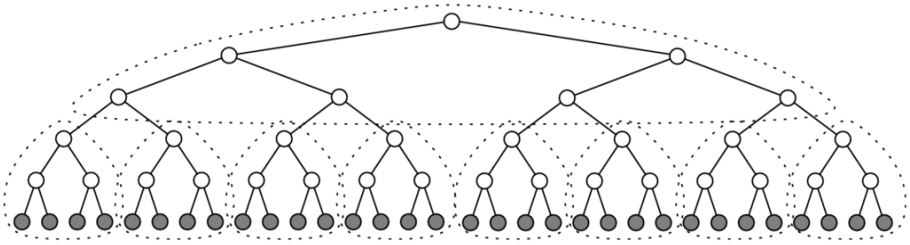


Figura 3.18: La división de un árbol en ramos separados.

vértice padre (y posiblemente del padre una estructura de punteros a sus hijos). árboles son "listas estructuradas"; se utilizan como índices de bases de datos. Cada elemento consiste de una *clave*(no necesariamente único) y un dato. árboles permiten realizar eficientemente inserciones, eliminaciones, y búsquedas.

Es necesario que exista un *orden* sobre el espacio de las claves de los elementos. En un **árbol binario**, cada vértice que no es una hoja tiene al máximo dos vértices hijos: su hijo **izquierdo** y su hijo **derecho**. Si por máximo un vértice cuenta con solamente un hijo, se dice que el árbol está **lleno**.

Su uso como índices es relativamente fácil también para bases de datos muy grandes, como diferentes ramos y partes del árbol se puede guardar en diferentes **páginas** de la memoria física de la computadora; la figura 3.18 ilustra este tipo de división.

El problema con árboles binarios es que su forma depende del orden de inserción de los elementos y en el peor caso puede reducir a casi una lista. A esto se le llama *imbalance* (figura 3.19.

Existen muchas herramientas en línea que visualizan operaciones en árboles; por ejemplo `http://btv.melezinek.cz/binary-search-tree.html`. Una implementación de un árbol simple en Python es el siguiente (guárdalo en un **archivo** `arbol.py`):

```
class Nodo:

 def __init__(self):
  self.contenido = None
  self.izquierdo = None
  self.derecho = None
```

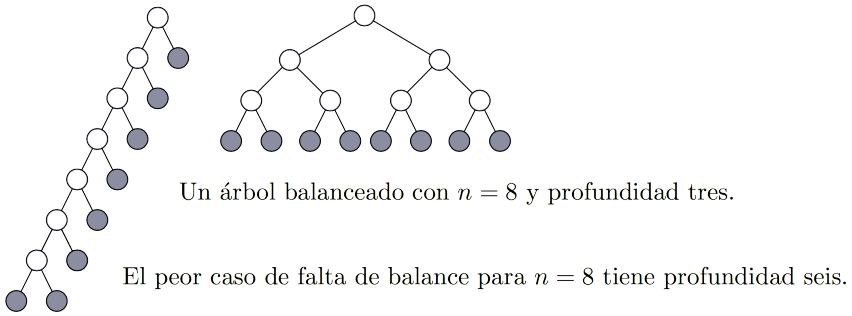


Figura 3.19: Ejemplo de un imbalance en un árbol.

```
 def __str__(self):
  if self.contenido is None:
   return ''
  else: # lo de %s es esencialmente lo mismo que lo de {:s} en .format()
   return '%s [%s | %s]' % (self.contenido, self.izquierdo, self.derecho)

 def agrega(self, elemento):
  if self.contenido is None:
   self.contenido = elemento
  else:
   if elemento < self.contenido:
 if self.izquierdo is None:
  self.izquierdo = Nodo()
 self.izquierdo.agrega(elemento)
   if elemento > self.contenido:
 if self.derecho is None:
  self.derecho = Nodo()
 self.derecho.agrega(elemento)

class Arbol:

 def __init__(self):
  self.raiz = None

 def __str__(self):
  return str(self.raiz)

 def __repr__(self):
  return str(self.raiz)

 def agrega(self, elemento):
  if self.raiz is None:
```

```
    self.raiz = Nodo()
  self.raiz.agrega(elemento)
```

Para usarlo, procedemos como con los grafos anteriormente (ahora se han agregado subrutinas para que Python sepa imprimir el contenido de un árbol):

```
from arbol import Arbol
a = Arbol()
a.agrega(50)
a.agrega(18)
a.agrega(74)
a.agrega(7)
a.agrega(22)
print(a)
a
```

3.8.1. Búsqueda de una clave

Utilicemos en los ejemplos enteros positivos como las claves. Para buscar la hoja con clave i, se empieza del raíz del árbol y progresa recursivamente al hijo izquierdo si el valor del raíz es *mayor* a i y al hijo derecho si el valor es *menor o igual* a i. Cuando la búsqueda llega a una hoja, se evalúa si el valor de la hoja es i o no. Si no es i, el árbol no contiene la clave i en ninguna parte.

```
def ubicar(nodo, buscado):
    if nodo.contenido == buscado:
        return True
    if buscado < nodo.contenido and nodo.izquierdo is not None:
        return ubicar(nodo.izquierdo, buscado)
    if buscado > nodo.contenido and nodo.derecho is not None:
       return ubicar(nodo.derecho, buscado)
    return False

ubicar(a.raiz, 15) # probar
ubicar(a.raiz, 22)
```

El árbol está **balanceado** si el largo máximo es k, el largo mínimo tiene que ser mayor o igual a $k-1$. En este caso, el número de *hojas* del árbol $n = 2^k \leq n < 2^{k+1}$.

La *altura* de un ramo de un vértice v, es decir, un subárbol la raíz de cual es v es la altura de v. La altura del árbol entero es la altura de su raíz.

$$\mathcal{A}(v) = \begin{cases} 1, \text{ si } v \text{ es una hoja} \\ \max\{\mathcal{A}(\text{izq}(t)), \mathcal{A}(\text{der}(t))\} + 1, \text{ si } v \text{ es de ruteo.} \end{cases} \tag{3.23}$$

La *profundidad* de cada vértice del árbol:

$$\mathcal{D}(v) = \begin{cases} 0, \text{ si } v \text{ es la raíz,} \\ \mathcal{D}(v.\mathcal{P}) + 1, \text{ en otro caso.} \end{cases} \tag{3.24}$$

La profundidad del árbol entero es simplemente $\max_v \mathcal{D}(v)$. Aplica que $\mathcal{D} = \mathcal{A} - 1$.

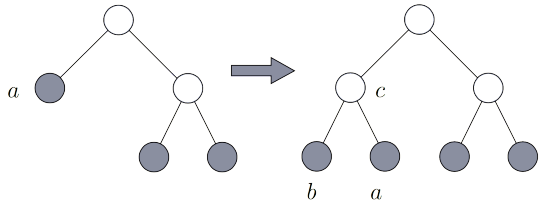


Figura 3.20: Insertando una clave

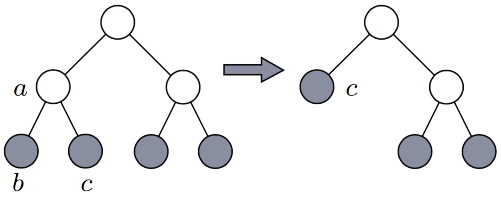


Figura 3.21: Eliminando una clave

Para **insertar** un elemento nuevo al árbol de índice, *primero hay que buscar* la ubicación de la clave del elemento. Llegando a la hoja v_h donde debería estar la clave, hay que *crear un vértice de ruteo* v_r nuevo. La hoja v_h va a ser uno de los hijos del vértice de ruteo y el otro hijo será un vértice nuevo v_n creado para el elemento que está insertado. Por ejemplo, el elemento menor de v_h y v_n será el hijo izquierdo y el mayor el hijo derecho. El valor del vértice de ruteo v_r así creado será igual al valor de su hijo derecho. La figura 3.20 ilustra el procedimiento.

Para **eliminar** un elemento del árbol, hay que primero ubicar su posición y después *eliminar además de la hoja su vértice de ruteo* v_r y mover el otro vértice hijo del vértice de ruteo v_h a la posición que ocupó v_r. La figura 3.21 ilustra esto.

Las operaciones de insertar y remover claves modifican la forma del árbol. La garantía de tiempo de acceso $\mathcal{O}(\log n)$ está solamente válida a árboles *balanceados*.

Un árbol está *perfectamente balanceado* si su estructura es óptima con respeto al largo del camino de la raíz a cada hoja: todas las hojas están en el mismo nivel, es decir, el largo máximo de tal camino es igual al largo mínimo de tal camino sobre todas las hojas. Esto es solamente posible cuando el número de hojas es 2^k para $k \in \mathbb{Z}^+$, en que caso el largo de todos los caminos desde la raíz hasta las hojas es exactamente k.

Necesitamos operaciones para "recuperar" la forma balanceada después de inserciones y eliminaciones de elementos, aunque no cada operación causa una falta de balance en el árbol. Estas operaciones se llaman **rotaciones**. La rotación adecuada se elige según las alturas de los ramos que están fuera de balance, es decir, tienen diferencia de altura mayor o igual a dos. Si se balancea después de *cada inserción y eliminación* siempre y cuando es necesario, la diferencia será siempre menor o igual a dos.

Sean los hijos de t que están fuera de balance u y v. Consideramos primero el caso que $\mathcal{A}(u) \geq \mathcal{A}(v) + 2$. Hay dos opciones: si $\mathcal{A}(A) \geq \mathcal{A}(B)$, toca una rotación simple a la derecha, pero si $\mathcal{A}(A) < \mathcal{A}(w)$, toca una rotación doble izquierda-derecha.

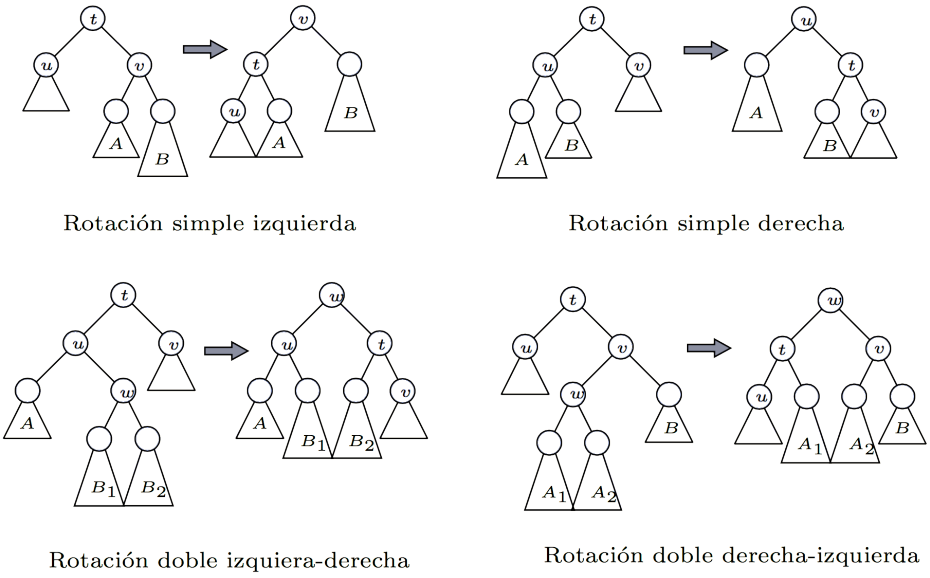


Figura 3.22: Rotaciones

En el caso que $\mathcal{A}(u) \leq \mathcal{A}(v) - 2$, si $\mathcal{A}(A) \geq \mathcal{A}(B)$, se hace una rotación simple a la izquierda, mientras si $\mathcal{A}(B) < \mathcal{A}(w)$, corresponde una rotación doble derecha-izquierda.

Con esas rotaciones, ilustradas en la figura 3.22, ninguna operación va a aumentar la altura de un ramo, pero la puede reducir por una unidad. La manera típica de encontrar el punto de rotación t es regresar hacía la raíz después de haber operado con una hoja para verificar si todos los vértices en camino todavía cumplan con la condición de balance.

3.8.2. Montículos

Un **montículo** (inglés: heap) es una estructura compuesta por árboles. Existen muchas variaciones de montículos. La implementación típica de un montículo se basa de árboles, mientras árboles se puede guardar en arreglos. Entonces, las implementaciones se basan en arreglos o el uso de elementos enlazados.

3.9. Almacenaje y manipulación de grafos

Una representación para un grafo $\mathcal{G} = (V, E)$ de n vértices etiquetados $1, 2, 3, \ldots, n$ es guardar su matriz de adyacencia como un arreglo bidimensional $n \times n$. Esto es eficiente sólo cuando $\mathcal{G}$ es medianamente denso; si estuviera muy denso, sería mejor guardar su complemento con listas de adyacencia. La matriz ocupa $\mathcal{O}(n^2)$ elementos y si m es mucho menor que n^2, la mayoría del espacio reservado tiene el valor cero.

Para guardar en su lugar *listas de adyacencia*, se puede usar un arreglo $a[]$ donde cada elemento de cuál es una lista de largo dinámico. La idea es que $a[i]$ contenga (las etiquetas

de) los vecinos del vértice i. El tamaño de la estructura de listas de adyacencia es $\mathcal{O}(n+m) \leq \mathcal{O}(m) = \mathcal{O}(n^2)$ (aunque en muchos casos es mucho menor).

Un **recorrido** es un un método sistemático para *visitar cada vértice* (por lo menos una vez). Una **búsqueda** es un recorrido cuyo propósito es *encontrar* un vértice del $\mathcal{G}$ que tenga una cierta propiedad. Algoritmos de recorrido y/o búsqueda comúnmente utilizan **colas** o **pilas** como estructuras auxiliares que guian el proceso. Sus usos incluyen los siguientes:

- Búsqueda de vértices.
- Construcción de caminos.
- Computación distancias.
- Detección de ciclos.
- Identificación de los componentes conexos.

3.9.1. Búsqueda en profundidad DFS

Dado $\mathcal{G}$ y un vértice inicial $v \in V$:

1. Crea una cola vacía $\mathcal{L}$.
2. Asigna $u := v$.
3. Marca u visitado.
4. Añade los vecinos **no marcados** de v al *comienzo* de $\mathcal{L}$.
5. Quita del comienzo de $\mathcal{L}$ todos los vértices marcados.
6. Si $\mathcal{L}$ está vacía, termina.
7. Asigna $u :=$ el *primer* vértice en $\mathcal{L}$.
8. Quita el primer vértice de $\mathcal{L}$ y llámalo v.
9. Continua del segundo paso.

```
def dfs(actual, adj):
 cola = [actual]
 visitados = set()
 while len(cola) > 0:
  actual = cola.pop(0)
  visitados.add(actual)
  for vecino in adj[actual]:
   if vecino not in visitados:
 cola.append(vecino)
  return visitados

E = {('a', 'b'), ('a', 'c'), ('b', 'd'), \
     ('c', 'e'), ('d', 'f'), ('e', 'f')}
adj = dict() # un mapeo de conjuntos de adyacencia
for arista in E:
 for vertice in arista:
  if vertice not in adj:
   adj[vertice] = set()
```

```
 (u, v) = arista
 adj[v].add(u)
 adj[u].add(v) # no dirigido; vale en ambos sentidos

componente = dfs('a', adj)
```

De forma recursiva es un poco más breve:

```
# suponiendo que adj es el mismo que antes

def dfs_r(actual, adj, visitados):
    if actual in visitados:
        return
    visitados.add(actual)
    for vecino in adj[actual]:
        dfs_r(vecino, adj, visitados)

comp = set()
dfs_r('a', adj, comp)
print(comp)
```

Un recorrido DFS puede progresar en varias maneras. El orden de visitas a los vértices depende de cómo se elige a cuál vecino se va. Las opciones son "visitar" antes o después de llamar la subrutina para los vecinos; si la visita se realiza *antes* de la llamada recursiva se llama el *preorden*. En el caso de visitar *después*, se llama *postorden*.

```
def dfs_imprime(actual, adj, visitados, orden=None):
    if actual in visitados:
        return
    visitados.add(actual)
       if orden == 'pre':
     print(actual)
    for vecino in adj[actual]:
        dfs_imprime(vecino, adj, visitados, orden)
    if orden == 'post':
        print(actual)

dfs_imprime('a', adj, set(), pre')
dfs_imprime('a', adj, set(), 'post')
dfs_imprime('a', adj, set())
```

La complejidad asintótica del recorrido DFS es $\mathcal{O}(n+m)$: cada arista es procesada por máximo una vez "de ida" y otra "de vuelta", mientras cada vértice es procesado una vez; los "ya marcados" no serán revisitados.

DFS produce una *clasificación* de las aristas: las aristas de árbol son las aristas por las cuales progresa el procedimiento, es decir, en la formulación recursiva, w fue visitado por una llamada de v o vice versa. Estas aristas forman un *árbol de expansión* del componente conexo del vértice de inicio. Depende de la manera en que se ordena los vecinos que habrá que visitar cuáles aristan serán aristas de árbol.

- v es el *padre* (directo o inmediato) de w si v lanzó la llamada recursiva para visitar a w.
- Si v es el padre de w, w es *hijo* v.
- Cada vértice, salvo que el vértice de inicio, que se llama la *raíz*, tiene un vértice padre único.
- El número de hijos que tiene un vértice puede variar.

Se dice que v es un *antepasado* de w si existe una sucesión de vértices $v = u_1, u_2, \ldots, u_k = w$ tal que u_i es el padre de u_{i+1}. En ese caso, w es un *descendientes* de v.

- $k = 2$: v es el padre de w, v es el antepasado *inmediato* de w y w es un descendiente inmediato de v.
- La raíz es un antepasado de todos los otros vértices.
- Los vértices sin descendientes son *hojas*.
- Una *arista procedente* conectan un antepasado a un descendiente *no inmediato*.
- Una *arista retrocedente* conecta un descendiente a un antepasado *no inmediato*.
- Una *arista transversa* conecta un vértice a otro tal que no son ni antepasados ni descendientes uno al otro — están de diferentes *ramos* del árbol.
- Las aristas procedentes y retrocedentes son lo mismo en un grafo no dirigido.

El *nivel* de la raíz es cero, y el nivel de v es el nivel de su padre más uno. La *altura* de cada hoja es cero y la altura de un nodo interior v es el máximo de las alturas de sus hijos más uno. El *subárbol* de v es el árbol que es un subgrafo del árbol de expansión donde v es la raíz; solamente vértices que son descendientes de v están incluidos además de v mismo.

Se puede utilizar un algoritmo de recorrido para determinar los *componentes conexos* de un grafo. Un recorrido efectivamente explora todos los caminos que pasan por v. Por iniciar DFS en el vértice v, el conjunto de vértices visitados por el recorrido corresponde al componente conexo de v. Si el grafo tiene vértices que no pertenecen al componente de v, elegimos uno de esos vértices u y corremos DFS desde u para encontrar el componente conexo que contiene a u. Iterando se determina todos los componentes conexos.

En un grafo conexo, podemos etiquetar las aristas según un recorrido DFS:

- Asignamos al vértice inicial la etiqueta "uno".
- Siempre al visitar a un vértice por la primera vez, le asignamos una etiqueta numérica uno mayor que la última etiqueta asignada.
- Los vértices llegan a tener etiquetas únicas en $[1, n]$.
- La etiqueta obtenida es *el número de inicio* $I(v)$.

Asignamos otra etiqueta a cada vértice tal que la asignación ocurre cuando todos los vecinos han sido recurridos, empezando de 1. Así el vértice de inicio tendrá la etiqueta n. Estas etiquetas son los *números de final* $F(v)$. Las $I(v)$ definen el orden previo del recorrido, mientras las $F(v)$ definen el orden posterior del recorrido.

Al ejecutar el algoritmo iterativo, basta con mantener las dos contadores. Al ejecutar el algoritmo recursivo, hay que pasar los valores de las últimas etiquetas asignadas en la llamada recursiva (iniciar con cero y cero).

Una arista $\{v, u\}$ (dirigida) es

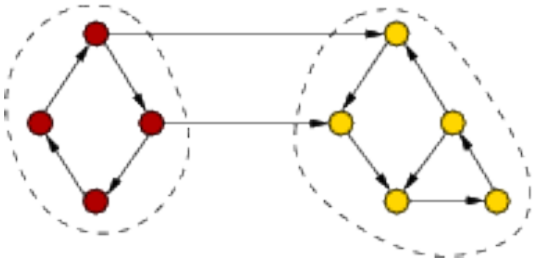


Figura 3.23: Componentes fuertemente conexos

- una *arista de árbol* si el recorrido primero llegó a u desde v,
- una *arista retrocedente* si y sólo si $(I(u) > I(v)) \wedge (F(u) < F(v))$,
- una *arista transversa* si y sólo si $(I(u) > I(v)) \wedge (F(u) > F(v))$, y
- una *arista procedente* si v es un antepasado de u (esto solamente tiene sentido para grafos dirigidos).

Un grafo no dirigido es k-conexo si de cada vértice hay por lo menos k caminos distintos a cada otro vértice. El requisito de ser distinto puede ser de parte de

- o los vértices tal que no pueden pasar por los mismos vértices ningunos de los k caminos (inglés: vertex connectivity),
- o de las aristas tal que no pueden compartir ninguna arista los caminos (inglés: edge connectivity).

Un grafo dirigido es *fuertemente conexo* si de cada uno de sus vértices existe un camino dirigido a cada otro vértice. Los componentes fuertemente conexos de un grafo son los subgrafos maximales fuertemente conexos.

Los componentes fuertemente conexos de $\mathcal{G} = (V, E)$ determinan una *partición de los vértices* de $\mathcal{G}$ a las clases de equivalencia según la relación de clausura reflexiva y transitiva de la relación de aristas E.

Las aristas entre los componentes fuertemente conexos determinan una orden parcial en el conjunto de componentes. Ese orden parcial se puede aumentar a un orden lineal por un algoritmo de *ordenación topológica*.

3.9.2. Ordenación topológica

Cuando uno realiza un DFS, los vértices de un componente conexo se quedan en el mismo ramo del árbol de expansión. El vértice que queda como la raíz del ramo se dice la raíz del componente. Nuestra meta ahora es encontrar las raíces de los componentes según el orden de su $F(v)$.

Al llegar a una raíz v_i, su componente está formado por los vértices que fueron visitados en el ramo de v_i pero no fueron clasificados a ninguna raíz anterior $v_1, \ldots, v_{i-1}$. Esto se puede implementar fácilmente con una pila auxiliar $\mathcal{P}$, empujando los vértices en la pila en el orden de visita en un DFS y al llegar a una raíz, quitándolos del encima de la pila hasta llegar a la raíz misma. Así nada más el componente está eliminado de la pila.

Si el grafo contiene solamente aristas de árbol, cada vértice forma su propio componente.

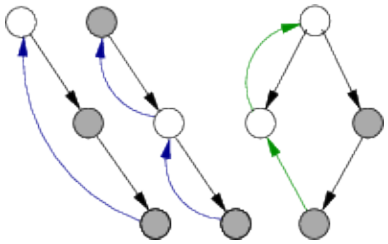


Figura 3.24: Las aristas azules son retrocedentes y las aristas verdes transversas. Las aristas de árbol están dibujados en negro y otros vértices (del mismo componente) en blanco.

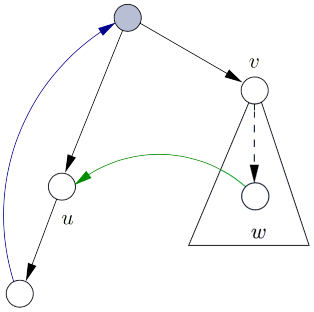


Figura 3.25: La raíz del componente está dibujado en gris y la flecha no continua es un camino, no necesariamente una arista directa.

Nota que las aristas procedentes no tienen ningún efecto en los componentes. Para que una vértice pueda pertenecer en el mismo componente con otro vértice, tienen que ser conectados por un camino que contiene aristas retrocedentes o transversas. La figura 3.24 muestra las situaciones en las cuales dos vértices (en gris) pueden pertenecer en el mismo componente fuertemente conexo.

Utilizamos un arreglo auxiliar $\mathcal{A}(v)$ para guadrar un número para cáda vértice encontrado: $\mathcal{A}(v)$ es el mínimo entre los $I(v)$ los v para los cuales $\exists w$ que es descendiente de v y además $\{w, u\}$ es retrocedente o transversal y la raíz del componente de u es un antepasado de v.

Si $\mathcal{A}(v) = I(v)$, sabemos que v es una raíz de un componente. Entonces, $\forall v \in V$: $\mathcal{A}(v) < I(v)$, porque $\mathcal{A}(v)$ contiene el valor del $I(v)$ de un vértice anteriormente recorrido por una arista retrocedente o transversa o alternativamente un valor de $\mathcal{A}(v)$ está pasado a v por un descendiente. Aplica que $\mathcal{A}(v)$ es el mínimo entre tres conjuntos: los $I(v)$, los $\mathcal{A}(u)$ que contiene aquellos u que son hijos de v, y los $I(u)$ para los cuales $\{v, u\}$ es retrocedente o transversa y además la raíz del componente de u es un antepasado de v.

Durante la ejecución el algoritmo, en la pila auxiliar $\mathcal{P}$ habrá vértices los componentes de los cuales no han sido determinados todavía. Checamos la pila al momento de procesar una arista retrocedente o transversa $\{v, u\}$ si la raíz de u es un antepasado de v. Si lo es, u está en el mismo componente con v y los dos vértices v y la raíz están todavía en la pila.

```
E = {('a', 'b'), ('a', 'c'), ('b', 'd'), \
     ('c', 'a'), ('d', 'e'), ('e', 'b')}
```

```
acc = dict() # adj es un mapeo de conjuntos de adyacencia
for (u, v) in E:
    if u not in acc:
        acc[u] = set()
    acc[u].add(v) # dirigido; vale solamente en un sentido

i = 0
I = dict()
A = dict()

def fuerteconexo(v, comp, P, acc): # algoritmo de Tarjan
    global i, I, A
    A[v] = i
    I[v] = i
    i += 1
    if v not in comp:
        comp[v] = set()
    if v not in acc: # no salen aristas
        return # no hay nada en el componente
    P.insert(0, v) # al inicio de la pila
    for u in acc[v]:
        if not u in comp:
            fuerteconexo(u, comp, P, acc)
            A[v] = min(A[v], A[u])
        elif u in P:
            A[v] = min(A[v], I[u])
    if A[v] == I[v]:
        while True:
            u = P.pop(0) # quitar el primero
            comp[v].add(u)
            if u == v:
               break

def componentes(acc):
    comp = dict() # componentes
    for v in acc: # para cada origen
        if v not in comp: # si no tiene componente asignado
            fuerteconexo(v, comp, list(), acc) # calcularlo
             print(comp[v]) # imprimirlo

componentes(acc)
```

3.9.3. Búsqueda en anchura BFS

1. Crea una cola vacía $\mathcal{L}$.
2. Asigna $u := v$.

3. Marca u visitado.
4. Añade cada vértice **no marcado** en $\Gamma(v)$ al *final* de $\mathcal{L}$.
5. Si $\mathcal{L}$ está vacía, concluye.
6. Asigna $u :=$ el **primer** vértice en $\mathcal{L}$.
7. Continua del tercer paso.

```
def bfs(inicio, adj):
    visitados = []
    cola = [inicio]
    while len(cola) > 0:
        actual = cola.pop(0) # primero
        if actual in visitados:
            continue
        else:
            visitados.append(actual)
            for vecino in adj[actual]:
                cola.append(vecino)
    return visitados

E = {('a', 'b'), ('a', 'c'), ('b', 'd'), ('c', 'e'), ('d', 'f')}
adj = dict() # adj es un mapeo de conjuntos de adyacencia
for arista in E:
    for vertice in arista:
        if vertice not in adj:
            adj[vertice] = set()
    (u, v) = arista
    adj[v].add(u)
    adj[u].add(v) # no dirigido; vale en ambos sentidos

bfs('b', adj)
```

Usos del BFS incluyen obviamente la búsqueda y el cálculo de distancias desde un vértice específico (hay que recordar cuál vértice puso a cuál en la cola, darle distancia cero al inicial y luego a los demás uno más el valor de quién lo agregó) y la detección de ciclos (por ver si se vuelve a un vértice ya visitado). También permite calcular los componentes conexos y hasta la detección de grafos bipartitos (por coloreo con $k = 2$).

3.10. Diseño de algoritmos

La meta al diseñar un algoritmo para un problema es encontrar una manera eficiente a llegar a la solución deseada. El diseño empieza por buscar un punto de vista adecuado al problema. Muchos problemas tienen transformaciones que permiten pensar en el problema en términos de otro problema, mientras en optimización, los problemas tienen *problemas duales* que tienen la misma solución pero pueden resultar más fáciles de resolver.

Algunos problemas se puede dividir en **subproblemas** así que la solución del problema entero estará compuesta por las soluciones de sus partes y las partes pueden ser solucionados

(completamente o relativamente) independientemente. La composición de tal algoritmo puede ser iterativo o recursivo. Hay casos donde los mismos subproblemas ocurren varias veces y es importante evitar tener que resolverlos varias veces.

La formación de una solución óptima por mejoramiento de una solución factible. En algunos casos es mejor hacer cualquier aumento, aunque no sea el mejor ni localmente, en vez de considerar todas las alternativas para poder después elegir vorazmente o con otra heurística una de ellas.

En general, algoritmos heurísticos pueden llegar al óptimo (global) en algunos casos, mientras en otros casos terminan en una solución factible que no es la óptima, pero ninguna de las operaciones de aumento utilizados logra mejorarla. Este tipo de solución se llama un **óptimo local**.

El método **dividir y conquistar** divide un problema grande en varios subproblemas así que cada subproblema tiene la misma pregunta que el problema original, solamente con una instancia de entrada más simple. Después se solucionan todos los subproblemas de una manera recursiva. Las soluciones a los subproblemas están combinadas a formar una solución del problema entero. Dividir-conquistar es el tema de la sección 10.6 de Grimaldi [2017].

Para las instancias del tamaño mínimo, es decir, las que ya no se divide recursivamente, se utiliza algún procedimiento de solución simple. La idea es que el tamaño mínimo sea constante y su solución lineal en su tamaño por lo cual la solución de tal instancia también es posible en tiempo constante desde el punto de vista del método de solución del problema entero.

Para que sea eficiente el método para el problema entero, además de tener un algoritmo de tiempo constante para las instancias pequeñas básicas, es importante que

- el costo computacional de *dividir* un problema a subproblemas sea bajo y
- la computación de *juntar* las soluciones de los subproblemas sea eficiente.

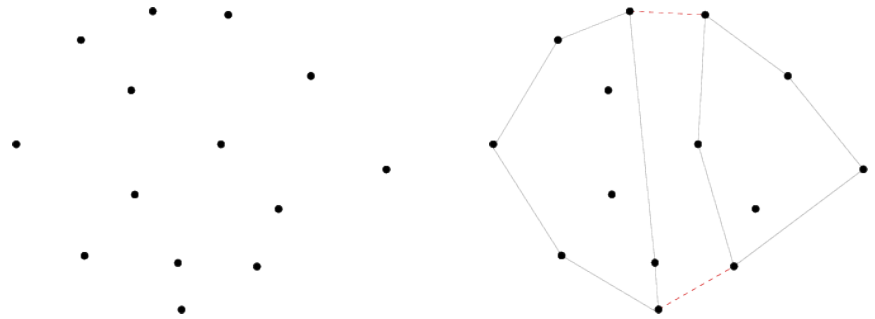
Las divisiones a subproblemas generan un árbol abstracto, la altura de cual determina el número de niveles de división. Para lograr un árbol abstracto balanceado, normalmente es deseable dividir un problema a subproblemas de más o menos el mismo tamaño en vez de dividir a unos muy grandes y otros muy pequeños. Un buen ejemplo del método dividir y conquistar es ordenamiento por fusióque tiene complejidad asintótica $\mathcal{O}(n \log n)$.

Como ejemplo, la **cubierta convexa** es la *región convexa* mínima que contiene un dado conjunto de puntos en $\mathbb{R}^2$. Una región es convexa si todos los puntos de un segmento de línea que conecta dos puntos incluidos en la región, también están incluidos en la misma región.

1. Dividir el conjunto de puntos en dos subconjuntos de aproximadamente el mismo tamaño.
2. Calcular la cubierta de cada parte.
3. Juntar las soluciones de los subproblemas a una cubierta convexa de todo el conjunto.

La división se itera hasta llegar a un sólo punto; la cubierta de un sólo punto es el punto mismo.

Procedemos a dividir el conjunto en dos partes así que uno esté completamente a la izquierda del otro por ordenarlos según su coordenada x (en tiempo $\mathcal{O}(n \log n)$ para n



(a) Puntos. (b) Puentes.

Figura 3.26: Un ejemplo de puntos y puentes para el problema de la cubierta convexa.

puntos). Para juntar dos cubiertas, el caso donde una está completamente a la izquierda de la otra es fácil: basta con buscar dos segmentos de "**puente**" que tocan en cada cubierta pero no corten ninguna. Tales puentes se encuentra por examinar en orden los puntos de las dos cubiertas, asegurando que la línea infinita definida por el segmento entre los dos puntos elegidos no corta ninguna de las dos cubiertas. También hay que asegurar que los dos puentes no corten uno al otro. Hay un ejemplo en la figura 3.26.

Un orden posible para evaluar pares de puntos como candidatos de puentes es empezar del par donde uno de los puntos maximiza la coordinada y en otro maximiza (o minimiza) la coordinada x. Hay que diseñar cómo avanzar al elegir nuevos pares utilizando alguna heurística que observa cuáles de las dos cubiertas están cortadas por el candidato actual. Lo importante es que cada punto está visitado por máximo una vez al buscar uno de los dos puentes.

La complejidad es una ecuación de recursión

$$S(n) = \begin{cases} \mathcal{O}(1), & \text{si } n = 1, \\ 2S\left(\frac{n}{2}\right) + \mathcal{O}(n), & \text{en otro caso,} \end{cases} \tag{3.25}$$

cuya solución es $\mathcal{O}(n \log n)$.

Como ejemplo, utilizamos el algoritmo de Strassen que es un algoritmo para multiplicar dos matrices. Sean $A = (a_{ij})$ y $B = (b_{ij})$ matrices de dimensión $n \times n$.

Sabemos que $AB = C = (c_{ij})$ donde $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. La computación de cada c_{ij} toma tiempo $\Theta(n)$ (por las n multiplicaciones y las $n - 1$ sumaciones) y son exactamente n^2 elementos, por lo cual la complejidad asintótica del algoritmo ingenuo es $\Theta(n^3)$.

Un mejor algoritmo tiene la siguiente idea; sea $n = k^2$ para algún entero positivo k. Si $k = 1$, utilizamos el método ingenuo. En otro caso, divimos las matrices de entrada en cuatro partes

$$\begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} \tag{3.26}$$

y hacemos lo mismo para la matriz B así que cada parte tiene dimensión $\frac{n}{2} \times \frac{n}{2}$. Luego calculamos multiplicaciones con estas partes,

1. $A_{11} \cdot B_{11}$,
2. $A_{12} \cdot B_{21}$,

3. $A_{11} \cdot B_{12}$,
4. $A_{12} \cdot B_{22}$,
5. $A_{21} \cdot B_{11}$,
6. $A_{22} \cdot B_{21}$,
7. $A_{21} \cdot B_{12}$,
8. $A_{22} \cdot B_{22}$,

y luego sumamos algunos de estos para obtener los pedazos de la matriz producto C:

1. $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$,
2. $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$,
3. $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$,
4. $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$.

Ahora bien, en el algoritmo de Strassen se hace lo siguiente con menos operaciones:

1. $S_1 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$,
2. $S_2 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$,
3. $S_3 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$,
4. $S_4 = (A_{11} + A_{12}) \cdot B_{22}$,
5. $S_5 = A_{11} \cdot (B_{12} - B_{22})$,
6. $S_6 = A_{22} \cdot (B_{21} - B_{11})$,
7. $S_7 = (A_{21} + A_{22}) \cdot B_{11}$,
8. $C_{11} = S_1 + S_2 - S_4 + S_6$,
9. $C_{12} = S_4 + S_5$,
10. $C_{21} = S_6 + S_7$,
11. $C_{22} = S_2 - S_3 + S_5 - S_7$.

3.10.1. Podar-buscar

El método **podar-buscar** (inglés: prune and search) es parecido a dividir-conquistar, con la diferencia que después de dividir, el algoritmo ignora la otra mitad por saber que la solución completa se encuentra por solamente procesar en la otra parte. Una consecuencia buena es que tampoco hay que unir soluciones de subproblemas.

Como un ejemplo, analicemos la búsqueda entre n claves de la clave en posición i en orden decreciente de los datos. Para $i = \frac{n}{2}$, lo que se busca es la **mediana** del conjunto. Un algoritmo ingenuo sería buscar el mínimo y eliminarlo i veces, llegando a la complejidad asintótica $\Theta(i \cdot n)$. Aplicando un algoritmo de ordenación de un arreglo, llegamos a la complejidad asíntótica $\Theta(n \log n)$.

Solucionar esto por podar-buscar resulta parecido al ordenación rápida:

1. elegir un elemento pivote p;
2. dividor los elementos en dos conjuntos:
   - A donde las claves son menores a p y

- B donde son mayores o iguales;

3. continuar de una manera recursiva solamente con uno de los dos conjuntos A y B;
4. elegir el conjunto que contiene al elemento $i$ésimo en orden decreciente;
5. para saber dónde continuar, basta con comprar i con $|A|$ y $|B|$.

El truco en este algoritmo es en la elección del elemento pivote así que la división sea buena. Queremos asegurar que exista una constante q tal que $\frac{1}{2} \leq q < 1$ así que el conjunto mayor de A y B contiene nq elementos. Así podríamos llegar a la complejidad

$$T(n) = T(qn) + \Theta(n) \leq cn \sum_{i=0}^{\infty} q^i = \frac{cn}{1-q} = \mathcal{O}(n). \tag{3.27}$$

La elección del pivote es como sigue:

1. Divide los elementos en grupos de cinco (o menos en el último).
2. Denota el número de grupos por $k = \lceil \frac{n}{5} \rceil$.
3. Ordena en tiempo $\mathcal{O}5 \in \mathcal{O}1$ cada grupo.
4. Elige la mediana de cada grupo.
5. Entre las k medianas, elige su mediana p recursivamente.
6. El elemento p será el pivote.

De esta manera podemos asegurar que por lo menos $\lfloor \frac{n}{10} \rfloor$ medianas son mayores a p y para cada mediana mayor a p hay dos elementos mayores más en su grupo, por lo cual el número máximo de elementos menores a p son

$$3\lfloor \frac{n}{10} \rfloor < \frac{3}{4}n \tag{3.28}$$

para $n \geq 20$. También sabemos que el número de elementos mayores a p es por máximo $\frac{3}{4}n$ para $n \geq 20$, por lo cual aplica que $\frac{n}{4} \leq p \leq \frac{3n}{4}$ para $n \geq 20$.

- $|M| = \lceil \frac{n}{5} \rceil$.
- La llamada recursiva $\mathrm{pivote}(\lceil \frac{|M|}{2} \rceil, M)$ toma tiempo $T(\lceil \frac{n}{5} \rceil)$ por máximo.
- También sabemos que $\mathrm{máx}\{|A|, |B|\} \leq \frac{3n}{4}$.
- La Llamada recursiva del último paso toma al máximo tiempo $T(\frac{3n}{4})$.

Entonces, para alguna constante c,

$$T(n) = \begin{cases} c, & \text{si } n < 20 \\ T(\frac{n}{5} + T(\frac{3n}{4}) + cn, & \text{si } n \geq 20. \end{cases} \tag{3.29}$$

Con inducción se llega a $T(n) \leq 20cn \in \mathcal{O}(n)$.

3.10.2. Programación dinámica

En **programación dinámica**, uno empieza a construir la solución desde las soluciones de los subproblemas más pequeños, guardando las soluciones en una forma sistemática para construir soluciones a problemas mayores. Típicamente las soluciones parciales están

Cuadro 3.3: El inicio del triángulo de Pascal.

n \ k								
0	1	2	3	4	5	6	7	8
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1
8	1	8	28	56	70	56	28	8	1

guardadas en un arreglo para evitar a tener que solucionar un subprobelma igual más tarde el la ejecución del algoritmo. El algoritmo pseudo-polinomial que vimos para el problema de la mochila es esencialmente un algoritmo de programación dinámica (PD). En general, la utilidad de PD está en problemas donde la solución del problema completo contiene las soluciones de los subproblemas — una situación que ocurre en algunos problemas de optimización.

Como un ejemplo, calculamos los coeficientes binomiales. Si uno lo aplica de la manera *dividir y conquistar*, es necesario volver a calcular varias veces algunos coeficientes pequeños, llegando a la complejidad asintótica

$$\Omega\left(\binom{n}{k}\right) = \Omega\left(\frac{2^n}{\sqrt{n}}\right). \tag{3.30}$$

Por guardar las soluciones parciales, se llega a $\mathcal{O}(nk)$; sin embargo, por guardarlas, la complejidad de espacio crece.

Las coeficientes binomiales forman el *triángulo de Pascal* (cuadro 3.3).

La **distancia de edición** (inglés: edit distance) es una medida de similitud de sucesiones de símbolos (o sea, palabras formadas por un alfabeto). Se define como el *número mínimo de operaciones de edición* que se necesita aplicar a palabra P para llegar a la palabra Q. Aplican $d(P,Q) = d(Q,P)$ y $d(P,P) = 0$. Las operaciones son las siguientes:

- **Insertar** un símbolo en posición i.
- **Eliminar** un símbolo de posición i.
- **Reemplazar** el símbolo en posición i con otro.

Típicamente todas las operaciones tienen el mismo costo (sea uno), aunque se puede dar diferentes costos a diferentes operaciones. Aplica la *desigualdad de triángulo* $d(P,Q) \leq d(P,R)+d(R,Q)$ en el caso básico, pero existen variaciones de la medida que no lo cumplen. Se puede calcular por programación dinámica (la figura 3.4 muestra un ejemplo):

```
def editdist(p, q):
    d = dict()
    np = len(p) + 1
    nq = len(q) + 1
```

Cuadro 3.4: Un ejemplo del cálculo de una distancia de edición con costos unitarios.

		D	I	F	I	C	I	L
	0	1	2	3	4	5	6	7
F	1	1	2	2	3	4	5	6
A	2	2	2	3	3	4	5	6
C	3	3	3	3	4	3	4	5
I	4	4	3	4	3	4	3	4
L	5	5	4	4	4	4	4	3

```
    ce = 1
    ci = 1
    cr = 1
    for i in range(np):
        d[(i, 0)] = i * ci
    for j in range(nq):
        d[(0, j)] = j * ce
    for i in range(1, np):
        for j in range(1, nq):
            eli = d[(i - 1, j)] + ce
            ins = d[(i, j - 1)] + ci
            incr = cr * (p[i - 1] != q[j - 1])
            ree = d[(i - 1, j - 1)] + incr
            d[(i, j)] = min(eli, ins, ree)
    return d[(np -1, nq - 1)]

print(editdist('perro', 'mero'))
```

3.11. Ejemplos de tipos de algoritmos

Una solución ingenua a un problema de optimización combinatorial es hacer una lista completa de todas las soluciones factibles y evaluar la función objetivo para cada una, eligiendo al final la solución cual dio el mejor valor. La complejidad de ese tipo de solución es *por lo menos* $\Omega(|F|)$ donde F es el conjunto de soluciones factibles. El número de soluciones factibles suele ser algo como $\Omega(2^n)$, por lo cual el algoritmo ingenuo tiene complejidad asintótica **exponencial**. Si uno tiene un método eficiente para generar en una manera ordenada soluciones factibles y rápidamente decidir sí o no procesarlos (en el sentido de podar-buscar), no es imposible utilizar un algoritmo exponencial.

Otra opción es buscar por soluciones *aproximadas*, o sea, soluciones cerca de ser óptima sin necesariamente serlo. Una manera de aproximación es utilizar **métodos heurísticos**, donde uno aplica una regla simple para elegir candidatos. Si uno siempre elije el candidatos que desde el punto de vista de evaluación local se ve el mejor, la heurística es **voraz**.

Muchos problemas de optimización combinatorial consisten de una parte de construcción de cualquier solución factible. Esta construcción tiene la misma complejidad que el proble-

ma de decisión de la *existencia* de una solución factible. No es posible que sea más fácil solucionar el problema de optimización que el problema de decisión a cual está basado.

Como un ejemplo, buscamos encontrar un árbol de expansión mínimo en un grafo ponderado no dirigido. Para construir un árbol de expansión cualquiera — o sea, una solución factible — podemos empezar de cualquier vértice, elegir una arista, y continuar al vecino indicado, asegurando al añadir aristas que nunca regresamos a un vértice ya visitado con anterioridad. Logramos a encontrar la solución óptima con una heurística voraz: siempre elige la arista con menor peso para añadir en el árbol que está bajo construcción.

En lo siguiente se presentan dos algoritmos, de Prim y de Kruskal, que son el tema de la sección 13.2 de Grimaldi [2017].

3.11.1. Algoritmo de Prim

Empezamos por incluir en el árbol **la arista de peso mínimo**. En cada paso, se elige entre los vecinos de los vértices ya incluidos aquel que se pueda añadir con el menor peso entre los candidatos. Se guarda el "costo" de añadir un vértice en un arreglo auxiliar $c[v]$ y asignamos $c[v] = \infty$ para los que no son vecinos de vértices ya marcados. Para saber cuales vértices ya están incluidos, se necesita una estructura de datos; con montículos de Fibonacci complejidad de $\mathcal{O}(m + n \log n)$.

En la implementación ejemplo en Python se usa una librería `copy`[1] para crear una copia auxiliar del grafo que se pueda modificar durante el algoritmo, quitando las aristas que ya han sido consideradas.

```
grafo = {(1, 3): 4, (2, 3): 2, (2, 4): 1, \
         (3, 4): 2, (4, 6): 1, (5, 6): 2}
from copy import deepcopy
pendientes = deepcopy(grafo)
arista = min(pendientes.keys(), key = (lambda k: pendientes[k]))
del pendientes[arista] # se consideran una sola vez
arbol = {arista}
peso = grafo[arista]
incluidos = set()
incluidos.update(arista)

while len(pendientes) > 0:
    candidatos = dict()
    redundantes = set()
    for arista in pendientes:
       comunes = len(incluidos.intersection(arista))
        if comunes == 0:
            continue # no es candidato
        elif comunes == 1:
            candidatos[arista] = grafo[arista] # es candidato
        else: # forma un ciclo
             redundantes.add(arista)
```

[1] https://docs.python.org/dev/library/copy.html

```
    for inutil in redundantes:
        del pendientes[inutil]
    if len(candidatos) == 0:
        break
    sel = min(candidatos.keys(), key = (lambda k: candidatos[k]))
    del pendientes[sel] # se consideran una sola vez
    arbol.add(sel)
    peso += grafo[sel]
    incluidos.update(sel)

print('MST con peso', peso, ':', arbol)
```

3.11.2. Algoritmo de Kruskal

Empezar a añadir aristas, de la menos pesada a la más pesada, cuidando a no formar ciclos por marcar vértices al haberlos tocado con una arista. El algoritmo termina cuando todos los vértices están en el mismo árbol. Su complejidad es $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$.

```
grafo = {(1, 3): 4, (2, 3): 2, (2, 4): 1, \
         (3, 4): 2, (4, 6): 1, (5, 6): 2}

from copy import deepcopy
cand = deepcopy(grafo)
arbol = set()
peso = 0
comp = dict()

while len(cand) > 0:
    arista = min(cand.keys(), key = (lambda k: cand[k]))
    del cand[arista] # se consideran una sola vez
    (u, v) = arista
    c = comp.get(v, {v})
    if u not in c:
        arbol.add(arista)
        peso += grafo[arista]
        nuevo = c.union(comp.get(u, {u}))
        for w in nuevo:
            comp[w] = nuevo
print('MST con peso', peso, ':', arbol)
```

3.11.3. ¿Cómo guiar la búsqueda?

En los algoritmos de optimización combinatorial que evalúan propiedades de varios y posiblemente todos los candidatos de solución, es esencial saber "guiar" la búsqueda de la solución y evitar evaluar "candidatos malos". Un ejemplo de ese tipo de técnica es el método *podar-buscar*. Algoritmos que avancen siempre en el candidato *localmente óptimo* se llaman **voraces**.

En el método de "vuelta atrás" (inglés: backtracking) se aumenta una solución parcial utilizando candidatos de aumento. En cuanto una solución está encontrada, el algoritmo vuelve a examinar un ramo de aumento donde no todos los candidatos han sido examinados todavía.

Cuando uno utiliza **cotas** para decidir cuáles ramos dejar sin explorar, la técnica se llama *ramificar-acotar* (inglés: branch and bound). Es recomendable utilizar métodos tipo ramificar-acotar solamente en casos donde uno no conoce un algoritmo eficiente y no basta con una aproximación. Los ramos de la computación consisten de soluciones factibles distintas y la subrutina para encontrar una cota (superior para maximización y inferior para minimización) debería ser rápida. Normalmente el recorrido del árbol de soluciones factibles se hace **en profundidad**. Cada hoja del árbol corresponde a una *solución factible*, mientras los vértices internos son las operaciones de aumento que construyen las soluciones factibles. El algoritmo *tiene que recordar el mejor resultado visto* para poder eliminar ramos que por el valor de su cota no pueden contener soluciones mejores a la ya conocida.

En la versión de optimización del problema de viajante TSP, se busca por el ciclo de menor costo/peso en un grafo ponderado. En el caso general, podemos pensar que el grafo sea no dirigido y completo. Utilizamos un método tipo ramificar-acotar para buscar la solución óptima. Suponemos que el orden de procesamiento de las aristas es fijo.

El árbol de soluciones consiste en decicir para cada uno de los $\binom{n}{2}$ aristas del grafo sí o no está incluida en el ciclo. Para el largo $\mathcal{L}(R)$ de cualquier ruta R aplica que

$$\mathcal{L}(R) = \frac{1}{2}\sum_{i=1}^{n}\left(\mathcal{L}(v_{i-1}, v_i) + \mathcal{L}(v_i, v_{i+1})\right), \tag{3.31}$$

donde los vértices de la ruta han sido numerados según su orden de visita en la ruta así que el primer vértice tiene dos números v_1 y v_{n+1} y el último se conoce como v_n y v_0 para dar continuidad a la ecuación.

Para cualquier ruta R el costo de la arista incidente a cada vértice es por lo menos el costo de la arista más barata incidente a ese vértice. Para la ruta más corta $R_{\text{mín}}$ aplica que $\mathcal{L}(R_{\text{mín}}) \geq \frac{1}{2}\sum_{v\in V}$ los largos de las dos aristas más baratas incidentes a v. Al procesar la arista $\{v, w\}$, el paso "ramificar" es el siguiente:

1. Si al excluir $\{v, w\}$ resultaría que uno de los vértices v o w tenga menos que dos aristas incidentes para la ruta, ignoramos el ramo de excluirla.
2. Si al incluir $\{v, w\}$ resultaría que uno de los vértices v o w tenga más que dos aristas incidentes para la ruta, ignoramos el ramo de inclusión.
3. Si al incluir $\{v, w\}$ se generaría un ciclo en la ruta actua sin haber incluido todos los vértices todavía, ignoramos el ramo de inclusión.

Después de haber eliminado o incluído aristas así, computamos un nuevo valor de $R_{\text{mín}}$ para las elecciones hechas y lo utilizamos como la cota inferior. Si ya conocemos una solución mejor a la cota asi obtenida, ignoramos el ramo. Al cerrar un ramo, regresamos por el árbol (de la manera DFS) al nivel anterior que todavía tiene ramos sin considerar. Cuando ya no queda ninguno, el algoritmo termina.

3.11.4. Soluciones no-óptimas

En situaciones donde todos los algoritmos conocidos son lentos, vale la pena considerar la posilibidad de usar una solución aproximada, o sea, una solución que tiene un valor de la función objetivo *cerca del valor óptimo*, pero no necesariamente el óptimo mismo. Depende del área de aplicación sí o no se puede hacer esto eficientemente. En muchos casos es posible llegar a una solución aproximada muy rápidamente mientras encontrar la solución óptima puede ser imposiblemente lento.

Un algoritmo de aproximación puede ser determinista o no determinista. Si el algoritmo de aproximación **no es determinista** y ejecuta muy rápidamente, es común *ejecutarlo varias veces* y elegir el mejor de las soluciones aproximadas así producidas. Un algoritmo de aproximación bien diseñado cuenta con un análisis formal que muestra que la diferencia entre su solución y la solución óptima es de un *factor constante*. Este factor se llama el **factor de aproximación**; este factor es < 1 para maximización y > 1 para minimización. Depende de la aplicación qué tan cerca debería ser la solución aproximada a la solución óptima.

El *valor extremo* del factor sobre el conjunto de todas las instancias del problema es la **tasa** o **índice de aproximación** (inglés: approximation ratio). Un algoritmo de aproximación tiene **tasa constante** si el valor de la solución encontrada es por máximo un múltiple constante del valor óptimo. También habrá que mostrar formalmente que el algoritmo de aproximación tiene complejidad polinomial. En el caso de algoritmos de aproximación probabilistas, basta con mostrar que sea polinomial con alta probabilidad.

Definición 16. Bin Packing *(empaque de cajas)*
Entrada*: un conjunto finito de objetos* $\Phi = \{\varphi_1, \varphi_2 \ldots, \varphi_N\}$*, cada uno con un tamaño definido* $t(\varphi_i) \in \mathbb{R}$.
Pregunta*: ¿Cómo empaquetear en cajas de tamaño fijo* T *los objetos así que* $T \geq \max\{t(\varphi_i) \mid \varphi_i \in \Phi\}$ *y que el número de cajas utilizadas sea mínima.*

Bin Packing también es NP-completo. Un *algoritmo de aproximación* para ello es el siguiente:

1. Ordenar las cajas en una fila.
2. Procesamos los objetos en orden.
3. Primero intentamos poner el objecto actualmente procesado en la primera caja de la fila.
4. Si cabe, lo ponemos allí, y si no, intentamos en la siguiente caja.
5. Iterando así obtenemos alguna asignación de objetos a cajas.

Denotamos con $\mathrm{OPT}(\Phi)$ el número de cajas que contienen *por lo menos un objeto* en la asignación óptima. Se puede mostrar que el algoritmo de aproximación simple utiliza al máximo $\frac{17}{10}\mathrm{OPT}(\Phi) + 2$ cajas. Esto significa que nunca alejamos a más de 70 % de la solución óptima.

Podemos mejorar aún por ordenar los objetos así que intentamos primero el más grande y después el segundo más grande. Para este caso se puede mostrar que llegamos a utilizar al máximo $\frac{11}{9}\mathrm{OPT}(\Phi) + 4$ cajas, que nos da una distancia máxima de unos 22 % del óptimo.

3.12. Algoritmos de aproximación

Una versión del TSP con los pesos involucra un grafo completo ponderado con *distancias* entre los vértices $d(v, w)$ que cumplen con la *desigualdad de triángulo* $d(v, u) \leq d(v, w) + d(w, u)$. También es un problema NP-completo. Un *algoritmo de aproximación* es el siguiente:

1. Construye un árbol de expansión mínimo en tiempo $\mathcal{O}(\log n)$.
2. Elige un vértice de inicio cualquiera v.
3. Recorre el árbol con DFS en tiempo $\mathcal{O}(m + n)$ e imprime los vértices en preorden.
4. Imprime v en tiempo $\mathcal{O}(1)$.

El DFS recorre *cada arista del árbol dos veces*; podemos pensar en el recorrido como una ruta larga R' que visita cada vértice por lo menos una vez, pero varias vértices más de una vez. "Cortamos" de la ruta larga R' cualquier visita a un vértice que ya ha sido visitado, así logrando el mismo efecto de imprimir los vértices en preorder. Por la desigualdad de triángulo, sabemos que la ruta cortada R no puede ser más cara que la ruta larga R'. El costo total de R' es dos veces el costo del árbol de expansión mínimo.

Para lograr a comparar el resultado con el óptimo, hay que analizar el óptimo en términos de árboles de expansión. Si eliminamos cualquier arista de la ruta óptima R_{OPT}, obtenemos un árbol de expansión. El peso de este árbol es por lo menos el mismo que el peso de un árbol de expansión mínimo C. Entonces, si marcamos el costo de la ruta R con $c(R)$, hemos mostrado que necesariamente $c(R) \leq c(R') =\leq 2C \leq 2c(R_{\mathsf{OPT}})$.

Como otro ejemplo, regresamos al problema MINCUT. Vamos a considerar multigrafos, o sea, permitimos que entre un par de vértices exista más que una arista en el grafo de entrada $\mathcal{G}$. Considerando que un grafo simple es un caso especial de un multigrafo, el resultado del algoritmo que presentamos aplica igual a grafos simples.

- MINCUT $\in$ P.
- Estamos buscando un corte $C \subseteq V$ de $\mathcal{G}$.
- La capacidad del corte es el número de aristas que lo crucen.
- Suponemos que la entrada $\mathcal{G}$ sea conexo.
- Todo lo que mostramos aplicaría también para grafos (simples o multigrafos) ponderados con pesos no negativos.

Un algoritmo determinista es el siguiente:

- A través de flujo máximo: $\mathcal{O}(nm \log(n^2/m))$.
- Habría que repetirlo para considerar *todos los pares de fuente-sumidero*.
- Se puede demostrar que basta con $(n - 1)$ repeticiones.
- $\Longrightarrow$ MINCUT $\in \Omega(n^2 m)$.
- Con unos trucos: MINCUT $\in \mathcal{O}(nm \log(n^2/m))$.
- Grafos densos: $m \in \mathcal{O}(n^2)$.

Al contraer la arista $\{u, v\}$ reemplazamos los dos vértices u u v por un vértice nuevo w. La arista contraída desaparece, y para toda arista $\{s, u\}$ tal que $s \notin \{u, v\}$, "movemos" la arista a apuntar a w por reemplazarla por $\{s, w\}$. Igualmente reemplazamos aristas $\{s, u\}$

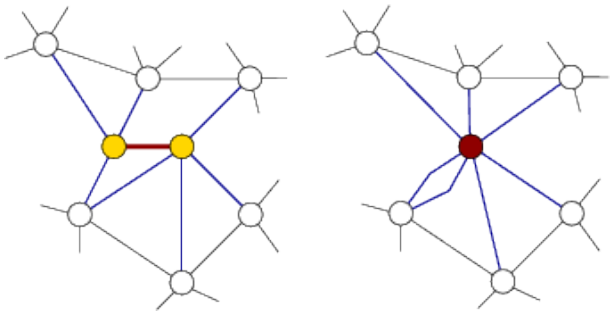


Figura 3.27: Un ejemplo de una contracción de una arista.

labelcon

por aristas $\{s, w\}$ para todo $s \notin \{u, v\}$.

Si contraemos un conjunto $E' \subseteq E$, el resultado no depende del orden de contracción. Después de las contracciones, los vértices que quedan representan subgrafos conexos del grafo original. Empezando con el grafo de entraga $\mathcal{G}$, si elegimos iterativamente al azar entre las aristas presentes una para contracción hasta que quedan sólo dos vértices, el número de aristas en el multigrafo final entre esos dos vértices corresponde a un corte de $\mathcal{G}$.

Al contraer la arista $\{u, v\}$, el nombre del conjunto combinado en la estructura será v y sus miembros son u y v; originalmente cada vértice tiene su propio conjunto. Entonces, podemos imprimir los conjuntos C y $V \setminus C$ que corresponden a los dos vértices que quedan en la última iteración.

- La elección uniforme de una arista para contraer se puede lograr en $\mathcal{O}(n)$.
- En cada iteración eliminamos un vértice, por lo cual el algoritmo de contracción tiene complejidad cuadrática en n.
- Lo que queda mostrar es que el corte así producido sea el mínimo con una probabilidad no cero.
- Así por repetir el algoritmo, podríamos aumentar la probabilidad de haber encontrado el corte mínimo.
- Si la capacidad del corte mínimo es k, ningún vértice puede tener grado menor a k.
- El número de aristas satisface $m \geq \frac{1}{2}nk$ si la capacidad del corte mínimo es k.
- Lax capacidad del corte mínimo en $\mathcal{G}$ después de la contracción de una arista es mayor o igual a la capacidad del corte mínimo en $\mathcal{G}$.

3.13. Tarea 5

Las tareas se responden en línea. Hay que usar el mismo usuario para cada tarea y registrar el usuario con la matrícula correspondiente en la página de resultados del semestre actual para recibir puntos por ellas.

1. ¿Cuántas llamadas a la rutina `bbinaria` se necesita al buscar por el valor especificado en la tarea personalizada en un arreglo cuyo contenido son los enteros ordenados desde un valor especificado en la tarea personalizada hasta el valor límite especificado en pasos del tamaño especificado?
2. ¿Qué es la altura mínima posible de un árbol binario lleno con la cantidad de nodos especificada en la tarea personalizada?
3. Agregando en el orden proporcionado los elementos proporcionados en la tarea personalizada al árbol binario simple del código ejemplo, ¿cuántas llamadas a la rutina `ubicar` se realizan al buscar por el elemento solicitado en la tarea personalizada en el árbol resultante?
4. Si los costos de las operaciones de edición toman los valores especificados en la tarea personalizada, ¿cuánto es la distancia de edición entre las palabras "sábado" y "domingo"?
5. Asignando el peso especificado a la arista especificada en la tarea personalizada (sustituyendo el valor existente en el caso que haya) a la instancia del problema de árbol de expansión mínimo, ¿cuánto vale el óptimo?

3.13.1. Preguntas de verificación

Discute lo siguiente con los compañeros y con la profesora hasta que esté todo claro. Conviene consultar el libro de texto (capítulos y secciones indicados en el material de la unidad en esta página) y hasta buscar por videos en la web. Cuando ya no cabe duda, procede al proyecto individual.

Describe en palabras propias y dibujos de apoyo la forma de operación de las siguientes estructuras de datos: arreglo, lista, pila, cola, árbol binario.

¿Qué es el propósito de algoritmos de búsqueda, algoritmos de recorrido y algoritmos de ordenamiento? Menciona ejemplos de los tres tipos de algoritmos.

¿Por qué existen mecanismos para balancear árboles binarios? ¿Por qué no conviene llenarlas sin preocuparse del balance?

¿Qué técnicas existen para el diseño de algoritmos? ¿Qué aplicaciones prácticas puedes pensar para el cálculo de distancias de edición y para la construcción de árboles de expansión? ¿Por qué a veces se usan algoritmos aproximados en luhgar de algoritmos exactos?

Proyecto individual

El alumno puede iniciar el proyecto una vez que haya terminado las cinco tareas en línea (con puntaje parcial o completa). Se recomienda consultar con la profesora en persona en clase antes de iniciar cada fase para recomendaciones, consejos y una revisión de las fases anteriores. Las fases a seguir son:

Propuesta de tema Identifica un fenómeno/situación que presenta un área de oportunidad de mejora en algúna forma de toma de decisiones (por lo general un problema de optimización, aunque problemas de decisión también son posibles) que se pueda atender con los conceptos y técnicas que se discuten en la unidad de aprendizaje.

Revisión de literatura Busca libros, artículos, software, sitios web, etc. relacionados con el tema y se resumen los hallazgos en forma organizada con palabras propias; se incluye una bibliografía completa y en formato consistente al final del reporte, detallando los trabajos consultados. Modelado del problema: se expresa el problema seleccionado en términos de conceptos de las matemáticas discretas (por ejemplo lógica, conjuntos, relaciones, árboles, grafos o una combinación de estos) con una notación matemática clara y consistente, definiendo los datos de entrada, las restricciones, las variables, los objetivos y las salidas deseadas.

Instancias con datos verdaderos Obten información *no-confidencial* de uno o más casos prácticos donde se presenta el problema y se representan esos casos en términos del modelo planteado, citando en la bibliografía las fuentes de información utilizadas.

Diseño de la solución propuesta Identifican y/o construye una o más técnicas (es decir, algoritmos) que permiten resolver el problema planteado, explicando en qué consisten, a qué se basan y cómo funcionan (citando referencias bibliográficas donde necesario). Implementación: se detalla la forma de ejecutar (por lo menos un) método de solución, sea a mano o automatizado a través de un código, para por lo menos una de las instancias identificadas, paso por paso, de forma clara y consistente.

Evaluación Analiza y discute la calidad de la solución propuesta en términos tales como precisión y eficiencia, terminando con conclusiones sobre el trabajo realizado y posibles mejoras que se le pudieran realizar en un futuro.

En cada fase conviene consultar con la profesora en clase por retroalimentación; se pueden consultar múltiples fases en una sola consulta, pero no se permite llevar acabo proyectos sin haber consultado la profesora (es decir, no se reciben reportes de proyectos cuyo desarrollo no ha sido discutido en persona en clase). Al mostrar avances a la profesora, ella podrá indicar cosas que faltan o que no estén aún satisfactoriamente atendidos en el reporte.

Al concluir las fases el alumno entrega su reporte escrito (ojo, no se aceptan diapositivas tipo PowerPoint), sin efectos decorativos, sin hoja portada, sin color de fondo, un solo archivo en formato PDF (no se aceptan otros formatos y no se aceptan entregas en impreso) por correo electrónico a la profesora, indicando en el tema del correo que es un proyecto de clase, nombrando el archivo .pdf con su número de matrícula solamente. No incluir el nombre del alumno dentro del documento. No incluir la matrícula en el contenido del documento, solamente en el nombre del archivo.

Es importante comenzar el reporte con un título que describe su tema (ponerle "proyecto integrador de matemáticas discretas" no sustituye el título). A cada fase se atiende en el reporte con una sección que consiste en texto original bien redactado, sin errores de

gramática, ortografía o puntuación, acompañado con ilustraciones originales y cuadros que cuentan con su texto descriptivo correpsondiente (en el caso de que sean con licencia de reuso libre no comercial, indicando fuente y tipo de licencia en el texto descriptivo). Los textos descriptivos de los cuadros se posicionan encima, mientras en figuras van por debajo.

Todas las fuentes de consulta se indican claramente dentro del texto con un estilo de ingeniería y se detallan en una sección de referencias al final del reporte. Es importante entender que un listado de URLs no cuenta como una bibliografía adecuada sino hay que darle formato. La mejor forma de aclarar dudas es mostrando un borrador a la profesora en clase para que ella pueda orientar sobre el proceso correcto.

En el caso de contar con código, se anexa al final del reporte en un formato adecuado con colores para sintáxis. Jamás se usa capturas de pantalla para incluir código. Capturas de pantalla sirven para mostrar el funcionamiento de un software, aunque en el caso de demostraciones es más recomendable grabar un video del escritorio con comentario en voz y subirlo a un repositorio en línea como por ejemplo YouTube y luego mencionar la URL de video en el reporte. Lo mismo sirve para prototipos físicos: primero incluir fotografías originales en la sección de implementación y concluir con un video.

La fecha límite para entrega de proyecto es el último día hábil anterior al examen ordinario a mediodía (ojo, no medianoche sino mediodía); para segunda oportunidad, es lo mismo pero antes del examen extraordinario. La entrega no vale hasta que la profesora haya contestado de recibido al correo, ya que es común que alumnos anexan archivos rotos (a veces a propósito) o escriben mal el correo. Todo lo que llega después de la hora indicada ya no cuenta para la primera oportunidad, sino solamente se toma en cuenta en el caso que el alumno ocupa presentar extraordinario.

La calificación es por máximo 10 puntos, considerando la calidad, claridad y profundidad en la cual se ha documentado cada fase del proyecto. Se reducen puntos por errores de ortografía, mala estructuración, mala calidad de las ilustraciones, errores técnicos e inconsistencias del reporte. No se podrá mejorar un reporte ya entregado.

Reportes que contienen elementos no originales (es decir, cualquier cosa — figura, código, párrafo de texto — que no estén hechos por el alumno mismo) resultará en una calificación de cero puntos sin posibilidad de mejorar el reporte.

Bibliografía

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, July 2009. ISBN 978-0262033848.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Professional, 2 edition, March 1994. ISBN 978-0201558029.

Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics*. Pearson Modern Classics for Advanced Mathematics Series. Pearson, 5 edition, April 2017. ISBN 978-0321385024.