**6.1 Repaso de temas pasados**

A continuación dejamos unos links a unos videos sobre un par de ejercicios de la clase 5:

1. Ejercicio [Ejercicio 5.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/01_Random.md" \l "ejercicio-52-generala-no-necesariamente-servida) sobre la probabilidad de obtener una generala no servida resuelto por [Matias](https://youtu.be/D_mipwwZjhM) y por [Rafael](https://youtu.be/c2SO3-iSd04).
2. Ejercicio [Ejercicio 5.14](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/03_Figuritas.md" \l "ejercicio-514) del [álbum de Figuritas](https://youtu.be/lSVNxPoRLJA)

**Análisis de alternativas para *propagar***

Los siguientes tres ejercicios proponen diferentes soluciones al [Ejercicio 4.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md#ejercicio-46-propagaci%C3%B3n) de propagación del fuego. Vamos a analizar sus diferencias y comenzar a pensar en su eficiencia. Algunas soluciones tienen errores que deberás corregir oportunamente. ¡Usá el debugger de Python!

*Observación: Cuando te pidamos que cuentes cuántas operaciones hace una función, no nos va a importar el detalle de las constantes. Por ejemplo: si una función para una entrada de largo n hace n+2 operaciones y otra hace 3n+5 nos va a importar que ambas hacen una cantidad****lineal****de operaciones en el tamaño de la entrada, pero no las constantes 2, 3 y 5 que figuran en cada caso. Diremos que la cantidad de operaciones es O(n) (se lee 'o de n'). En cambio, sí vamos a hacer una diferencia si una función hace n y otra hace n^2 operaciones (una va a tener complejidad O(n) y la otra O(n^2)). Volveremos sobre estos temas más adelante.*

**Ejercicio 6.1: Propagar por vecinos**

El siguiente código propaga el fuego de cáda fósforo encendido a sus vecinos inmediatos (si son fósforos nuevos) a lo largo de toda la lista. Y repite esta operación mientras sea necesario. ¿Te animás a estimar cuántas operaciones puede tener que hacer, en el peor caso?

def propagar\_al\_vecino(l):

modif = False

n = len(l)

for i,e in enumerate(l):

if e==1 and i<n-1 and l[i+1]==0:

l[i+1] = 1

modif = True

if e==1 and i>0 and l[i-1]==0:

l[i-1] = 1

modif = True

return modif

def propagar(l):

m = l.copy()

veces=0

while propagar\_al\_vecino(l):

veces += 1

print(f"Repetí {veces} veces la función propagar\_al\_vecino.")

print(f"Con input {m}")

print(f"Y obtuve {l}")

return m

#%%

propagar([0,0,0,0,1])

propagar([0,0,1,0,0])

propagar([1,0,0,0,0])

**Preguntas:**

1. ¿Por qué los tests l[i+1]==0 y l[i-1]==0 de la función propagar\_al\_vecino no causan un IndexError en los bordes de la lista?
2. ¿Por qué propagar([0,0,0,0,1]) y propagar([1,0,0,0,0]), siendo entradas perfectamente simétricas, no generan la misma cantidad de repeticiones de llamadas a la función propagar\_al\_vecino?
3. Sobre la complejidad. Si te sale, calculá:
   * ¿Cuántas veces como máximo se puede repetir el ciclo while en una lista de largo n?
   * ¿Cuántas operaciones hace "propagar\_al\_vecino" en una lista de largo n?
   * Entonces, ¿cuántas operaciones hace como máximo esta versión de propagar en una lista de largo n? ¿Es un algoritmo de complejidad lineal o cuadrática?

**Ejercicio 6.2: Propagar por como el auto fantástico**

El siguiente código propaga el fuego inspirado en las luces del [auto fantástico](https://youtu.be/oNeQi8-PXAU?t=11).

def propagar\_a\_derecha(l):

n = len(l)

for i,e in enumerate(l):

if e==1 and i<n-1:

if l[i+1]==0:

l[i+1] = 1

return l

#%

def propagar\_a\_izquierda(l):

return propagar\_a\_derecha(l[::-1])[::-1]

#%

def propagar(l):

ld=propagar\_a\_derecha(l)

lp=propagar\_a\_izquierda(ld)

return lp

#%%

l = [0,0,0,-1,1,0,0,0,-1,0,1,0,0]

print("Estado original: ",l)

print("Porpagando...")

lp=propagar(l)

print("Estado original: ",l)

print("Estado propagado: ",lp)

**Preguntas:**

1. ¿Por qué se modificó la lista original?
2. ¿Por qué no quedó igual al estado propagado?
3. Corregí el código para que no cambie la lista de entrada.
4. ¿Cuántas operaciones hace como máximo propagar\_a\_derecha en una lista de largo n?
5. Sabiendo que invertir una lista ([::-1]) requiere una cantidad lineal de operaciones en la longitud de la lista ¿Cuántas operaciones hace como máximo propagar en una lista de largo n?

**Ejercicio 6.3: Propagar con cadenas**

Esta versión usa métodos de *cadenas* para resolver el problema separando los fósforos en *grupos sin fósforos quemados* y analizando cada grupo. Sin embargo algo falla...

def trad2s(l):

'''traduce una lista con 1,0 y -1

a una cadena con 'f', 'o' y 'x' '''

d={1:'f', 0 :'o', -1:'x'}

s=''.join([d[c] for c in l])

return s

def trad2l(ps):

'''traduce cadena con 'f', 'o' y 'x'

a una lista con 1,0 y -1'''

inv\_d={'f':1, 'o':0, 'x':-1}

l = [inv\_d[c] for c in ps]

return l

def propagar(l, debug = True):

s = trad2s(l)

if debug:

print(s)#, end = ' -> ')

W=s.split('x')

PW=[w if ('f' not in w) else 'f'\*len(w) for w in W]

ps=''.join(PW)

if debug:

print(ps)

return trad2l(ps)

#%%

l = [0,0,0,-1,1,0,0,0,-1,0,1,0,0]

lp = propagar(l)

print("Estado original: ",l)

print("Estado propagado: ",lp)

**Preguntas:**

1. ¿Porqué se acorta la lista?
2. ¿Podés corregir el error agregando un solo caracter al código?
3. ¿Te parece que este algoritmo es cuadrático como el [Ejercicio 6.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/01_Repaso.md#ejercicio-61-propagar-por-vecinos) o lineal como el [Ejercicio 6.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/01_Repaso.md#ejercicio-62-propagar-por-como-el-auto-fant%C3%A1stico)?

**6.2 Scripting**

En esta sección profundizaremos en el proceso de crear scripts en Python. Esta y las próximas secciones tienen un [video](https://youtu.be/spjiuud3dQ4) asociado que podés ver antes de leer este texto.

**¿Qué es un script?**

Un *script* es un programa que ejecuta una serie de comandos y termina. *Programa* en el sentido clásico de la palabra: una secuencia de eventos. Su traducción literal es **guión**, como el guión de una película, con introducción, nudo y desenlace.

# programa.py

comando1

comando2

comando3

...

Hasta aquí mayormente hemos escrito scripts.

**Un problema**

Cuando hayas escrito un script útil, éste va a comenzar a crecer en funciones y opciones. Vas a querer aplicarlo a otros problemas. Con el tiempo puede convertirse en un programa esencial, pero si no lo cuidás puede convertirse en un lío enorme, en un gran embrollo. Veamos como lo organizamos.

**Definir nombres**

Los nombres deben estar definidos antes de usarse.

def cuadrado (x):

return x\*x

a = 42

b = a + 2 # Requiere que 'a' haya sido definido antes.

z = cuadrado (b) # Requiere que 'cuadrado' y 'b' estén definidos.

**El orden importa.** Casi siempre definimos las variables y las funciones al comienzo.

**Definir funciones**

Es muy útil agrupar todo el código relevante a una misma *tarea* en el mismo lugar. Para eso sirven las funciones.

def leer\_precios(nombre\_archivo):

precios = {}

with open(nombre\_archivo) as f:

f\_csv = csv.reader(f)

for linea in f\_csv:

precios[linea[0]] = float(linea[1])

return precios

Una función simplifica las operaciones repetitivas.

preciosviejos = leer\_precios('preciosviejos.csv')

preciosnuevos = leer\_precios('preciosnuevos.csv')

**¿Qué es una función?**

Una función es una secuencia de comandos, con un nombre.

def nombrefunc(args):

comando

comando

...

return resultado

*Cualquier* comando de Python puede usarse dentro de una función.

def foo():

import math

print(math.sqrt(2))

help(math)

No existen comandos *especiales* en Python (lo cual es muy fácil de recordar).

**Dónde definir funciones**

En Python podemos *definir* funciones en cualquier orden.

def foo(x):

bar(x)

def bar(x):

comandos

# OR

def bar(x):

comandos

def foo(x):

bar(x)

El único requisito es que la función esté definida al momento de ser *usada* (o llamada) durante la ejecución de un programa.

foo(3) # foo tiene que haber sido definida antes

El estilo que preferimos es definir funciones desde abajo hacia arriba ("*bottom-up*")

**El estilo *Bottom-Up***

Este estilo trata a las funciones como ladrillos. Los ladrillos simples ó más pequeños se definen primero, y luego se usan para armar funciones más complejas.

# miprograma.py

def foo(x):

...

def bar(x):

...

foo(x) # Definida antes

...

def spam(x):

...

bar(x) # Definida antes

...

spam(42) # El código que \*usa\* la función está al final

Las funciones complejas se basan en funciones más simples, definidas antes; aunque esto es sólo una cuestión de estilo. Lo único que realmente importa en ése programa es que la llamada a spam(42) esté después que la declaración de las funciones que éste invoca. El orden de las definiciones puede variar, siempre que sea anterior a su uso real.

**Diseño de funciones**

Lo ideal es que una función sea una *caja negra*. Una función debería operar únicamente sobre los parámetros provistos, evitar variables globales y efectos secundarios no esperados. Hay dos conceptos clave: **Diseño Modular** y **Predecibilidad**.

**Doc-strings**

Es buena costumbre incluir documentación en forma de doc-strings. Un doc-string ó "texto de documentación" es texto ubicado en la línea inmediata después del nombre de la función. El doc-string provee información a quien lee la función, pero también se integra con la función help(), IDEs y otras herramientas de programación y documentación.

def leer\_precios(nombre\_archivo):

'''

Lee precios de un archivo de datos CSV con dos columnas.

La primera columna debe contener un nombre y la segunda un precio.

'''

precios = {}

with open(nombre\_archivo) as f:

f\_csv = csv.reader(f)

for row in f\_csv:

precios[linea[0]] = float(linea[1])

return precios

Un doc-string debe ser conciso e indicar qué hace la función. Si es necesario, podés incluir un ejemplo corto de uso y una descripción de los argumentos.

Veremos también la clase que viene que es posible incluir en el doc-string una descripción de lo que se espera que cumplan los parámetros y lo que garantizamos que cumpla la salida (como un contrato).

**Notas sobre el tipo de datos**

También podés agregar, en la definición de funciones, notas sobre el tipo de datos de los parámetros y de la función.

def leer\_precios(nombre\_archivo: str) -> dict:

'''

Lee precios de un archivo de datos CSV con dos columnas.

La primera columna debe contener un nombre y la segunda un precio.

Devuelve un diccionario {nombre:precio, ...}

'''

precios = {}

with open(nombre\_archivo) as f:

f\_csv = csv.reader(f)

for linea in f\_csv:

precios[linea[0]] = float(linea[1])

return precios

Estas notas no modifican al programa y son puramente informativas. Aún así pueden ser usadas por IDEs, comprobadores de código, y otras herramientas.

Aunque -> dict indica al programador que la función devuelve un diccionario, es útil anotar en el doc-string la estructura del diccionario devuelto.

**Ejercicios**

En el [Ejercicio 3.16](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/05_Formato.md#ejercicio-316-un-desaf%C3%ADo-de-formato) (o el [Ejercicio 3.15](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/05_Formato.md#ejercicio-315-agregar-encabezados)) escribiste un programa llamado tabla\_informe.py que imprime un informe con el balance de compra y venta de frutas en un camión.

El programa tiene algunas funciones, como:

# tabla\_informe.py

import csv

def leer\_camion(nombre\_archivo):

'''

Lee un archivo de lotes en un camión

y lo devuelve como lista de diccionarios con claves

nombre, cajones, precio.

'''

camion = []

with open(nombre\_archivo) as f:

rows = csv.reader(f)

headers = next(rows)

for row in rows:

record = dict(zip(headers, row))

cajon = {

'nombre' : record['nombre'],

'cajones' : int(record['cajones']),

'precio' : float(record['precio'])

}

camion.append(cajon)

return camion

...

Sin embargo había también partes del programa que ejecutaban una serie de cálculos en forma de script. Este código estaba casi al final del programa. Por ejemplo:

...

# Output the informe

headers = ('Nombre', 'Cajones', 'Precio', 'Cambio')

print('%10s %10s %10s %10s' % headers)

print(('-' \* 10 + ' ') \* len(headers))

for row in informe:

print('%10s %10d %10.2f %10.2f' % row)

...

En el siguiente ejercicio vamos a volver a ese programa y organizarlo mejor usando funciones.

**Ejercicio 6.4: Estructurar un programa como una colección de funciones**

Volvé a tu programa tabla\_informe.py y modificalo de modo que todas las operaciones principales, incluyendo cálculos e impresión, sean llevados a cabo por una colección de funciones. Guarda la nueva versión en un archivo informe\_funciones.py. Más específicamente:

* Creá una función imprimir\_informe(informe) que imprima el informe.
* Cambiá la última parte del programa de modo que consista sólo en una serie de llamados a funciones, sin ningún cómputo.

**Ejercicio 6.5: Crear una función de alto nivel para la ejecución del programa.**

Juntá la última parte de tu programa en una única función informe\_camion(nombre\_archivo\_camion, nombre\_archivo\_precios). Deberías obtener una función que al llamarla como sigue, imprima el informe:

informe\_camion('../Data/camion.csv', '../Data/precios.csv')

En su versión final tu programa será una serie de definiciones de funciones seguidos por un único llamado a la funcion informe\_camion() (la cual ejecuta todos los pasos que constituyen tu programa).

Cuando tu programa es una única función, es muy simple ejecutarlo con diferentes entradas. Por ejemplo, después de ejecutar tu programa probá estos comandos en modo interactivo:

>>> informe\_camion('../Data/camion2.csv', '../Data/precios.csv')

... mirá el resultado ...

>>> files = ['../Data/camion.csv', '../Data/camion2.csv']

>>> for name in files:

print(f'{name:-^43s}')

informe\_camion(name, '../Data/precios.csv')

print()

... mirá el resultado ...

>>>

**Comentario**

En Python es muy fácil escribir código en forma de un script relativamente poco estructurado, en el que tenés un archivo que contiene una secuencia de comandos. A la larga, casi siempre es mejor convertir estos scripts en funciones para organizar el código.

En algún momento, si ese script crece, vas a desear haber sido un poco más organizado desde el comienzo. Tratá de organizar tu código en funciones simples. Es un buen principio es que cada función haga una sola cosa sencilla y concreta, que tenga una sola responsabilidad.

**6.3 Funciones**

Aunque ya hablamos sobre funciones, dimos pocos detalles sobre su funcionamiento a un nivel algo más profundo. En esta sección esperamos completar algunos conceptos como convenciones de uso, alcance (*scope*) y otros temas.

**Llamando a una función**

Imaginá la siguiente función:

def leer\_precios(nombre\_archivo, debug):

...

Podés llamar a la función pasando los argumentos por orden:

precios = leer\_precios('precios.csv', True)

O podés llamarla usando palabras clave (*keywords*):

precios = leer\_precios(nombre\_archivo = 'precios.csv', debug = True)

**Argumentos por omisión**

Si preferís que un argumento sea opcional (que tenga un valor *por omisión* o *by default*), en ese caso asignale un valor en la definición de la función. Ése será el valor del argumento si llamás a la función sin especificar un valor para ese argumento.

def leer\_precios(nombre\_archivo, debug = False):

...

En la declaración de la función podés asignar un valor a un argumento. Entonces, ese argumento será opcional al invocar a esa funcion y si lo omitís al invocar a la función va a tomar su valor asignado. A ese valor lo llamamos valor por omisión.

d = leer\_precios('precios.csv')

e = leer\_precios('precios.dat', True)

*Nota: Todos los argumentos con valores por omisión deben aparecer al final de la lista de argumentos (primero se declaran todos los argumentos no-opcionales)*

**Si un argumento es opcional, dale un nombre.**

Comparemos estos dos estilos de invocar funciones:

cortar\_datos(data, False, True) # ?????

cortar\_datos(data, ignore\_errores = True)

cortar\_datos(data, debug = True)

cortar\_datos(data, debug = True, ignore\_errores = True)

En la mayoría de los casos los argumentos con nombre hacen al código más claro, más fácil de entender, especialmente si estos argumentos son booleanos, que determinan opciones si-no.

**Buenas prácticas de diseño**

Compará estas dos formas de declarar una misma función. Para comprender cómo usar la primera, tendríamos que explorar dentro de la función y saber que significan sus parámetros. Usá siempre nombres cortos para los argumentos, pero con significado.

def leer\_precios(f, d = False):

...

def leer\_precios(nombre\_archivo, debug = False):

...

Quien use la función podría elegir llamarla con argumentos nombrados.

d = leer\_precios('precios.csv', debug = True)

Hay herramientas que crean automáticamente documentación sobre el uso de las funciones y sus argumentos. Si los nombres tienen significado, la documentación resulta más clara.

**Devolver un resultado**

El comando return termina la función y devuelve un valor.

def cuadrado(x):

return x \* x

Si no se define un resultado, o si falta el comando return, la función devuelve la constante None.

def bar(x):

instrucciones

return

a = bar(4) # a = None

# O TAMBIEN...

def foo(x):

instrucciones # No hay `return`

b = foo(4) # b = None

**Devolver múltiples resultados**

Las funciones sólo pueden devolver una cosa. Si necesitás devolver más de un valor, podés armar una tupla con ellos y devolver la tupla.

def dividir(a,b):

c = a // b # Cociente

r = a % b # Resto

return c, r # Devolver una tupla con c y r

Ejemplo:

x, y = dividir(37,5) # x = 7, y = 2

x = dividir(37, 5) # x = (7, 2)

**Alcance de variables**

En un programa se declaran variables y se les asignan valores. Esto ocurre dentro y fuera de funciones.

x = valor # Variable Global

def foo():

y = valor # Variable Local

Las variables declaradas fuera de funciones son "globales". Las variables declaradas dentro de funciones son "locales". A esto se llama el alcance (*scope*) de una variable.

**Variables locales**

Las variables locales, declaradas dentro de funciones, son privadas.

def leer\_camion(nombre\_archivo):

camion = []

for linea in open(nombre\_archivo):

campos = line.split(',')

s = (campos[0], int(campos[1]), float(campos[2]))

camion.append(s)

return camion

En este ejemplo, nombre\_archivo, camion, linea, campos y s son variables locales.

>>> cajones = leer\_camion('camion.csv')

>>> campos

Traceback (most recent call last):

File "<stdin>", line 1, in ?

NameError: name 'campos' is not defined

>>>

El error significa: *Error de Nombre: el nombre 'campos' no está definido.*

No hay conflicto entre variables locales y variables declaradas en otras partes (funciones o globales).

**Variables globales**

Desde cualquier función se puede acceder a las variables globales declaradas en ese mismo archivo.

nombre = 'Dave'

def saludo():

print('Hola', nombre) # Usa la variable global `nombre`

Las funciones, sin embargo, no alteran normalmente el valor de una variable global.

nombre = 'Dave'

def spam():

nombre = 'Guido'

spam()

print(nombre) # imprime 'Dave'

Aquí hay dos variables diferentes: nombre global, que vale 'Dave', y nombre local, declarada dentro de la función spam() que vale 'Guido'. Cambiar una no cambia la otra: al cambiar el valor de nombre local, nombre global no cambia.

**Acordate: Las asignaciones de valores a variables y las declaraciones de variables dentro de funciones son locales.**

**Modificar el valor de una variable global**

Si necesitás modificar el valor de una variable global desde dentro de una función, la variable tiene que estar declarada como global dentro de la misma función.

nombre = 'Dave'

def spam():

global nombre

nombre = 'Guido' # Cambia el valor de la variable global

Si declaramos global nombre dentro de la función, entonces nombre fuera de la función spam() y dentro de la función spam() refieren a la misma variable, y al modificar una de ellas modificás la otra.

La declaración de globalidad de la variable (con la palabra reservada global) tiene que aparecer antes del uso de la variable dentro de una función, y la declaración de la variable global fuera de la función debe ocurrir en el mismo archivo que ésta.

Dicho esto, hay que decir también que usar variables globales se considera una mala práctica. Tratá de evitar completamente el uso de global. Si tenés una función que depende del estado de una variable global, tu programa es menos modular: no podés reutilizar la función en otro contexto sin agregar una variable global. Si necesitás que una función modifique el estado de algo fuera de esa función, es mejor entonces usar una clase en lugar de una función. Hablaremos de ésto más adelante, en la segunda mitad de la materia.

**Pasaje de argumentos**

Cuando llamás a una función, los argumentos son los nombres que refieren a los valores que le pasás. Estos valores no son copias de los originales (ver [Sección 4.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/04_Objetos.md#44-objetos)). Si le pasás tipos mutables, como listas o diccionarios, la función *sí* los puede modificar.

def foo(items):

items.append(42) # Cambia el valor de items

a = [1, 2, 3]

foo(a)

print(a) # [1, 2, 3, 42]

**Fundamental: Las funciones no reciben una *copia* de los argumentos, sino los argumentos mismos.**

**Reasignar versus modificar**

Existe una sutil pero importante diferencia entre *modificar* el valor de una variable y *reasignar* una variable.

Es importante que entiendas esta diferencia.

def foo(items):

items.append(42) # Modifica el valor de 'items'

a = [1, 2, 3]

foo(a)

print(a) # [1, 2, 3, 42]

# Versus

def bar(items):

items = [4,5,6] # Cambia la variable local 'items' y

# hace que apunte a otro objeto completamente diferente.

b = [1, 2, 3]

bar(b)

print(b) # imprime [1, 2, 3]

*Recordá: reasignar una variable nunca sobreescribe la memoria que ocupaba. Sólo se asocia el nombre de la variable a un nuevo valor.*

**Ejercicios**

Este conjunto de ejercicios te llevan a implementar un programa medianamente complejo. Es no trivial. Hay varios pasos involucrados e implica articular muchos conceptos al mismo tiempo.

La solución que vas a desarrollar involucra sólo unas 25 líneas de código, pero tomate tu tiempo para asegurarte de que entendés cada concepto y cada parte del código por separado.

La parte central del programa informe\_funciones.py resuelve la lectura de archivos de tipo CSV. Por ejemplo, la función leer\_camion() lee un archivo que contiene los datos de un camión organizados como filas, y la función leer\_precios() lee un archivo que contiene precios. En ambas funciones hay una variedad de acciones detallistas y minuciosas, por ejemplo, ambos abren un archivo y lo envuelven con el módulo csv y ambos convierten cada uno de los campos a un tipo de datos diferente.

Si tu tarea fuera de verdad leer datos de archivos, entonces querrías limpiar este código un poco, hacerlo más prolijo, y aplicable a un uso más general. Ésa es nuestra intención ahora:

Comenzá este ejercicio creando un nuevo archivo fileparse.py en la carpeta ejercicios\_python/Clase06. Ahí vamos a trabajar.

*Nota:* En inglés *to parse* significa analizar gramaticalmente (por ejemplo una frase), separándola en sus partes constitutivas. Es un término muy usado en ciencias de la computación que no tiene una traducción compacta al castellano. Mucha gente usa el anglicismo *parsear* para referirse a esta actividad.

**Ejercicio 6.6: Parsear un archivo CSV**

Vamos a empezar por el problema simple de leer un archivo CSV para guardar los datos que contiene en una lista de diccionarios. En el archivo fileparse.py definí la siguiente función:

# fileparse.py

import csv

def parse\_csv(nombre\_archivo):

'''

Parsea un archivo CSV en una lista de registros

'''

with open(nombre\_archivo) as f:

rows = csv.reader(f)

# Lee los encabezados

headers = next(rows)

registros = []

for row in rows:

if not row: # Saltea filas sin datos

continue

registro = dict(zip(headers, row))

registros.append(registro)

return registros

Esta función lee un archivo CSV y arma una lista de diccionarios a partir del contenido del archivo CSV. La función aísla al programador de los múltiples pequeños pasos necesarios para abrir un archivo, "envolverlo" con el módulo csv, ignorar líneas vacías, y demás minucias.

(*un "wrapper" (envoltorio) en programación es una estructura que expone la interfase de un objeto, pero aísla al usuario de los detalles de funcionamiento de ese objeto.*)

Probémoslo en tu IDE o con python3 -i fileparse.py.

>>> camion = parse\_csv('../Data/camion.csv')

>>> camion

[{'nombre': 'Lima', 'cajones': '100', 'precio': '32.2'}, {'nombre': 'Naranja', 'cajones': '50', 'precio': '91.1'}, {'nombre': 'Caqui', 'cajones': '150', 'precio': '103.44'}, {'nombre': 'Mandarina', 'cajones': '200', 'precio': '51.23'}, {'nombre': 'Durazno', 'cajones': '95', 'precio': '40.37'}, {'nombre': 'Mandarina', 'cajones': '50', 'precio': '65.1'}, {'nombre': 'Naranja', 'cajones': '100', 'precio': '70.44'}]

>>>

La función hace lo que queríamos, pero no podemos usar los resultados para hacer cálculos porque todos los datos recuperados son de tipo cadena (*string*). Ya vamos a solucionar esto. Por ahora sigamos extendiendo sus funciones.

**Ejercicio 6.7: Selector de Columnas**

La mayoría de los casos, uno no está interesado en todos los datos que contiene el archivo CSV, sino sólo en algunas columnas. Modifiquemos la función parse\_csv de modo que permita al usuario elegir (opcionalmente) algunas columnas del siguiente modo:

>>> # Lee todos los datos

>>> camion = parse\_csv('../Data/camion.csv')

>>> camion

[{'nombre': 'Lima', 'cajones': '100', 'precio': '32.2'}, {'nombre': 'Naranja', 'cajones': '50', 'precio': '91.1'}, {'nombre': 'Caqui', 'cajones': '150', 'precio': '103.44'}, {'nombre': 'Mandarina', 'cajones': '200', 'precio': '51.23'}, {'nombre': 'Durazno', 'cajones': '95', 'precio': '40.37'}, {'nombre': 'Mandarina', 'cajones': '50', 'precio': '65.1'}, {'nombre': 'Naranja', 'cajones': '100', 'precio': '70.44'}]

>>> # Lee solo algunos datos

>>> cajones\_retenidos = parse\_csv('../Data/camion.csv', select=['nombre','cajones'])

>>> cajones\_retenidos

[{'nombre': 'Lima', 'cajones': '100'}, {'nombre': 'Naranja', 'cajones': '50'}, {'nombre': 'Caqui', 'cajones': '150'}, {'nombre': 'Mandarina', 'cajones': '200'}, {'nombre': 'Durazno', 'cajones': '95'}, {'nombre': 'Mandarina', 'cajones': '50'}, {'nombre': 'Naranja', 'cajones': '100'}]

>>>

Vimos un ejemplo de un selector de columnas en el [Ejercicio 4.11](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/03_Comprension_Listas.md#ejercicio-411-extraer-datos-de-un-archivo-csv). De todos modos, ésta es otra forma de resolverlo:

# fileparse.py

import csv

def parse\_csv(nombre\_archivo, select = None):

'''

Parsea un archivo CSV en una lista de registros.

Se puede seleccionar sólo un subconjunto de las columnas, determinando el parámetro select, que debe ser una lista de nombres de las columnas a considerar.

'''

with open(nombre\_archivo) as f:

filas = csv.reader(f)

# Lee los encabezados del archivo

encabezados = next(filas)

# Si se indicó un selector de columnas,

# buscar los índices de las columnas especificadas.

# Y en ese caso achicar el conjunto de encabezados para diccionarios

if select:

indices = [encabezados.index(nombre\_columna) for nombre\_columna in select]

encabezados = select

else:

indices = []

registros = []

for fila in filas:

if not fila: # Saltear filas vacías

continue

# Filtrar la fila si se especificaron columnas

if indices:

fila = [fila[index] for index in indices]

# Armar el diccionario

registro = dict(zip(encabezados, fila))

registros.append(registro)

return registros

Esta parte es un toque técnica y merece una mirada de más cerca. El paso más delicado es traducir los nombres de las columnas seleccionadas a índices. Por ejemplo, supongamos que los encabezados en el archivo de entrada fueran los siguientes:

>>> encabezados = ['nombre', 'dia', 'hora', 'cajones', 'precio']

>>>

Y que las columnas seleccionadas fueran:

>>> select = ['nombre', 'cajones']

>>>

Para hacer la selección correctamente, tenés que conventir los nombres de las columnas listadas en select a índices (posiciones) de columnas en el archivo. Esto es exactamente lo que hace este paso:

>>> indices = [encabezados.index(nombre\_columna) for nombre\_columna in select ]

>>> indices

[0, 3]

>>>

En otras palabras, "nombre" es la columna 0 y "cajones" es la columna 3. Al leer una línea de datos del archivo, usás los índices para filtrarla y rescatar sólo las columnas que te interesan:

>>> linea = ['Lima', '6/11/2007', '9:50am', '100', '32.20' ]

>>> linea = [ linea[indice] for indice in indices ]

>>> linea

['Lima', '100']

>>>

**Ejercicio 6.8: Conversión de tipo**

Modificá la función parse\_csv() de modo que permita, opcionalmente, convertir el tipo de los datos recuperados antes de devolverlos.

>>> camion = parse\_csv('../Data/camion.csv', types=[str, int, float])

>>> camion

[{'nombre': 'Lima', 'cajones': 100, 'precio': 32.2}, {'nombre': 'Naranja', 'cajones': 50, 'precio': 91.1}, {'nombre': 'Caqui', 'cajones': 150, 'precio': 103.44}, {'nombre': 'Mandarina', 'cajones': 200, 'precio': 51.23}, {'nombre': 'Durazno', 'cajones': 95, 'precio': 40.37}, {'nombre': 'Mandarina', 'cajones': 50, 'precio': 65.1}, {'nombre': 'Naranja', 'cajones': 100, 'precio': 70.44}]

>>> cajones\_lote = parse\_csv('../Data/camion.csv', select=['nombre', 'cajones'], types=[str, int])

>>> cajones\_lote

[{'nombre': 'Lima', 'cajones': 100}, {'nombre': 'Naranja', 'cajones': 50}, {'nombre': 'Caqui', 'cajones': 150}, {'nombre': 'Mandarina', 'cajones': 200}, {'nombre': 'Durazno', 'cajones': 95}, {'nombre': 'Mandarina', 'cajones': 50}, {'nombre': 'Naranja', 'cajones': 100}]

>>>

Ya vimos esto en el [Ejercicio 4.12](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/04_Objetos.md#ejercicio-412-datos-de-primera-clase). Vas a necesitar insertar la siguiente porción de código en tu implementación:

...

if types:

fila = [func(val) for func, val in zip(types, fila) ]

...

**Ejercicio 6.9: Trabajando sin encabezados**

Algunos archivos CSV no tiene información de los encabezados. Por ejemplo, el archivo precios.csv se ve así:

Lima,40.22

Uva,24.85

Ciruela,44.85

Cereza,11.27

...

Modificá la función parse\_csv() de forma que (opcionalmente) pueda trabajar con este tipo de archivos, creando tuplas en lugar de diccionarios cuando no haya encabezados. Por ejemplo:

>>> precios = parse\_csv('../Data/precios.csv', types=[str,float], has\_headers=False)

>>> precios

[(Lima,40.22), (Uva,24.85), (Ciruela,44.85), (Cereza,11.27), (Frutilla,53.72), (Caqui,105.46), (Tomate,66.67), (Berenjena,28.47), (Lechuga,24.22), (Durazno,73.48), (Remolacha,20.75), (Habas,23.16), (Frambuesa,34.35), (Naranja,106.28), (Bruselas,15.72), (Batata,55.16), (Rúcula,36.9), (Radicheta,26.11), (Repollo,49.16), (Cebolla,58.99), (Cebollín,57.1), (Puerro,27.58), (Mandarina,80.89), (Ajo,15.19), (Rabanito,51.94), (Zapallo,24.79), (Espinaca,52.61), (Acelga,29.26), (Zanahoria,49.74), (Papa,69.35)]

>>>

Para hacer este cambio, vas a tener que modificar el código de forma que, si le pasás el parámetro has\_headers = False, la primera línea de datos no sea interpretada como encabezado. Además, en ese caso, vas a tener que asegurarte de no crear diccionarios, dado que no tenés más los nombres de las columnas para usar en el encabezado. Vale aclarar que este parámetro debe tener como valor por omisión True, con lo que la función sigue funcionando igual que antes si no se especifica has\_headers = False.

Si bien no es difícil, este es un cambio muy grande en esta función. Un camino posible es poner un if has\_headers al principio y resolver cada caso por separado. Otro camino es poner condicionales en cada paso donde sea necesario operar de manera diferente.

Incorporá todos estos cambios en el archivo fileparse.py.

**Comentario**

Llegaste lejos. Hasta este punto creaste una biblioteca de funciones que es genuinamente útil. La podés usar para parsear archivos CSV de formato arbitrario, eligiendo las columnas relevantes y cambiando el tipo de datos devuelto, todo esto sin tener que preocuparte mucho por el manejo de archivos o entender cómo funciona el módulo csv.

**6.4 Módulos**

En esta sección vamos a introducir conceptos que nos permiten crear módulos y trabajar con programas cuyas partes están repartidas en múltiples archivos.

**Módulos y la instrucción import**

Todos los archivos con código Python son módulos.

# foo.py

def grok(a):

...

def spam(b):

...

El comando import carga un módulo y lo *ejecuta*.

# programa.py

import foo

a = foo.grok(2)

b = foo.spam('Hola')

...

**Namespaces**

Se puede decir que un módulo es una colección de valores asignados a nombres. A ésto se lo llama un *namespace* (espacio de nombres). Es el contexto en el cual esos nombres existen: todas las variables globales y las funciones definidas en un módulo *pertenecen* a ese módulo. Una vez importado, el nombre del módulo se usa como un prefijo al nombrar esas variables y funciones. Por eso se llama un namespace.

import foo

a = foo.grok(2)

b = foo.spam('Hello')

...

El nombre del módulo es el nombre del archivo que lo contiene.

**Definiciones globales**

El espacio de nombres contiene todo aquello definido con visibilidad *global*. Supongamos dos módulos diferentes que definen cada uno una variable x:

# foo.py

x = 42

def grok(a):

...

# bar.py

x = 37

def spam(a):

...

Entonces hay dos definiciones de x y cada una refiere a una variable diferente. Una de ellas es foo.x y la otra es bar.x. De este modo, diferentes módulos tienen la libertad de definir variables con el mismo nombre sin que existan ambigüedades ni conflictos.

**Los módulos están aislados uno de otro.**

**Módulos como entornos**

Los módulos crean un entorno que contiene a todo el código definido ahí.

# foo.py

x = 42

def grok(a):

print(x)

Incluso las variables *globales* son visibles sólo dentro del módulo en que fueron definidas (el mismo archivo). Cada módulo es un pequeño universo.

**Ejecución de módulos**

Cuando importás un módulo se ejecutan *todas* las instrucciones en ese módulo, una tras otra, hasta llegar al final del archivo. El *namespace* del módulo está poblado por todas las funciones y variables globales cuya definición siga vigente al terminar de ejecutar el módulo. Si existen comandos que se ejecutan en el *namespace* global del módulo y hacen tareas como crear archivos, imprimir mensajes, etc., se van a ejecutar al importar el módulo.

**El comando import as**

En el momento de importar un módulo, podés cambiar el nombre que le asignás dentro del contexto en que lo importás.

import math as m

def rectangular(r, theta):

x = r \* m.cos(theta)

y = r \* m.sin(theta)

return x, y

Funciona del mismo modo que un import común salvo que, para quien lo importa, el nombre del módulo cambia.

**from módulo import nombre**

Este comando toma ciertos nombres selectos de un módulo, y los hace accesibles localmente.

from math import sin, cos

def rectangular(r, theta):

x = r \* cos(theta)

y = r \* sin(theta)

return x, y

Esta forma de importar te permite usar partes de un módulo sin necesidad de especificar la pertenencia a un módulo como prefijo. Es útil para nombres (funciones o variables) que se usan mucho.

Si usás from math import \* vas a importar *todas* las funciones y constantes del módulo math como si estuvieran definidas localmente. No es coveniente hacer esto ya que se pierden las ventajas que da trabajar con namespaces.

**Notas sobre import**

Estas distintas formas de usar import *no modifican* el funcionamiento de un módulo.

import math

# vs

import math as m

# vs

from math import cos, sin

...

Más específicamente, import siempre ejecuta el módulo completo, y los módulos siguen siendo pequeños entornos aislados. El comando import módulo as sólo cambia el nombre local del módulo. El comando from math import cos, sin, aunque sólo hace accesibles las funciones sin y cos, de todos modos carga todo el módulo y lo ejecuta. La única diferencia es que copia los nombres de las funciones sin y cos al namespace local.

**Carga de módulos**

Cada módulo es cargado y ejecutado sólo *una* vez.

*Observación: Repetir la instrucción import sólo devuelve una referencia al módulo ya cargado.*

La variable sys.modules es un diccionario de los módulos cargados.

>>> import sys

>>> sys.modules.keys()

['copy\_reg', '\_\_main\_\_', 'site', '\_\_builtin\_\_', 'encodings', 'encodings.encodings', 'posixpath', ...]

>>>

**Precaución:** Si cambiás el código de un módulo y lo volvés a cargar sucede algo que suele causar confusión hasta que lo entendés: Dado que existe la lista de módulos cargados sys.modules, un pedido de cargar un módulo por segunda vez siempre devolverá el módulo ya cargado, aún si el módulo fue modificado, si se trata de una versión nueva de ese módulo y si el archivo en disco ha sido modificado. Es posible usar reload(módulo) pero sólo en ciertos casos. El método que asegura que el módulo se vuelva a cargar es cerrar y volver a abrir el intérprete de Python.

**Ejercicios**

Para estos ejercicios que involucran módulos, es de suma importancia que te asegures de que estás ejecutando Python en el directorio adecuado.

**Ejercicio 6.10: Importar módulos**

En el [Ejercicio 6.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/03_Funciones.md#ejercicio-66-parsear-un-archivo-csv) creamos una función llamada parse\_csv() para parsear el contenido de archivos de datos en formato CSV. Ahora vamos a ver cómo usar esa función en otros programas.

Empezá por copiarte los archivos rebotes.py, hipoteca.py y fileparse.py a la carpeta de ejercicios de esta clase. Los vamos a importar.

Con el directorio de trabajo adecuado (puede que tengas que reiniciar tu intérprete para que tome efecto un cambio), intentá importar los programas que escribiste antes. Con sólo importarlos deberías ver su salida exactamente como cuando los terminaste de escribir.

Repetimos: al importar un módulo ejecutás su código.

>>> import rebotes

#... mirá la salida ...

>>> import hipoteca

#... mirá la salida ...

>>> import informe\_funciones

#... mirá la salida ...

>>>

Si nada de esto funciona, es probable que estés ejecutando Python desde la carpeta equivocada.

Ahora probá importar tu módulo fileparse y pedile help.

>>> import fileparse

>>> help(fileparse)

... mirá la salida ...

>>> dir(fileparse)

... mirá la salida ...

>>>

Intentá usar el módulo para leer datos de un archivo:

>>> camion = fileparse.parse\_csv('../Data/camion.csv', select = ['nombre', 'cajones', 'precio'], types = [str, int, float])

>>> camion

... mirá la salida ...

>>> lista\_precios = fileparse.parse\_csv('../Data/precios.csv', types = [str, float], has\_headers = False)

>>> lista\_precios

... mirá la salida ...

>>> precios = dict(lista\_precios)

>>> precios

... fijate la salida ...

>>> precios['Naranja']

106.28

>>>

Importá sólo la función para evitar escribir el nombre del módulo:

>>> from fileparse import parse\_csv

>>> camion = parse\_csv('../Data/camion.csv', select = ['nombre', 'cajones', 'precio'], types = [str, int, float])

>>> camion

... fijate la salida ...

>>>

**Ejercicio 6.11: Usemos tu módulo**

En el [Ejercicio 6.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/02_Scripts.md#ejercicio-64-estructurar-un-programa-como-una-colecci%C3%B3n-de-funciones) escribiste un programa informe\_funciones.py que produce un informe como éste:

Nombre Cajones Precio Cambio

---------- ---------- ---------- ----------

Lima 100 $32.20 8.02

Naranja 50 $91.10 15.18

Caqui 150 $103.44 2.02

Mandarina 200 $51.23 29.66

Durazno 95 $40.37 33.11

Mandarina 50 $65.10 15.79

Naranja 100 $70.44 35.84

Retomá ese programa (si lo perdiste, te dejamos una versión para que la leas y la puedas usar) y modificalo de modo que todo el procesamiento de archivos de entrada de datos se haga usando funciones del módulo fileparse. Para lograr éso, importá fileparse como un módulo y cambiá las funciones leer\_camion() y leer\_precios() para que usen la función parse\_csv() .

Guiate por el ejemplo interactivo que dimos un poco más arriba. Al final, deberías obtener exactamente el mismo resultado que al principio.

**Ejercicio 6.12: Un poco más allá**

En [Ejercicio 2.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/02_Estructuras_y_Funciones/02_Funciones.md#ejercicio-26-transformar-un-script-en-una-funci%C3%B3n) escribiste el programa costo\_camion.py que lee, mediante una función llamada costo\_camion() los datos de un camión y calcula su costo.

>>> import costo\_camion

>>> costo\_camion.costo\_camion('../Data/camion.csv')

47671.15

>>>

Modificá el archivo costo\_camion.py para que use la función informe\_funciones.leer\_camion() del programa informe\_funciones.py.

**Comentario**

Al terminar este ejercicio tenés tres programas. fileparse.py contiene una función para parsear datos de archivos CSV en general, parse\_csv(). Por otra parte, informe\_funciones.py que produce un bello informe, y que contiene las funciones leer\_camion() y leer\_precios(). Finalmente, costo\_camion.py calcula el costo de un camión, pero usando la función leer\_camion() que fue escrita para el programa que genera el informe.

**6.5 Búsqueda binaria**

Hace un par de clases vimos la búsqueda secuencial de un elemento en una lista. Si la lista está previamente ordenada, ¿podemos encontrar una manera más eficiente de buscar elementos sobre ella?

**Búsqueda sobre listas ordenadas**

Si la lista está ordenada, hay una modificación muy simple que podemos hacer sobre el algoritmo de búsqueda lineal: si estamos buscando el elemento e en una lista que está ordenada de menor a mayor, en cuanto encontremos algún elemento mayor a e podemos estar seguros de que e no está en la lista, por lo que no es necesario continuar recorriendo el resto.

**Ejercicio 6.13: Búsqueda lineal sobre listas ordenadas.**

Modificá la función busqueda\_lineal(lista, e) de la [Sección 4.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md#b%C3%BAsqueda-lineal) para el caso de listas ordenadas, de forma que la función pare cuando encuentre un elemento mayor a e. Llamá a tu nueva función busqueda\_lineal\_lordenada(lista,e) y guardala en el archivo busqueda\_en\_listas.py.

En el peor caso, ¿cuál es nuestra nueva hipótesis sobre comportamiento del algoritmo? ¿Es realmente más eficiente?

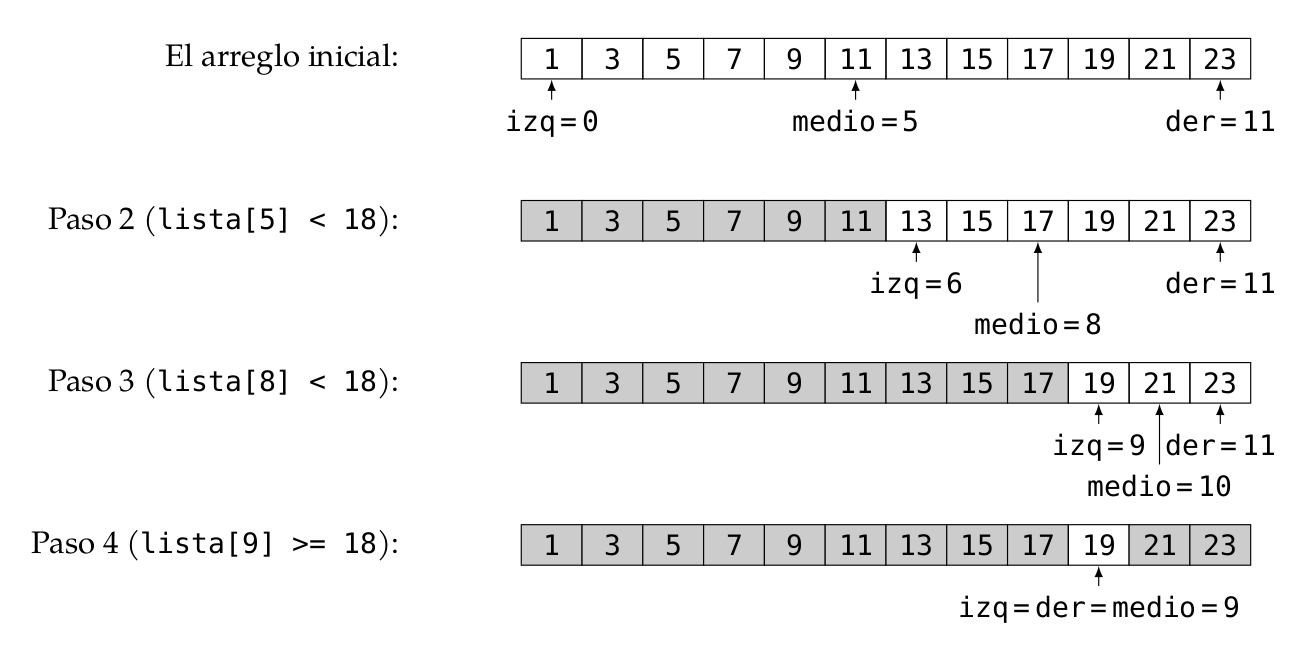
**Búsqueda binaria**

¿Podemos hacer algo mejor? Trataremos de aprovechar el hecho de que la lista está ordenada y vamos a hacer algo distinto: nuestro espacio de búsqueda se irá achicando a segmentos cada vez menores de la lista original. La idea es descartar segmentos de la lista donde el valor seguro que no puede estar:

* Consideramos como segmento inicial de búsqueda a la lista completa.
* Analizamos el punto medio del segmento (el valor central); si es el valor buscado, devolvemos el índice del punto medio.
* Si el valor central es mayor al buscado, podemos descartar el segmento que está desde el punto medio hacia la derecha.
* Si el valor central es menor al buscado, podemos descartar el segmento que está desde el punto medio hacia la izquierda.
* Una vez descartado el segmento que no nos interesa, volvemos a analizar el segmento restante, de la misma forma.
* Si en algún momento el segmento a analizar tiene longitud 0 significa que el valor buscado no se encuentra en la lista.

Para señalar la porción del segmento que se está analizando a cada paso, utilizaremos dos variables (izq y der) que contienen la posición de inicio y la posición de fin del segmento que se está considerando. De la misma manera usaremos la varible medio para contener la posición del punto medio del segmento.

A continuación ilustramos qué pasa cuando se busca el valor 18 en la lista [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23].

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/bbin.png) Ejemplo de una búsqueda usando el algoritmo de búsqueda binaria. Como no se encontró al valor buscado, devuelve -1.

El siguiente fragmento de código muestra una implementación de este algoritmo, incluyendo una instrucción de depuración (debug) con print para verificar su funcionamiento.

def busqueda\_binaria(lista, x, verbose = False):

'''Búsqueda binaria

Precondición: la lista está ordenada

Devuelve -1 si x no está en lista;

Devuelve p tal que lista[p] == x, si x está en lista

'''

if verbose:

print(f'[DEBUG] izq |der |medio')

pos = -1 # Inicializo respuesta, el valor no fue encontrado

izq = 0

der = len(lista) - 1

while izq <= der:

medio = (izq + der) // 2

if verbose:

print(f'[DEBUG] {izq:3d} |{der:>3d} |{medio:3d}')

if lista[medio] == x:

pos = medio # elemento encontrado!

if lista[medio] > x:

der = medio - 1 # descarto mitad derecha

else: # if lista[medio] < x:

izq = medio + 1 # descarto mitad izquierda

return pos

A continuación varias ejecuciones de prueba:

>>> busqueda\_binaria([1, 3, 5], 0, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 2 | 1

[DEBUG] 0 | 0 | 0

-1

>>> busqueda\_binaria([1, 3, 5], 1, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 2 | 1

[DEBUG] 0 | 0 | 0

0

>>> busqueda\_binaria([1, 3, 5], 2, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 2 | 1

[DEBUG] 0 | 0 | 0

-1

>>> busqueda\_binaria([1, 3, 5], 3, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 2 | 1

[DEBUG] 2 | 2 | 2

1

>>> busqueda\_binaria([1, 3, 5], 5, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 2 | 1

[DEBUG] 2 | 2 | 2

2

>>> busqueda\_binaria([1, 3, 5], 6, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 2 | 1

[DEBUG] 2 | 2 | 2

-1

>>> busqueda\_binaria([], 0, verbose = True)

[DEBUG] izq |der |medio

-1

>>> busqueda\_binaria([1], 1, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 0 | 0

0

>>> busqueda\_binaria([1], 3, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 0 | 0

-1

>>> busqueda\_binaria([1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23],18, verbose = True)

[DEBUG] izq |der |medio

[DEBUG] 0 | 11 | 5

[DEBUG] 6 | 11 | 8

[DEBUG] 9 | 11 | 10

[DEBUG] 9 | 9 | 9

-1

**Pregunta**: En la línea medio = (izq + der) // 2 efectuamos la división usando el operador // en lugar de /. ¿Qué pasaría su usáramos /?

**¿Cuántas comparaciones hace este programa?**

Para responder esto pensemos en el peor caso, es decir, que se descartaron varias veces partes del segmento para finalmente llegar a un segmento vacío y el valor buscado se encontró en este último paso o directamente no se encontraba en la lista.

En cada paso el segmento se divide por la mitad y se desecha una de esas mitades, y en cada paso se hace una comparación con el valor buscado. Por lo tanto, la cantidad de comparaciones que hacen con el valor buscado es aproximadamente igual a la cantidad de pasos necesarios para llegar a un segmento de tamaño 1. Veamos el caso más sencillo para razonar, y supongamos que la longitud de la lista es una potencia de 2, digamos len(lista)*= 2^k*:

1. Antes del primer paso, el segmento a tratar es de tamaño *2^k*.
2. Antes del segundo paso, el segmento a tratar es de tamaño *2^(k-1)*.
3. Antes del tercer paso, el segmento a tratar es de tamaño *2^(k-2)*. ...
4. Antes del paso *k*, el segmento a tratar es de tamaño *2^(k-k)=2^0=1*.

Por lo tanto este programa hace a lo sumo (en el peor caso) *k* comparaciones con el valor buscado cuando len(lista)*= 2^k*. Pero si despejamos *k* de la ecuación anterior, podemos ver que este programa realiza aproximadamente log2(len(lista)) comparaciones.

Cuando len(lista) no es una potencia de 2 el razonamiento es menos prolijo, pero también vale que este programa realiza aproximadamente log2(len(lista)) comparaciones.

**Comparación entre ambos métodos**

Veamos un ejemplo para entender cuánto más eficiente es la búsqueda binaria. Supongamos que tenemos una lista con un millón de elementos.

1. El algoritmo de búsqueda lineal hará, en el peor caso, un millón de comparaciones. Este caso se da si el elemento buscado no está en la lista o está en la última posición. Como se ve la cantidad de operaciones es proporcional al largo de la lista. Si el elemento buscado está en la lista el algoritmo realizará, en promedio, 500,000 comparaciones.
2. El algoritmo de búsqueda binaria hará como máximo *log2(1,000,000)* comparaciones, o sea ¡no más que 20 comparaciones!.

*Conclusión*: Si una lista está previamente ordenada, podemos utilizar el algoritmo de búsqueda binaria, cuyo comportamiento es proporcional al *logaritmo* de la cantidad de elementos de la lista, y por lo tanto muchísimo más eficiente que la búsqueda lineal, especialmente si la lista es larga.

**Ejercicio 6.14: Búsqueda binaria**

Modificando la función busqueda\_binaria(lista, x) adecuadamente, definí una función donde\_insertar(lista, x) de forma que reciba una lista ordenada y un elemento y devuelva la posición de ese elemento en la lista (si se encuentra en la lista) o la posición donde se podría insertar el elemento para que la lista permanezca ordenada (si no está en la lista).

Por ejemplo: el elemento 3 podría insertarse en la posición 2 en la lista [0,2,4,6] para mantenerla ordenada. Por lo tanto, el llamado donde\_insertar([0,2,4,6], 3) deberá devolver 2, al igual que el llamado donde\_insertar([0,2,4,6], 4).

Guarda tu modificación en un archivo bbin.py.

**6.6 Complejidad de algoritmos**

**Resumen de algoritmos de Búsqueda**

1. La búsqueda de un elemento en una secuencia es un algoritmo básico pero importante. El problema que intenta resolver puede plantearse de la siguiente manera: Dada una secuencia de valores y un valor, devolver el índice del valor en la secuencia, si se encuentra, de no encontrarse el valor en la secuencia señalarlo apropiadamente.
2. Una de las formas de resolver el problema es mediante la **búsqueda lineal**, que consiste en ir revisando uno a uno los elementos de la secuencia y comparándolos con el elemento a buscar. Este algoritmo no requiere que la secuencia se encuentre ordenada, la cantidad de comparaciones que realiza es proporcional a len(secuencia).
3. Cuando la secuencia sobre la que se quiere buscar está ordenada, se puede utilizar el algoritmo de **búsqueda binaria**. Al estar ordenada la secuencia, se puede desacartar en cada paso la mitad de los elementos, quedando entonces con una eficiencia algorítmica proporcional a log2(len(secuencia)).

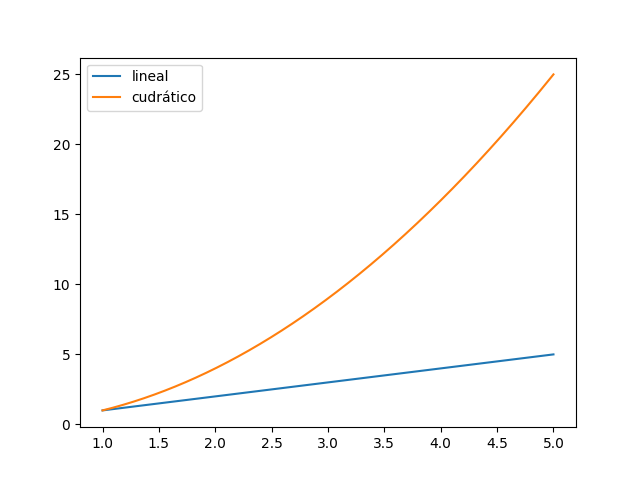
El análisis del comportamiento de un algoritmo puede ser muy engañoso si se tiene en cuenta el mejor caso, por eso suele ser mucho más ilustrativo tener en cuenta el **peor caso**. En algunos casos particulares podrá ser útil tener en cuenta, además, el **caso promedio**.

**Complejidad de algoritmos**

En las ciencias de la computación, el análisis de algoritmos es el proceso que permite determinar la complejidad de un algoritmo. Esta complejidad está típicamente medida en unidades de tiempo, o, análogamente, en la cantidad de operaciones que realiza el procesador antes de dar la respuesta. Esto permite comparar la eficiencia de diferentes algoritmos. Optimizar la eficiencia de los algoritmos es central en la tarea de un buen programador. Un algoritmo ineficiente puede no servir para nada. Esto es obvio en algoritmos que corren en tiempo real (imaginemos un algoritmo que conduce un vehículo y tarda demasiado en detectar a un peatón), pero también es importante en otros algoritmos.

Escribir programas eficientes no es una tarea sencilla. Muchas veces, las soluciones más directas no son las más eficientes. Los algoritmos más eficientes suelen aprovechar sutilezas que no son simples de comprender de un vistazo. En muchos casos les programadores deben incrementar la complejidad conceptual de un algoritmo para disminuír la complejidad computacional.

Por ejemplo, buscar la posición de un número en una lista recorriendo la lista lugar a lugar (búsqueda secuencial) demanda una cantidad de operaciones proporcional a la longitud de la lista (por cada elemento de la lista hacemos algunas operaciones fijas: comparar el elemento contra la clave, incrementar un contador, etc). Solemos decir que el algoritmo de búsqueda secuencial tiene un complejidad lineal en la longitud de la lista (ya que toma un tiempo f(n), donde f es una función lineal en n, la cantidad de elementos de la lista). No vamos a preocuparnos aquí si f (n) = 3 · n + 5 ó f (n) = 2 · n + 18. No nos importan las constantes: simplemente diremos que f (n) es lineal en n. En la literatura esto se escribe f(n) = O(n) y se lee *'la función f tiene orden n'*, o *'f es un O de n'*.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/lin_cuad.png)

En cambio, la búsqueda binaria que vimos anteriormente, si bien es conceptualmente más compleja, resulta mucho más eficiente. Dada una clave y una lista ordenada, este algoritmo aprovecha el orden de la lista para no tener que comparar la clave con todos los elementos. En un primer paso compara con el elemento central de la lista y descarta toda una mitad de la lista realizando una sola comparación. No es obvio cómo calcular la complejidad de este método, pero explicamos que si la lista tiene longitud n = 2^k, el algoritmo de búsqueda binaria realiza a lo sumo log2(n) + 1 = k + 1 comparaciones antes de dar la respuesta (hacé un ejemplo con n = 2^3 = 8 o n = 2^4 = 16 para convencerte). En general, procediendo de esta forma el algoritmo encuentra la posición de la clave en O(log2(n)) pasos. Decimos en este caso que el algoritmo requiere tiempo logarítmico.

Comparando la funcion f(n) = n con g(n) = log2(n) para valores grandes de n resulta claro que la búsqueda binaria es mucho más eficiente que la búsqueda secuencial para listas ordendas. En la próxima sección te vamos a proponer que hagas esta comparación gráficamente.

**Un algoritmo cuadrático**

Para dar un último ejemplo, supongamos que dada una lista de números (de longitud n) y un valor m queremos ver si m = p · q con p y q en la lista dada. Consideremos el siguiente algoritmo:

for p in lista :

for q in lista :

if m == p \* q :

print ( " %d= %d\* %d " %(m, p , q ) )

Este algoritmo realiza una comparación ( m == p\*q ) para cada elemento p y cada elemento q de la lista. Es decir, realiza n\*n = n^2 comparaciones. Es un algoritmo cuadrático. Su complejidad es O(n^2).

**Complejidad en el peor caso**

El término análisis de algoritmos fue acuñado por Donald Knuth, uno de los fundadores de las ciencias de la computación. El análisis de algoritmos es una parte de la teoría de la complejidad computacional que no solo estudia la complejidad de los algoritmos sino de los problemas computacionales (la pregunta general de la teoría de la complejidad no sería cuál es la complejidad de la búsqueda secuencial o binaria, sino cuál es la complejidad mínima que puede tener un algoritmo que realice la tarea de buscar un elemento en una lista ordenada). En general, y sin mencionarlo, hablamos de la complejidad en el peor caso de un algoritmo. En algunos casos puede ocurrir que la búsqueda secuencial sea más eficiente que la búsqueda binaria (por ejemplo, considerá el caso en que el elemento buscado es justo el primer elemento de la lista, ¿cuánto tarda cada método?). Al hablar de la complejidad de una algoritmo (salvo que se mencione otra cosa) hablamos del tiempo que tarda ese algoritmo en el peor caso posible.

**Estructuras de datos y Tipos Abstractos de Datos**

El diseño de un algoritmo eficiente para resolver un problema requiere comprender profundamente los datos que este algoritmo manipulará para poder diseñar adecuadamente las estructuras de datos que los contendrán. El diseño de algoritmos eficientes requiere del diseño simultáneo de algoritmos y estructuras de datos adecuadas. Diferentes estructuras de datos son adecuadas para diferentes tipos de aplicaciones y algunas estructuras están diseñadas especialmente para un problema concreto. Una estructura de datos eficiente puede ser la clave para el diseño de un algoritmo eficiente.

La estructura lógica de las estructuras de datos se llaman Tipos Abstractos de Datos (TAD). Estos TAD son el modelo matemático de las estructuras de datos. Un TAD es una abstracción del tipo de datos: define su comportamiento desde el punto de vista de le usuarie pero no dice cómo lo hace, no se mete en la implementación. Una estructura de datos concreta surge idealmente de la implementación de un TAD.

**Ejercicios:**

**Ejercicio 6.15: Insertar un elemento en una lista**

Uno de los problemas de la búsqueda binaria es que requiere que la lista esté ordenada. Si la lista se encuentra ordenada podemos mantener el orden evitando adjuntar nuevos elementos de forma desordenada.

Usando lo que hiciste en el [Ejercicio 6.14](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/05_BusqBinaria.md#ejercicio-614-b%C3%BAsqueda-binaria), agregale al archivo bbin.py una función insertar(lista, x) que reciba una lista ordenada y un elemento. Si el elemento se encuentra en la lista solamente devuelve su posición; si no se encuentra en la lista, lo inserta en la posición correcta para mantener el orden. En este segundo caso, también debe devolver su posición.

**Ejercicio 6.16: Cálcular la complejidad de dos resoluciones de propagar**

Ahora que tenés algunas herramientas teóricas más, volvé a leer las dos versiones de propagar del [Ejercicio 6.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/01_Repaso.md#ejercicio-61-propagar-por-vecinos) y el [Ejercicio 6.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/01_Repaso.md#ejercicio-62-propagar-por-como-el-auto-fant%C3%A1stico) y compará sus complejidades.

**Secuencias binarias**

Para nosotres, una **secuencia binaria** es una lista que contiene solo 0’s y 1’s. Por ejemplo s = [0, 1, 0, 0, 1] es una secuencia binaria de longitud 5. La *primera* secuencia binaria de esa longitud es [0, 0, 0, 0, 0], mientras que *la última* es [1, 1, 1, 1, 1]. Cada secuencia tiene una *siguiente* (salvo la última). No vamos a dar una definición precisa, pero escencialmente las secuencias pueden pensarse como representando números enteros en base dos y *la siguiente* secuencia es la que representa al siguiente número. Por convención, diremos que la secuencia siguiente de la última es la primera.

Ejemplos:

[0, 0, 0, 0, 0] -> [0, 0, 0, 0, 1]

[0, 0, 1, 1, 0] -> [0, 0, 1, 1, 1]

[0, 0, 1, 1, 1] -> [0, 1, 0, 0, 0]

[1, 1, 1, 1, 1] -> [0, 0, 0, 0, 0]

La función incrementar(s) calcula la secuencia siguiente de una secuencia dada:

def incrementar(s):

carry = 1

l = len(s)

for i in range(l-1,-1,-1):

if (s[i] == 1 and carry == 1):

s[i] = 0

carry = 1

else:

s[i] = s[i] + carry

carry = 0

return s

**Ejercicio 6.17: Complejidad de incrementar()**

Si tomamos n = len(s) podemos tratar de medir la complejidad de la función incrementar() en términos de la longitud n de la secuencia. ¿Te parece que incrementar() es una función lineal, cuadrática, logarítmica o exponencial? ¿Por qué?

**Ejercicio 6.18: Un ejemplo más complejo**

Por último, escribí una funcion listar\_secuencias(n) que devuelva una lista con **todas** las secuencias binarias de longitud n comenzando con la primera ([0]\*n) y usando en cada paso la función incrementar() definida más arriba. ¿Cuántas listas hay de longitud n? ¿Y de longitud n+1?

¿Podés correr listar\_secuencias(15)? ¿Y listar\_secuencias(20)? ¿Hasta cúanto llegas a correr en un tiempo razonable?

¿Te parece que listar\_secuencias(n) es una función lineal, cuadrática, logarítmica o exponencial en n? ¿Por qué?

**6.7 Gráficos de complejidad**

**Contar la cantidad de operaciones de un algoritmo**

La siguiente función realiza una búsqueda secuencial de un elemento en una lista. Devuelve la posición del elemento si lo encuentra y -1 si no lo encuentra.

def busqueda\_secuencial(lista, x):

'''Si x está en la lista devuelve el índice de su primera aparición,

de lo contrario devuelve -1.

'''

pos = -1

for i,z in enumerate(lista):

if z == x:

pos = i

break

return pos

Esta modificación de la función cuenta (y devuelve) además cuántas comparaciones (z == x) hace la función. Observá que devuelve un par de datos.

def busqueda\_secuencial\_(lista, x):

'''Si x está en la lista devuelve el índice de su primera aparición,

de lo contrario devuelve -1. Además devuelve la cantidad de comparaciones

que hace la función.

'''

comps = 0 # inicializo en cero la cantidad de comparaciones

pos = -1

for i,z in enumerate(lista):

comps += 1 # sumo la comparación que estoy por hacer

if z == x:

pos = i

break

return pos, comps

Si querés acceder a la posición podés usar busqueda\_secuencial\_(lista, x)[0] y para acceder a la cantidad de comparaciones que hizo busqueda\_secuencial\_(lista, x)[1].

**Ejercicio 6.19: Contar comparaciones en la búsqueda binaria**

Modificá el código de búsqueda binaria (busqueda\_binaria(lista, x)) introducido en la [Sección 6.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/05_BusqBinaria.md#b%C3%BAsqueda-binaria), de forma que devuelva (además de la posición del elemento en la lista) la cantidad de comparaciones que realizó el algoritmo para encontrarlo o decidir que no está.

**Gráficar la cantidad de comparaciones promedio**

La siguiente función generar\_lista(n, m) devuelve una lista ordenada de n elementos diferentes entre 0 y m-1, mientras que generar\_elemento(m) devuelve un elemento aleatorio en el mismo rango de valores.

import random

def generar\_lista(n, m):

l = random.sample(range(m), k = n)

l.sort()

return l

def generar\_elemento(m):

return random.randint(0, m-1)

Dada una lista ya generada, digamos que un *experimento elemental* es generar un elemento, buscarlo en la lista y contar la cantidad de comparaciones realizadas. Esta cantidad de operaciones es el *resultado* del experimento elemental.

m = 10000

n = 100

lista = generar\_lista(n, m)

# acá comienza el experimento

x = generar\_elemento(m)

comps = busqueda\_secuencial\_(lista, x)[1]

Entonces, el siguiente código da la cantidad de comparaciones *promedio* en k experimentos elementales. Observá que hay muchas variables diferentes dando vueltas: n, m y k.

m = 10000

n = 100

k = 1000

lista = generar\_lista(n, m)

def experimento\_secuencial\_promedio(lista, m, k):

comps\_tot = 0

for i in range(k):

x = generar\_elemento(m)

comps\_tot += busqueda\_secuencial\_(lista,x)[1]

comps\_prom = comps\_tot / k

return comps\_prom

Como las listas tienen n = 100 elementos y estoy buscando un número cualquiera entre m números diferentes, es casi seguro que no lo voy a encontrar y que voy a tener que recorrer toda la lista para concluir esto (aunque en algún caso puede ser que esté y lo encuentre antes de recorrerla toda!). Entonces el promedio de comparaciones va a dar cercano al largo n de la lista, quizás un poco menor. Tiene una componente aleatoria, es un *experimento* numérico.

Si decíamos que buscar un elemento era un *experimento elemental* digamos que repetir *k* experimentos elementales y calcular el promedio de comparaciones es un *experimento de promedios*.

Grafiquemos los resultados de estos *experimentos de promedios* para diferentes listas de largos n entre 1 y 256. Es decir, estaremos graficando la cantidad de comparaciones que hace en promedio el algoritmo de búsqueda secuencial sobre una lista de largo n, para diferentes valores de n.

import matplotlib.pyplot as plt

import numpy as np

m = 10000

k = 1000

largos = np.arange(256) + 1 # estos son los largos de listas que voy a usar

comps\_promedio = np.zeros(256) # aca guardo el promedio de comparaciones sobre una lista de largo i, para i entre 1 y 256.

for i, n in enumerate(largos):

lista = generar\_lista(n, m) # genero lista de largo n

comps\_promedio[i] = experimento\_secuencial\_promedio(lista, m, k)

# ahora grafico largos de listas contra operaciones promedio de búsqueda.

plt.plot(largos,comps\_promedio,label = 'Búsqueda Secuencial')

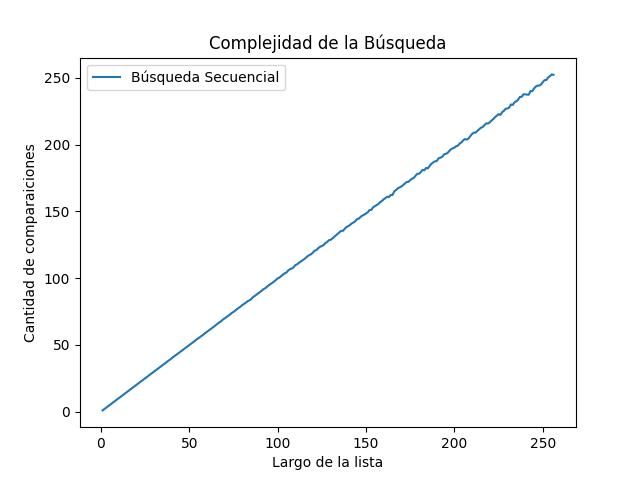
plt.xlabel("Largo de la lista")

plt.ylabel("Cantidad de comparaciones")

plt.title("Complejidad de la Búsqueda")

plt.legend()

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/compl_sec.png)

En la próxima clase estudiaremos en detalle la librería matplotlib que ya empezamos a usar la clase pasada. Por ahora solo agregamos la función plot(x, y) a la que se le pasan dos vectores (o listas) x e y y realiza una gráfico de líneas uniendo los puntos con esas coordenadas. El parámetro label permite ponerle un nombre a la curva que se muestra luego con la función plt.legend().

Este gráfico parece medio sonso, pero en el próximo ejercicio va a ir tomando color.

**Ejercicio 6.20: Búsqueda binaria vs. búsqueda secuencial**

En este Ejercicio vamos a rehacer los gráficos del ejemplo anterior, pero primero cambiando el algoritmo de búsqueda y luego comparando ambos algoritmos.

1. Usando experimento\_secuencial\_promedio(lista, m, k) como base, escribí una función experimento\_binario\_promedio(lista, m, k) que cuente la cantidad de comparaciones que realiza en promedio (entre k experimentos elementales) la búsqueda binaria sobre la lista pasada como parámetro.
2. Graficá los resultados de estos experimentos para listas de largo entre 1 y 256.
3. Graficá ambas curvas en una misma figura, nombrando adecuadamente las curvas, los ejes y la figura completa. Jugá con xlim e ylim para visualizar bien las dos curvas, aunque tengas que restringir el rango.
4. ¿Qué observas en estos gráficos? ¿Qué podés decir sobre la complejidad de cada algoritmo? ¿Son similares?

El código de este ejercicio guardalo en plot\_bbin\_vs\_bsec.py.

**6.8 Cierre de la clase**

En esta clase trabajamos con funciones y creamos módulos. También aprendimos algunas nociones de complejidad de algoritmos, estudiamos la búsqueda binaria y comparamos su performance con la de la búsqueda secuencial..

Para cerrar esta clase te pedimos dos cosas:

* Que recopiles las soluciones de los siguientes ejercicios:

1. El archivo fileparse.py del [Ejercicio 6.8](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/03_Funciones.md#ejercicio-68-conversi%C3%B3n-de-tipo) o del siguiente.
2. El archivo informe\_funciones.py de [Ejercicio 6.11](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/04_Modulos.md#ejercicio-611-usemos-tu-m%C3%B3dulo).
3. El archivo costo\_camion.py del [Ejercicio 6.12](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/04_Modulos.md#ejercicio-612-un-poco-m%C3%A1s-all%C3%A1).
4. El archivo bbin.py del [Ejercicio 6.15](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/06_Complejidad.md#ejercicio-615-insertar-un-elemento-en-una-lista).
5. El archivo plot\_bbin\_vs\_bsec.py del [Ejercicio 6.20](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/07_gr%C3%A1ficos_de_complejidad.md#ejercicio-620-b%C3%BAsqueda-binaria-vs-b%C3%BAsqueda-secuencial).

* Que completes [este formulario](https://docs.google.com/forms/d/1De-riQmiD2H9PZAVUkR5hf4lqU_9tp8IACqZYTvePkA) usando como identificación tu dirección de mail. Al terminar vas a obtener un link para enviarnos tus ejercicios y podrás participar de la revisión de pares.

¡Gracias!