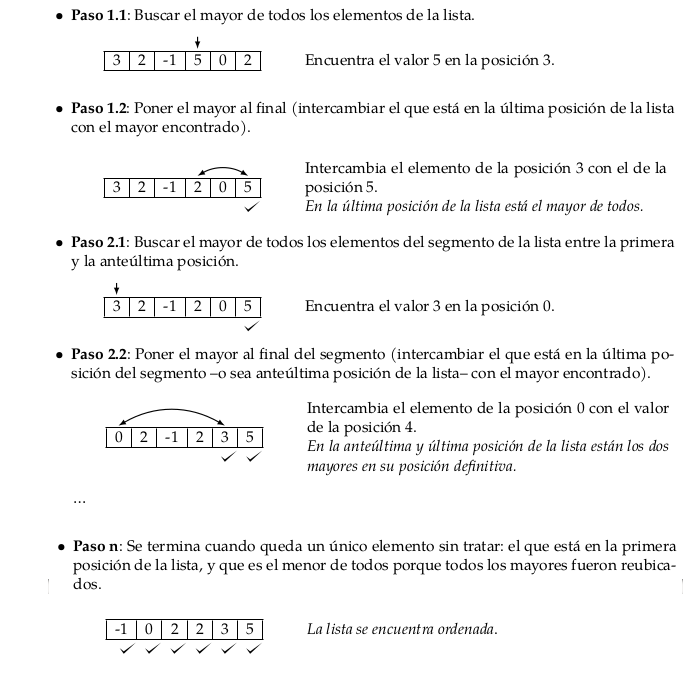
**12.2 Ordenamientos sencillos de listas**

Para esta sección tenemos un [video introductorio](https://youtu.be/ElHpgeTvZLY).

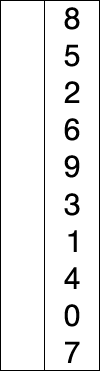
El problema del ordenamiento es tan fundamental que, a pesar de que Python ya lo hace con su método sort() por ejemplo, nos interesa discutirlo. Hay una diversidad de soluciones para ordenar listas. Vamos a empezar viendo las más sencillas de escribir (que en general suelen ser las más caras).

**Ordenamiento por selección**

El método de *ordenamiento por selección* se basa en la siguiente idea:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/seleccion.png)

La siguiente animación muestra un algoritmo de ordenamiento por selección (que busca el menor en cada paso, en lugar del mayor):

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/Selection-Sort-Animation.gif)

Una implementación en Python puede verse en el siguiente código.

def ord\_seleccion(lista):

"""Ordena una lista de elementos según el método de selección.

Pre: los elementos de la lista deben ser comparables.

Post: la lista está ordenada."""

# posición final del segmento a tratar

n = len(lista) - 1

# mientras haya al menos 2 elementos para ordenar

while n > 0:

# posición del mayor valor del segmento

p = buscar\_max(lista, 0, n)

# intercambiar el valor que está en p con el valor que

# está en la última posición del segmento

lista[p], lista[n] = lista[n], lista[p]

print("DEBUG: ", p, n, lista)

# reducir el segmento en 1

n = n - 1

def buscar\_max(lista, a, b):

"""Devuelve la posición del máximo elemento en un segmento de

lista de elementos comparables.

La lista no debe ser vacía.

a y b son las posiciones inicial y final del segmento"""

pos\_max = a

for i in range(a + 1, b + 1):

if lista[i] > lista[pos\_max]:

pos\_max = i

return pos\_max

La función principal, ord\_seleccion() es la encargada de recorrer la lista, ubicando el mayor elemento al final del segmento y luego reduciendo el segmento a analizar.

La función buscar\_max() busca el mayor elemento de un segmento de la lista y devuelve su posición.

A continuación, algunas ejecuciones de prueba de ese código:

>>> lista = [3, 2, -1, 5, 0, 2]

>>> ord\_seleccion(lista)

DEBUG: 3 5 [3, 2, -1, 2, 0, 5]

DEBUG: 0 4 [0, 2, -1, 2, 3, 5]

DEBUG: 1 3 [0, 2, -1, 2, 3, 5]

DEBUG: 1 2 [0, -1, 2, 2, 3, 5]

DEBUG: 0 1 [-1, 0, 2, 2, 3, 5]

>>> lista

[-1, 0, 2, 2, 3, 5]

>>> lista = []

>>> ord\_seleccion(lista)

>>> l = [1]

>>> ord\_seleccion(lista)

>>> lista

[1]

>>> lista = [1, 2, 3, 4, 5]

>>> ord\_seleccion(lista)

DEBUG: 4 4 [1, 2, 3, 4, 5]

DEBUG: 3 3 [1, 2, 3, 4, 5]

DEBUG: 2 2 [1, 2, 3, 4, 5]

DEBUG: 1 1 [1, 2, 3, 4, 5]

Podés observar que incluso cuando la lista ya está ordenada, se la recorre buscando los mayores elementos y ubicándolos en la misma posición en la que se encuentran.

**Invariante en el ordenamiento por selección**

Todo ordenamiento tiene un invariante que permite asegurarse de que cada paso que se toma va en la dirección de obtener una lista ordenada.

En el caso del ordenamiento por selección, el invariante es que los elementos en las posiciones desde n + 1 hasta el final de la lista están ordenados y son mayores que los elementos ubicados de 0 a n; es decir que ya están en su posición definitiva.

**¿Cuánto cuesta ordenar por selección?**

Como se puede ver en el código de la función buscar\_max, para buscar el máximo elemento en un segmento de lista se debe recorrer todo ese segmento, por lo que en nuestro caso debemos recorrer en el primer paso N elementos, en el segundo paso N-1 elementos, en el tercer paso N-2 elementos, etc. Cada visita a un elemento implica una cantidad constante y pequeña de comparaciones (que no depende de N). Por lo tanto tenemos que

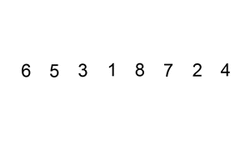
T(N) ~ c \* (2 + 3 + ... + N) ~ c \* N \* (N+1)/2 ~ N^2

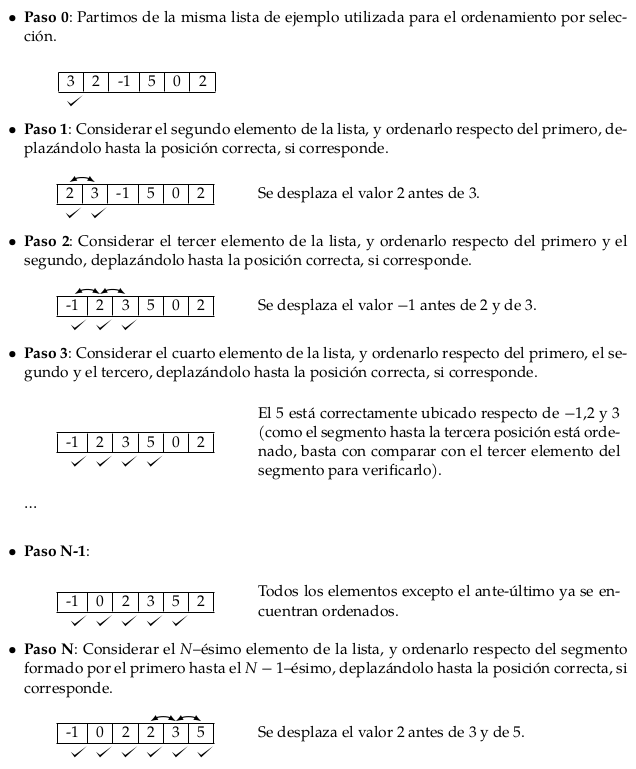
O sea que ordenar por selección una lista de tamaño N insume tiempo del orden de N^2. Como ya mencionamos, este tiempo es independiente de si la lista estaba previamente ordenada o no.

En cuanto al espacio utilizado, sólo se tiene en memoria la lista que se desea ordenar y algunas variables de tamaño 1.

**Ordenamiento por inserción**

El método de *ordenamiento por inserción* se basa en la siguiente idea:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/Insertion-sort-animation.gif)

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/insercion.png)

Una posible implementación en Python de este algoritmo se incluye en el siguiente código:

def ord\_insercion(lista):

"""Ordena una lista de elementos según el método de inserción.

Pre: los elementos de la lista deben ser comparables.

Post: la lista está ordenada."""

for i in range(len(lista) - 1):

# Si el elemento de la posición i+1 está desordenado respecto

# al de la posición i, reubicarlo dentro del segmento [0:i]

if lista[i + 1] < lista[i]:

reubicar(lista, i + 1)

print("DEBUG: ", lista)

def reubicar(lista, p):

"""Reubica al elemento que está en la posición p de la lista

dentro del segmento [0:p-1].

Pre: p tiene que ser una posicion válida de lista."""

v = lista[p]

# Recorrer el segmento [0:p-1] de derecha a izquierda hasta

# encontrar la posición j tal que lista[j-1] <= v < lista[j].

j = p

while j > 0 and v < lista[j - 1]:

# Desplazar los elementos hacia la derecha, dejando lugar

# para insertar el elemento v donde corresponda.

lista[j] = lista[j - 1]

j -= 1

lista[j] = v

La función principal, ord\_insercion(), recorre la lista desde el segundo elemento hasta el último, y cuando uno de estos elementos no está ordenado con respecto al anterior, llama a la función auxiliar reubicar(), que se encarga de colocar el elemento en la posición que le corresponde.

En la función reubicar() se busca la posición correcta donde debe colocarse el elemento, a la vez que se van corriendo todos los elementos un lugar a la derecha, de modo que cuando se encuentra la posición, el valor a insertar reemplaza al valor que se encontraba allí anteriormente.

En las siguientes ejecuciones puede verse que funciona correctamente.

>>> lista = [3, 2, -1, 5, 0, 2]

>>> ord\_insercion(lista)

DEBUG: [2, 3, -1, 5, 0, 2]

DEBUG: [-1, 2, 3, 5, 0, 2]

DEBUG: [-1, 2, 3, 5, 0, 2]

DEBUG: [-1, 0, 2, 3, 5, 2]

DEBUG: [-1, 0, 2, 2, 3, 5]

>>> lista

[-1, 0, 2, 2, 3, 5]

>>> lista = []

>>> ord\_insercion(lista)

>>> lista = [1]

>>> ord\_insercion(lista)

>>> lista

[1]

>>> lista = [1, 2, 3, 4, 5, 6]

>>> ord\_insercion(lista)

DEBUG: [1, 2, 3, 4, 5, 6]

DEBUG: [1, 2, 3, 4, 5, 6]

DEBUG: [1, 2, 3, 4, 5, 6]

DEBUG: [1, 2, 3, 4, 5, 6]

DEBUG: [1, 2, 3, 4, 5, 6]

>>> lista

[1, 2, 3, 4, 5, 6]

**Invariante del ordenamiento por inserción**

En el ordenamiento por inserción, en cada paso se satisface que los elementos que se encuentran en el segmento de 0 a i están ordenados, de manera que agregar un nuevo elemento implica colocarlo en la posición correspondiente y el segmento seguirá ordenado.

**¿Cuánto cuesta ordenar por inserción?**

Del código de ord\_insercion() se puede ver que la función principal avanza por la lista de izquierda a derecha, mientras que la función reubicar() cambia los elementos de lugar de derecha a izquierda.

Lo peor que le puede pasar a un elemento que está en la posición j es que deba ser ubicado al principio de la lista. Y lo peor que le puede pasar a una lista es que todos sus elementos deban ser reubicados.

Por ejemplo, en la lista [10, 8, 6, 2, -2, -5], todos los elementos deben ser reubicados al principio de la lista.

En el primer paso, el segundo elemento se debe intercambiar con el primero; en el segundo paso, el tercer elemento se compara con el segundo y el primer elemento, y se ubica adelante de todo; en el tercer paso, el cuarto elemento se compara con el tercero, el segundo y el primer elemento, y se ubica adelante de todo; etc...

T(N) ~ c \* (2 + 3 + \*s + N) ~ c \* N \* (N+1)/2 ~ N^2

Es decir que ordenar por inserción una lista de tamaño N puede insumir (en el peor caso) tiempo del orden de N^2 (*O(N^2)*). En cuanto al espacio utilizado, nuevamente sólo se tiene en memoria la lista que se desea ordenar y algunas variables de tamaño 1.

**Inserción en una lista ordenada**

Resulta interesante observar que cuando la lista de entrada se encuentra ordenada, este algoritmo no hace ningún movimiento de elementos. Simplemente compara cada elemento con el anterior, y si es mayor sigue adelante.

Es decir que para el caso de una lista de N elementos que se encuentra ordenada, el tiempo que insume el algoritmo de inserción es:

T(N) ~ N.

**Resumen**

* El *ordenamiento por selección* es uno de los más sencillos, pero es bastante ineficiente: se basa en la idea de *buscar el máximo* en una secuencia, ubicarlo al final y seguir analizando la secuencia sin el último elemento.

Tiene como ventaja que hace una baja cantidad de intercambios (N), pero como desventaja que necesita una alta cantidad de comparaciones (N^2). Siempre tiene el mismo comportamiento.

* El *ordenamiento por inserción* es un algoritmo bastante intuitivo y se suele usar para ordenar en la vida real. Se basa en la idea de ir *insertando ordenadamente*: en cada paso se considera la inserción de un elemento más de secuencia y la inserción se empieza a hacer desde el final de los datos ya ordenados.

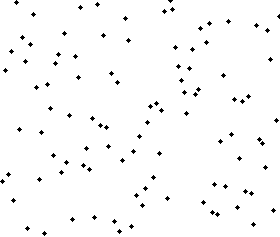
Tiene como ventaja que en el caso de tener los datos ya ordenados no hace ningún intercambio (y hace sólo N-1 comparaciones). En el peor caso, cuando la secuencia está invertida, se hace una gran cantidad de intercambios y comparaciones (N^2). Si bien es un algoritmo ineficiente, para secuencias cortas el tiempo de ejecución es bastante bueno.

**Ejercicios**

**Ejercicio 12.1:**

Describí los pasos del ordenamiento de la lista [0, 9, 3, 8, 5, 3, 2, 4] con los algoritmos de inserción y selección.

**Ejercicio 12.2: burbujeo**

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/Bubble_sort_animation.gif)

El ordenamiento por burbujeo se basa en una idea bastante sencilla. El algoritmo compara dos elementos contiguos de la lista y, si el orden es adecuado, los deja como están, si no, los intercambia. La repetición de este *paso elemental* (una burbuja) a lo largo de la lista (recorriéndola desde el comienzo hasta el final) garantiza llevar el mayor elemento al final de la lista, pero no garantiza que el menor elemento haya quedado en el primer lugar. De hecho, el menor elemento solo se mueve un paso hacia la izquierda en una recorrida completa de la lista. Es por esto que estas recorridas se repiten sucesivas veces (¿cuántas hace falta?) de manera de garantizar que el lista quede completamente ordenada.

Como en el primer paso tenemos la garantía de que el mayor elemento quedó al final de la lista, la segunda recorrida puede evitar llegar hasta esa última posición. Así, cada recorrida es más corta que la anterior. En cada recorrida se comparan todos los pares de elementos sucesivos (en el rango correspondiente) y se intercambian si no están ordenados.

Programá una función ord\_burbujeo(lista) que implemente este método de ordenamiento. ¿Cuántas comparaciones realiza esta función en una lista de largo n?

Probá tu código con las siguientes listas.

lista\_1 = [1, 2, -3, 8, 1, 5]

lista\_2 = [1, 2, 3, 4, 5]

lista\_3 = [0, 9, 3, 8, 5, 3, 2, 4]

lista\_4 = [10, 8, 6, 2, -2, -5]

lista\_5 = [2, 5, 1, 0]

Guardá tu solución en el archivo burbujeo.py comentando la complejidad del algoritmo y cómo la calculaste.

*Extra:* ¿Podés escribir una versión recursiva de este algoritmo?

**Ejercicio 12.3: ordernar a mano**

Elegí dos listas de las 5 del ejercicio anterior y ordenalas a mano (con papel y lápiz) con los 3 métodos: selección, inserción y burbujeo.

**Ejercicio 12.4: experimento con 3 métodos**

Hacé una función generar\_lista(N) que genere una lista aleatoria de largo N con números enteros del 1 al 1000 (puede haber repeticiones).

Modificá el código de las tres funciones para que cuenten cuántas comparaciones entre elementos de la lista realiza cada una. Por ejemplo, ord\_seleccion realiza comparaciones (entre elementos de la lista) sólo cuando llama a buscar\_max(lista, a, b) y en ese caso realiza b-a comparaciones.

Realizá un experimento que genere una lista de largo N y la ordene con los tres métodos (burbujeo, inserción y selección).

Para N = 10, realizá k = 100 repeticiones del siguiente experimento. Generar una lista de largo N, ordenarlas con los tres métodos y guardar la cantidad de operaciones. Al final, debe imprimir el promedio de comparaciones realizado por cada método.

*Cuidado*: usá las mismas listas para los tres métodos así la compración es justa.

**Ejercicio 12.5: comparar métodos gráficamente**

Vamos a tratar de comparar visualmente la cantidad de comparaciones que hacen estos algoritmos para diferentes largos de listas. Hacé un programa comparaciones\_ordenamiento.py que para N entre 1 y 256 genere una lista de largo N con números enteros del 1 al 1000, calcule la cantidad de comparaciones realizadas por cada método y guarde estos resultados en tres vectores de largo 256: comparaciones\_seleccion, comparaciones\_insercion y comparaciones\_burbujeo.

Graficá estos tres vectores. Si las curvas se superponen, graficá una de ellas con línea punteada para poder verlas bien. ¿Cómo dirías que crece la complejidad de estos métodos? ¿Para cuáles depende de la lista a ordenar y para cuáles solamente depende del largo de la lista?

Guardá comparaciones\_ordenamiento.py para seguir trabajando sobre él y para entregarlo.

¿Se te ocurre un algoritmo de ordenamiento que sea sustancialmente mejor que estos? Ese será el tema de la próxima sección.

*Extra:* ¿Las curvas de complejidad quedaron suaves? ¿Se te ocurre cómo hacer para suavizarlas?

**12.3 Divide y reinarás**

Para esta sección tenemos este [video introductorio](https://youtu.be/ZSRn0ob4ukM).

El problema del ordenamiento es un problema fundamental y hay [muchísimos algoritmos que lo resuelven](https://www.youtube.com/watch?v=kPRA0W1kECg). Los métodos de ordenamiento vistos en la sección anterior eran métodos iterativos cuyo tiempo de ejecución era cuadrático.

Veremos ahora el **merge sort** que es un algoritmo un poco más complejo conceptualmente pero menos complejo computacionalmente. El algoritmo está basado en una idea muy fecunda en el diseño de algoritmos eficientes que se denomina **divide y reinarás** (ó *divide and conquer* en inglés).

Divide y reinarás es un paradigma de diseño de algoritmos recursivos que trabaja partiendo (dividiendo) el problema original en subproblemas del mismo tipo pero más sencillos de resolver. Las soluciones de estos subproblemas luego se combinan para obtener una solución del problema original.

La correctitud de los algoritmos de este tipo suele probarse utilizando la inducción matemática y el cálculo de su complejidad involucra la resolución de ecuaciones de recurrencia cuyos detalles escapan el alcance de este curso.

**El algoritmo *merge sort* (u ordenamiento por mezcla)**

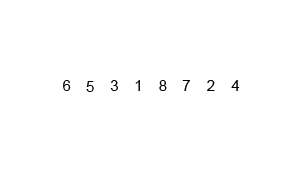
[](https://camo.githubusercontent.com/1072a8b1d58592e864ff227c7d6a171864899a6e3eca630146d039ffb40fc936/68747470733a2f2f696d672e64657672616e742e636f6d2f64657672616e742f72616e742f725f3935303433345f456a5057592e676966)

El *merge sort* se basa en la siguiente idea:

* Si la lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. De lo contrario hacer lo siguiente:
* Dividir la lista al medio, formando dos sublistas de (aproximadamente) el mismo tamaño cada una.
* Ordenar cada una de esas dos sublistas (usando este mismo método).
* Una vez que se ordenaron ambas sublistas, intercalarlas (mergearlas) de manera ordenada.

Por ejemplo, si la lista original es [6, 7, -1, 0, 5, 2, 3, 8] deberemos ordenar recursivamente [6, 7, -1, 0] y [5, 2, 3, 8] con lo cual obtendremos [-1, 0, 6, 7] y [2, 3, 5, 8]. Si intercalamos ordenadamente las dos listas ordenadas obtenemos la solución buscada: [-1, 0, 2, 3, 5, 6, 7, 8].

Veamos otro ejemplo con un gif animado:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/Merge-sort-example-300px.gif)

Diseñemos la función merge\_sort(lista):

* Si lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. Se devuelve lista original.
* De lo contrario:
  + medio = len(lista) // 2
  + izq = merge\_sort(lista[:medio])
  + der = merge\_sort(lista[medio:])
  + Se devuelve merge(izq, der).

Falta sólo diseñar la función merge: dadas dos listas ordenadas debe obtener una nueva lista que resulte de intercalar a ambas de manera ordenada:

* Utilizaremos dos índices, i y j, para recorrer cada una de las dos listas.
* Utilizaremos una tercera lista, resultado, donde almacenaremos el resultado.
* Mientras i sea menor que el largo de lista1 y j menor que el largo de lista2, significa que hay elementos para comparar en ambas listas.
* Si el menor es el de lista1:
  + Agregar el elemento lista1[i] al final de la lista resultado.
  + Incrementar el índice i.
* de lo contrario:
  + Agregar el elemento lista2[j] al final de la lista resultado.
  + Incrementar el índice j.
* Una vez que una de las dos listas se termina, simplemente hay que agregar todo lo que queda en la otra al final de la lista resultado.

El código resultante del diseño de ambas funciones puede verse a continuación:

import random

def merge\_sort(lista):

"""Ordena lista mediante el método merge sort.

Pre: lista debe contener elementos comparables.

Devuelve: una nueva lista ordenada."""

if len(lista) < 2:

lista\_nueva = lista

else:

medio = len(lista) // 2

izq = merge\_sort(lista[:medio])

der = merge\_sort(lista[medio:])

lista\_nueva = merge(izq, der)

return lista\_nueva

def merge(lista1, lista2):

"""Intercala los elementos de lista1 y lista2 de forma ordenada.

Pre: lista1 y lista2 deben estar ordenadas.

Devuelve: una lista con los elementos de lista1 y lista2."""

i, j = 0, 0

resultado = []

while(i < len(lista1) and j < len(lista2)):

if (lista1[i] < lista2[j]):

resultado.append(lista1[i])

i += 1

else:

resultado.append(lista2[j])

j += 1

# Agregar lo que falta de una lista

resultado += lista1[i:]

resultado += lista2[j:]

return resultado

El método **divide y reinarás** que hemos usado para resolver el problema de ordenar una lista puede aplicarse también en otras situaciones. Hace falta que sea posible resolver el problema partiéndolo en varios subproblemas de tamaño menor, resolver cada uno de esos subproblemas por separado aplicando la misma técnica (en nuestro caso ordenar por mezcla cada una de las dos sublistas), y finalmente juntar estas soluciones parciales en una solución completa del problema mayor (en nuestro caso la intercalación ordenada de las dos sublistas ordenadas).

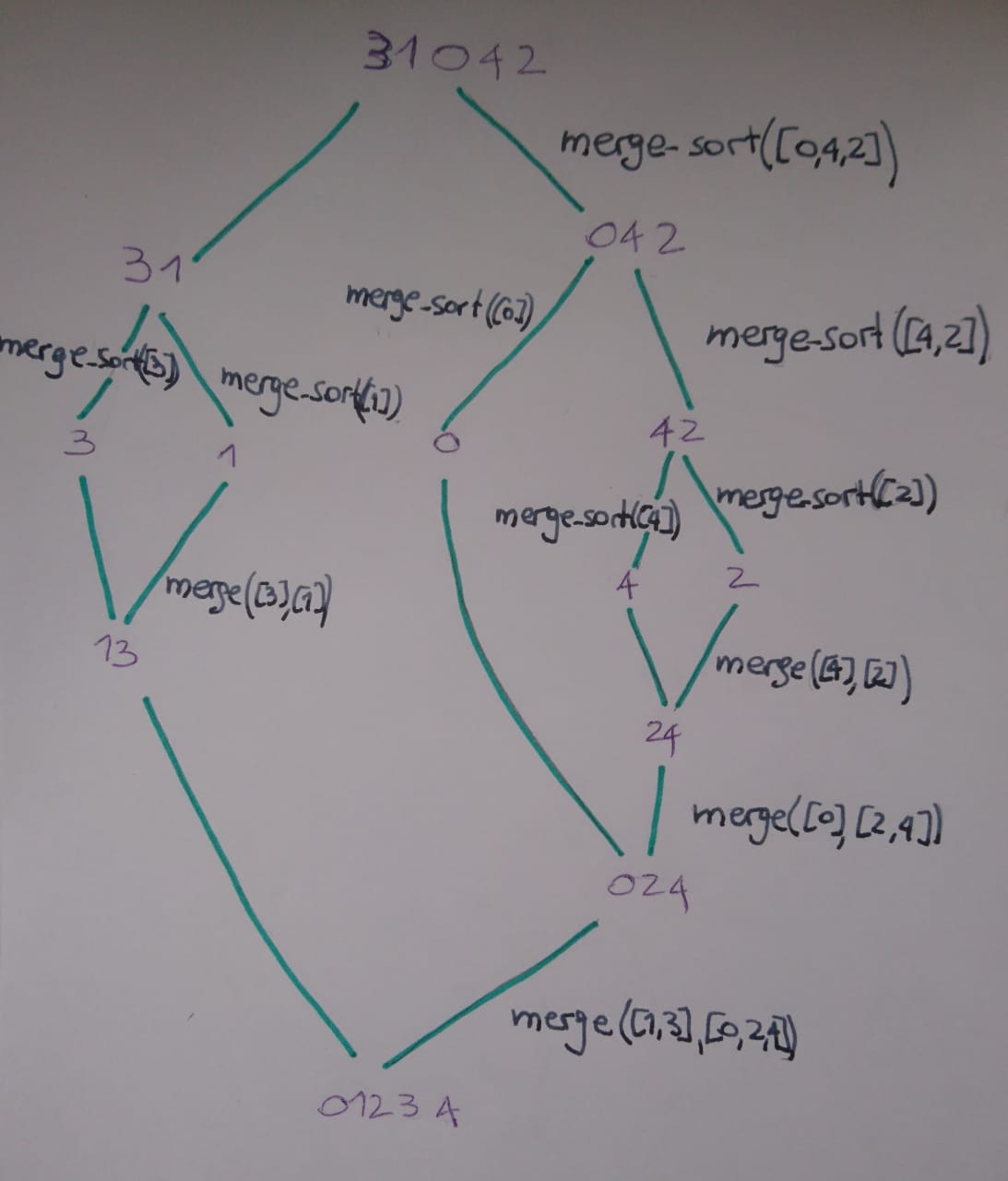
Como siempre sucede con las soluciones recursivas, debemos encontrar un caso base en el cual no se aplica la llamada recursiva (en nuestro caso: si la lista tiene largo cero o uno, ya está ordenada y no hay nada que hacer). Además debemos asegurar que siempre se alcanza el caso base, y en nuestro caso aseguramos eso porque, si no estamos en el caso base, la lista se divide en mitades decrementando su longitud.

El método **divide y reinarás** es fecundo y ha dado lugar a algoritmos muy eficientes para tareas muy disímiles como multiplicar matrices, calcular la transformada de Fourier o realizar análisis sintácticos (parsear).

**Ejemplo: Árbol de recursión**

Para representar gráficamente las llamadas recursivas de la función merge\_sort() podemos hacer un árbol similar al que mostramos en la [Sección 11.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/02_Recursion.md#un-ejemplo-de-recursi%C3%B3n-poco-eficiente) para la sucesión de Fibonacci. Esta vez, como queremos mostrar no sólo las llamadas sino también lo que devuelve cada llamada, espejaremos el árbol obteniendo un grafo.

Acá mostramos el árbol de recursión de merge\_sort([3, 1, 0, 4, 2]).

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/arbol_recursion.jpg)

**¿Cuánto cuesta el *Merge sort*?**

Supongamos que tenemos que ordenar una lista de l longitud N . Observamos lo siguiente:

* Para intercalar dos listas de longitud N/2 hace falta recorrer ambas listas que en total tienen N elementos. La cantidad de operaciones resulta proporcional a N. Llamemos a \* N a ese tiempo.
* Si llamamos T(N) al tiempo que tarda el algoritmo en ordenar una lista de longitud N, vemos que T(N) = 2 \* T(N/2) + a \* N.
* Además, cuando la lista es pequeña, la operación es de tiempo constante: T(1) = T(0) = b.

Para simplificar la cuenta vamos a suponer que N = 2^k.

T(N) = T(2^k) = 2 \* T(2^(k-1)) + a \* 2^k

= 2 \* ( 2 \* T(2^(k-2)) + a \* 2^(k-1)) + a \* 2^k

= 2^2 \* T(2^(k-2)) + a \* 2^k +a \* 2^k

.

.

.

= 2^i \* T(2^(k-i))+ i \* a \* 2^k

.

.

.

= 2^k \* T(1) + k \* a \* 2^k

= b \* 2^k + k \* a \* 2^k

Pero si N = 2^k entonces k = log2(N), y por lo tanto hemos demostrado que:

T(N) = b \* N + a \* N \* log2(N).

Como lo que nos interesa es aproximar el valor, diremos (despreciando el término de menor orden) que

T(N) ~ N \* log2(N)

Dado que log2(N) es un número mucho más pequeño que N, hemos mostrado entonces que el merge sort se porta mucho mejor (es decir, es más eficiente) que los tres métodos de ordenamiento que discutimos en la sección anterior (que eran cuadráticos).

Si analizamos el espacio que consume, vemos que a cada paso la función merge genera una nueva lista cuya longitud es la suma de los tamaños de las dos listas, por lo que merge\_sort usa el doble de espacio que la lista de entrada.

**Resumen**

* Los métodos de ordenamiento de selección, inserción y burbujeo presentados en la sección anterior son métodos conceptualmente sencillos pero costosos en cantidad de operaciones (intercambios y/o comparaciones). Sin embargo, es posible conseguir métodos más eficientes usando algoritmos recursivos.
* El algoritmo *merge sort* consiste en dividir la lista a ordenar hasta que tenga 1 ó 0 elementos y luego combinar la lista de forma ordenada. De esta manera se logra un tiempo proporcional a N \* log2(N).

**Ejercicios:**

**Ejercicio 12.6:**

Ordená la lista [6, 0, 3, 2, 5, 7, 4, 1] usando el método merge sort. Dibujá el árbol de recursión explicando las llamadas que se hacen en cada paso, y el orden en el que se realizan, como mostramos más arriba para la lista [3, 1, 0, 4, 2].

**Ejercicio 12.7:**

Modificá la función merge\_sort para que también devuelva la cantidad de comparaciones hechas. Rehacé el último ejercicio de la sección anterior ([Ejercicio 12.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/02_Ordenamiento_sencillo.md#ejercicio-125-comparar-m%C3%A9todos-gr%C3%A1ficamente)) incorporando el merge sort a la comparación y al gráfico. Describí con tus palabras qué observas.

Guardá el archivo comparaciones\_ordenamiento.py con estas modificaciones, para entregarlo.

**El módulo timeit**

Hay casos en que contar la cantidad de operaciones que un algoritmo realiza se vuelve muy engorroso. Una alternativa simple aunque menos exacta es medir su tiempo de ejecución para problemas de distintos tamaños y estimar el orden del algoritmo a partir del cambio en el tiempo de ejecución al cambiar el tamaño del problema.

Existe un [módulo llamado timeit](https://docs.python.org/3/library/timeit.html) que permite medir tiempos de ejecución de código Python.

El comando timeit() del modulo timeit devuelve, en segundos, el tiempo de ejecución total para el comando y número de repeticiones especificadas. El código a ejecutar y la cantidad de ejecuciones se pasan como parámetros. En esta sección, vamos a usarlo para comparar algoritmos de ordenamiento.

En el siguiente ejemplo se le pide a Python una demora de 1 segundo (sleep(1)) y se le pide a timeit() que devuelva el tiempo de ejecución de ese comando, ejecutado una sola vez). Probálo varias veces:

In [1]: import time

In [2]: import timeit as tt

In [3]: tt.timeit('time.sleep(1)',number = 1)

Out[3]: 1.0010360410087742

Notarás que el tiempo de ejecución está muy cerca del esperado (1 segundo), pero no es exactamente ese valor y además hay cierta variacion entre repeticiones.

Ahora evaluemos la siguiente expresión, que concatena en un string los primeros 100 números enteros. Ejecutala por lo menos diez veces y mirá como varía la salida:

In [4]: tt.timeit('"-".join(str(n) for n in range(100))', number = 1)

Out[4]: 6.670000296551734e-05

Si medimos un proceso que tarda poco tiempo como en el último ejemplo, obtendremos resultados con una variabilidad de magnitud similar a la duración del proceso. Si queremos comparar duraciones relativas es mejor medir procesos largos o, si se trata de un proceso corto, repetirlo muchas veces. Usando timeit() podemos hacer esto cambiando el parametro number.

Probá lo siguiente:

In [5]: tt.timeit('"-".join(str(n) for n in range(100))', number = 10000)

Out[5]: 0.3018611848820001

Ahora comparemos la duración de ese código python con la duración de otras expresiones que dan el mismo resultado. Ejecutá cada una varias veces:

In [6]: tt.timeit('"-".join(str(n) for n in range(100))', number = 10000)

Out[6]: 0.3018611848820001

In [7]: tt.timeit('"-".join([str(n) for n in range(100)])', number = 10000)

Out[7]: 0.2727368790656328

In [8]: tt.timeit('"-".join(map(str, range(100)))', number = 10000)

Out[8]: 0.23702679807320237

**Ejemplo: evaluar el método de selección con timeit.**

Queremos evaluar cuánto tarda del método de selección dependiendo de la longitud de la lista de entrada.

Para eso, primero generamos listas de longitudes entre 1 y 256.

listas = []

for N in range(1, 256):

listas.append(generar\_lista(N))

Luego, definimos una función experimento\_timeit\_seleccion(listas, num) que realiza un experimento usando timeit para evaluar el método de selección (repitiendo num veces) con las listas pasadas como entrada, y devuelve los tiempos de ejecución para cada lista en un vector.

def experimento\_timeit\_seleccion(listas, num):

"""

Realiza un experimento usando timeit para evaluar el método

de selección para ordenamiento de listas

con las listas pasadas como entrada

y devuelve los tiempos de ejecución para cada lista

en un vector.

El parámetro 'listas' debe ser una lista de listas.

El parámetro 'num' indica la cantidad de repeticiones a ejecutar el método para cada lista.

"""

tiempos\_seleccion = []

global lista

for lista in listas:

# evalúo el método de selección

# en una copia nueva para cada iteración

tiempo\_seleccion = tt.timeit('ord\_seleccion(lista.copy())', number = num, globals = globals())

# guardo el resultado

tiempos\_seleccion.append(tiempo\_seleccion)

# paso los tiempos a arrays

tiempos\_seleccion = np.array(tiempos\_seleccion)

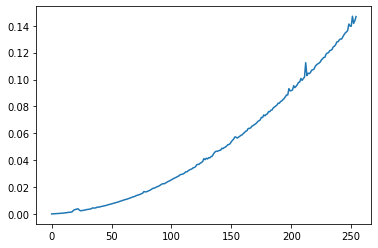
return tiempos\_seleccion

El parámetro globals = globals() permite a timeit() acceder a las variables y funciones definidas en el namespace global en que se está ejecutando. El comando global lista hace que la variable lista que va recorriendo la lista de listas, sea *global* y por lo tanto accesible a timeit().

Y ahora realizamos el experimento y lo graficamos.

tiempos\_seleccion = experimento\_timeit\_seleccion(listas, 100)

plt.plot(tiempos\_seleccion)

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/experimento_timeit_seleccion.jpg)

**Ejercicio 12.8:**

La idea de este ejercicio es, nuevamente, comparar los algoritmos de ordenamiento que vimos hasta ahora pero usando timeit() en lugar de contando a mano la cantidad de operaciones.

* Juntá en el archivo time\_ordenamiento.py los métodos de búsqueda del [Ejercicio 12.7](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/03_Divide_and_Conquer.md#ejercicio-127).
* Antes de empezar el experimento, eliminá de las funciones a medir todo código no esencial, en particular los prints para debug. Consumen tiempo y no son parte del algoritmo. También eliminá las cuentas de comparaciones, que ahora no son necesarias.
* Escribí un experimento que, tal como hiciste en el [Ejercicio 12.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/02_Ordenamiento_sencillo.md#ejercicio-125-comparar-m%C3%A9todos-gr%C3%A1ficamente), para N entre 1 y 256 genere una lista de largo N con números enteros del 1 al 1000, calcule el tiempo que tarda cada método en ordenar la lista y guarde estos resultados en vectores de largo 256.
* En este caso, vas a tener que generar y guardar todas las listas a ser utilizadas antes de correr el experimento, para poder usar las mismas para evaluar cada método. Definí para eso una función generar\_listas(Nmax) que genere una lista de listas, una de cada longitud entre 1 y Nmax, con valores aleatorios entre 1 y 1000.
* Asegurate de evaluar todos los métodos de ordenamiento con las mismas listas (siempre usá copias para no reordenar listas ya ordenadas) y guardar esta información para poder mostrarla o usarla.
* Graficá los datos de tiempos de ejecución en función de longitudes de la lista. ¿Coinciden las curvas con lo que habías predicho estimando el número de operaciones?
* Guardá el archivo time\_ordenamiento.py para entregarlo.

**Ejercicio 12.9:**

Opcional: Escribí una función merge3sort que funcione igual que el merge sort pero en lugar de dividir la lista de entrada en dos partes, la divida en tres partes. Deberás escribir la función merge3sort(lista1, lista2, lista3).

Probá tu función en las siguientes listas:

unalista = [1, 4, 3, 1, 7, 5]

otralista = [7, 6, 5, 4, 3, 2, 1, 0]

¿Cómo te parece que se va a comportar este método con respecto al merge sort original? Agregá este nuevo método a la comparación del ejercicio anterior.

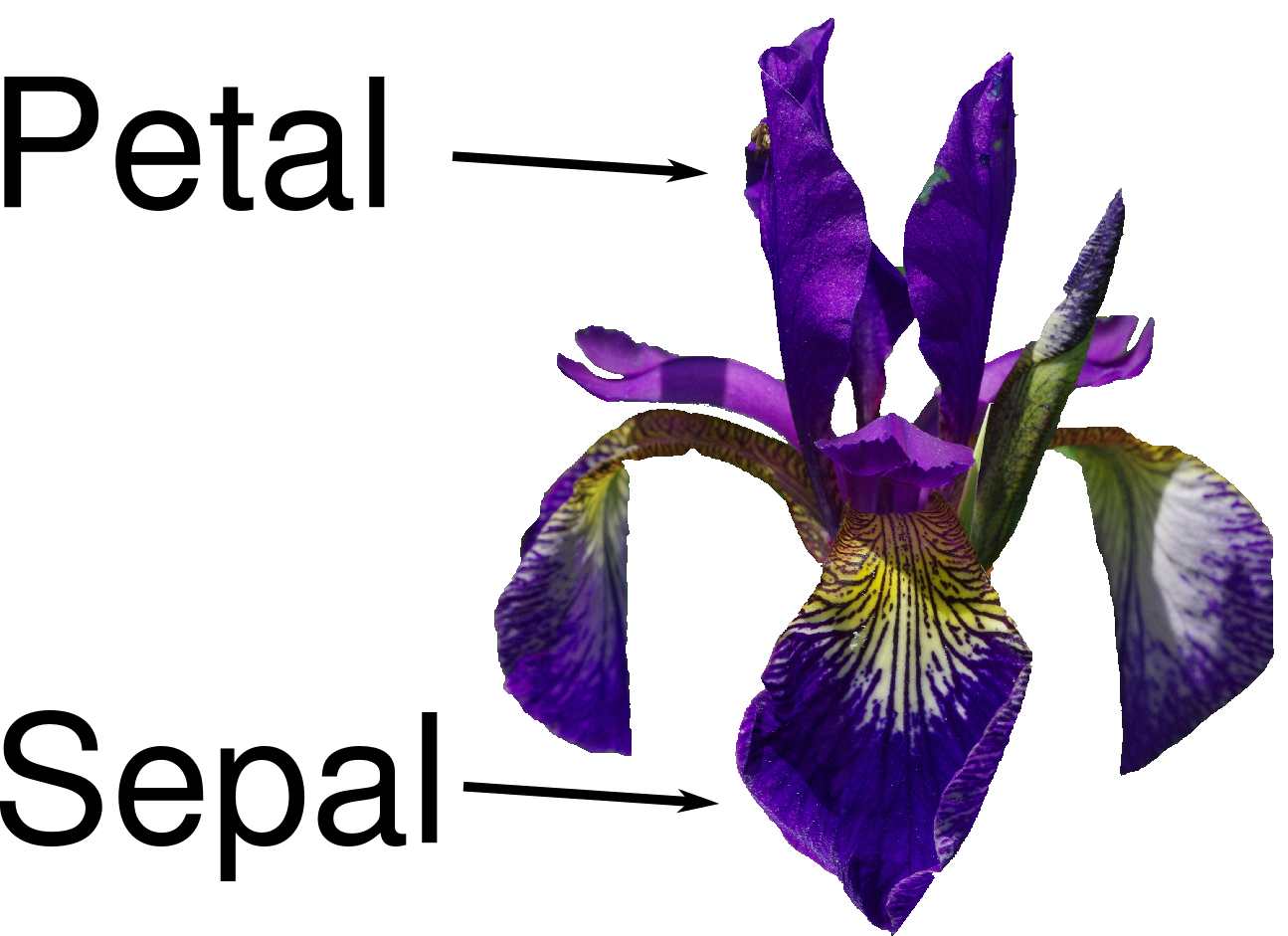
**12.4 Algoritmos de clasificación supervisada**

Para esta sección tenemos este [video introductorio](https://youtu.be/ygxoVQNRqs8).

En esta sección veremos un algoritmo de clasificación. Un problema de clasificación es un problema en el que tenemos algunas clases fijas (en nuestro ejemplo serán tres tipos de flores) y algunos atributos (medidas de los pétalos y sépalos, en nuestro ejemplo) a partir de los cuales queremos *inferir* la clase. Típicamente el algoritmo de clasificación se *entrena* con alguna parte de los datos para que *aprenda* y luego se *evalúa* cuán bien aprendió con el resto de los datos. Para esto hace falta tener un conjunto de datos *etiquetados* (es decir, con la clase bien definida). Luego, si funciona bien, el algoritmo podrá usarse para etiquetar nuevos datos de los que no se conoce la clase.

En esta sección nos concentraremos en el entrenamiento y la evaluación de los algoritmos.

Trabajaremos con la librería sklearn de python que está diseñada para realizar tareas de aprendizaje automático. La misma trae algunos conjuntos de datos de ejemplo. Trabajaremos con el clásico ejemplo de **Clasificación de Especies de flores Iris** según medidas del pétalo y el sépalo.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/iris_petal_sepal.png)

**Veamos los datos**

from sklearn.datasets import load\_iris

iris\_dataset = load\_iris()

Este dataset trae una serie de datos medidos de los pétalos y sépalos de 150 flores Iris y su clasificación en tres especies (setosa, versicolor y virginica). La idea es usar algunos de los datos de flores para entrenar un algoritmo y si podemos deducir la especie de las otras flores (no clasificadas) usando solo sus medidas.

El dataset es un diccionario con diferentes datos. Esencialmente en "data" tiene un array con las medidas de ancho y largo de pétalo y sépalo (atributos, o "features" en inglés) de 150 flores y en "target" tiene un numero (0, 1 ó 2) que representa la especie de estas flores. Veamos un poco la estructura de estos datos. El diccionario tiene las siguientes claves:

>>> print("Claves del diccionario iris\_dataset:\n", iris\_dataset.keys())

Claves del diccionario iris\_dataset:

dict\_keys(['data', 'target', 'frame', 'target\_names', 'DESCR', 'feature\_names', 'filename'])

Las flores se clasifican en tres:

>>> print("Target names:", iris\_dataset['target\_names'])

Target names: ['setosa' 'versicolor' 'virginica']

Y los atributos son cuatro por cada flor:

>>> print("Feature names:\n", iris\_dataset['feature\_names'])

Feature names:

['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

Son 150 flores etiquetadas, con cuatro atributos cada una, en un array de numpy. Las etiquetas son 0, 1 y 2 y se guardan también en un array:

>>> print("Type of data:", type(iris\_dataset['data']))

Type of data: <class 'numpy.ndarray'>

>>> print("Shape of data:", iris\_dataset['data'].shape)

Shape of data: (150, 4)

>>> print("First five rows of data:\n", iris\_dataset['data'][:5])

First five rows of data:

[[5.1 3.5 1.4 0.2]

[4.9 3. 1.4 0.2]

[4.7 3.2 1.3 0.2]

[4.6 3.1 1.5 0.2]

[5. 3.6 1.4 0.2]]

>>> print("Type of target:", type(iris\_dataset['target']))

Type of target: <class 'numpy.ndarray'>

>>> print("Shape of target:", iris\_dataset['target'].shape)

Shape of target: (150,)

>>> print("Target:\n", iris\_dataset['target'])

Target:

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2]

**Visualización de los datos**

Hagamos primero unos gráficos exploratorios para ver los datos y entender las correlaciones entre los atributos, usando un color diferente para cada especie de flor.

import pandas as pd

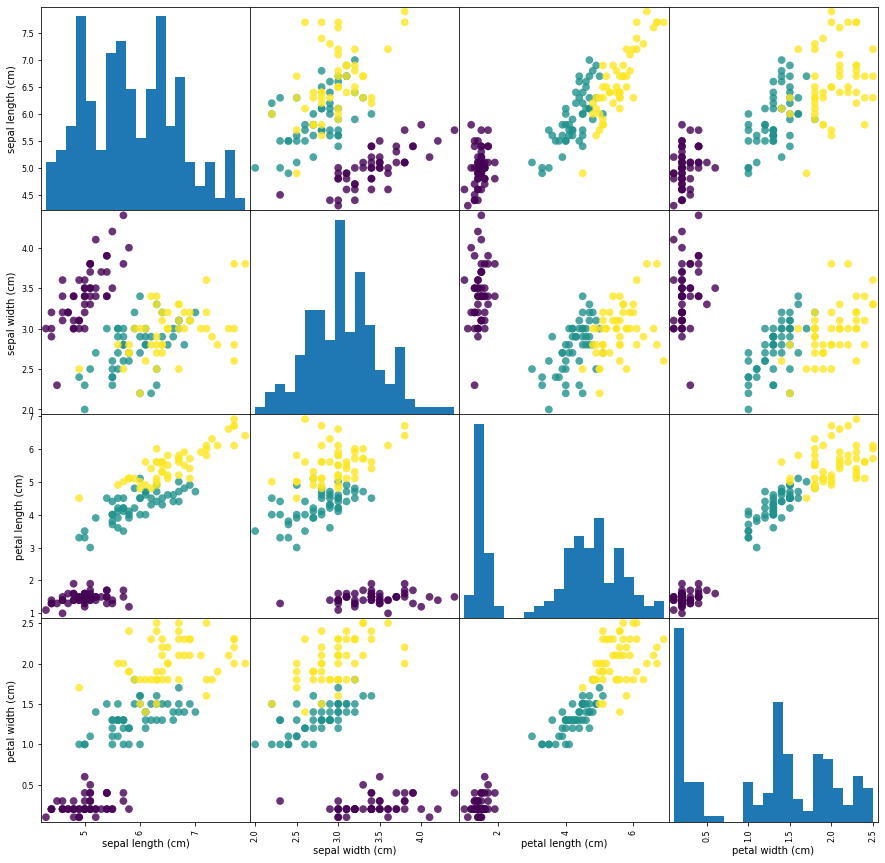
# creamos un dataframe de los datos de flores

# etiquetamos las columnas usando las cadenas de iris\_dataset.feature\_names

iris\_dataframe = pd.DataFrame(iris\_dataset['data'], columns = iris\_dataset.feature\_names)

# y hacemos una matriz de gráficos de dispersión, asignando colores según la especie

pd.plotting.scatter\_matrix(iris\_dataframe, c = iris\_dataset['target'], figsize = (15, 15), marker = 'o', hist\_kwds = {'bins': 20}, s = 60, alpha = 0.8)

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/output_27_1.png)

Notamos que una de las especies se distingue más fácilmente de las otras dos, mientras que las otras presentan cierta superposición.

**Ejercicio 12.10: Seaborn**

Repetí el gráfico anterior pero usando seaborn en lugar de pandas para graficar, y guardá el código correspondiente en un archivo iris\_seaborn.py para entregarlo.

*Sugerencia:* Usando iris\_dataframe['target'] = iris\_dataset['target'], agregá al DataFrame el atributo target de cada flor para poder hacer un sns.pairplot() seteando hue sobre las especies de iris.

**Training y testing**

Como dijimos antes, vamos a entrenar un algoritmo y luego a evaluar su capacidad de clasificar. Para evitar sesgos y sobreajustes tenemos que partir al conjunto de datos en dos:

* una parte de los datos (training) será de entrenamiento del algoritmo y
* otra parte (testing) será usada para la evaluación.

La librería sklearn trae funciones que hacen esta separación (split) de forma aleatoria, como se ve a continuación (en este caso fijamos una semilla con random\_state = 0, luego la sacaremos). Obviamente separamos tanto los atributos (features) como su clase (target). En este caso usaremos el 75% de los datos para entrenar y el 25% restante para evaluar.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

iris\_dataset['data'], iris\_dataset['target'], random\_state = 0)

>>> print("X\_train shape:", X\_train.shape)

>>> print("y\_train shape:", y\_train.shape)

X\_train shape: (112, 4)

y\_train shape: (112,)

>>> print("X\_test shape:", X\_test.shape)

>>> print("y\_test shape:", y\_test.shape)

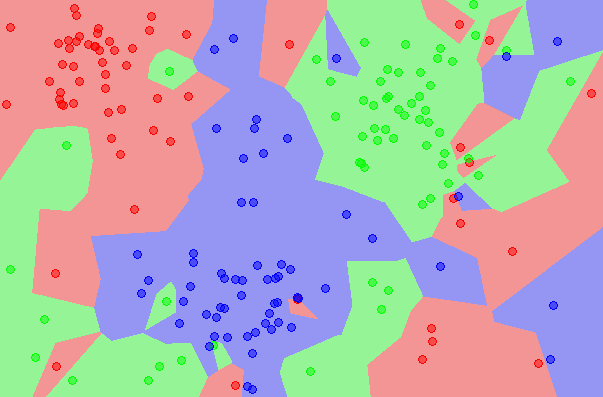
X\_test shape: (38, 4)

y\_test shape: (38,)

**Modelar**

Ahora vamos a construir nuestro primer modelo. Usaremos un algoritmo sencillo que se llama de "vecinos más cercanos" (K-nearest neighbors\_ en inglés, ver [wikipedia](https://es.wikipedia.org/wiki/K_vecinos_m%C3%A1s_pr%C3%B3ximos)). Lo entrenaremos con los datos de entrenamiento y al consultarle por un nuevo dato (de los de testing) lo que hará el algoritmo es buscar al dato de entrenamiento más cercano en el espacio de atributos y asignarle al nuevo dato la especie de esa flor. En otras palabras: cuando le preguntemos por la especie de una flor nueva va a contestarnos con la especie de la flor "más cercana" en el espacio de atributos (ancho y largo del pétalo y el sépalo).

De esta forma el espacio de atributos queda dividido en regiones a las que se asignará cada especie. En el siguiente gráfico puede verse una partición de un espacio de dos atributos y tres clases considerando un vecino más cercano (k=1) y entrenado con los datos del gráfico:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/Map1NN.png)

A un nuevo punto en este plano el clasificador así entrenado le asignará la clase correspondiente al color de fondo, que coincide con la clase del vecino más cercano.

Creamos una instancia de la clase KNeighborsClassifier

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n\_neighbors = 1)

Y la entrenamos con los datos de entrenamiento

knn.fit(X\_train, y\_train)

Listo, tenemos el clasificador entrenado. Ahora lo podemos usar para predecir la clase de una nueva flor a partir de sus cuatro medidas:

>>> import numpy as np

>>> X\_new = np.array([[5, 2.9, 1, 0.2]])

>>> print("X\_new.shape:", X\_new.shape)

X\_new.shape: (1, 4)

Grafiquemos este nuevo punto en rojo y veamos su relación con los datos de entrenamiento en dos de los atributos.

import matplotlib.pyplot as plt

plt.scatter(X\_train[:, 1], X\_train[:, 3], c = y\_train)

plt.scatter(X\_new[:, 1], X\_new[:, 3], c = 'red')

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/output_36_1.png)

Acá se ve que el punto rojo esta cerca de la clase "setosa". Utilicemos ahora el algoritmo knn entrenado para clasificar el punto X\_new:

>>> prediction = knn.predict(X\_new)

>>> print("Predicción:", prediction)

>>> print("Nombre de la Especie Predicha:",

iris\_dataset['target\_names'][prediction])

Predicción: [0]

Nombre de la Especie Predicha: ['setosa']

**Evaluación del modelo**

Finalmente, usemos el 25% de los datos etiquetados que nos guardamos para evaluar cuán bien funciona nuestro clasificador.

>>> y\_pred = knn.predict(X\_test)

>>> print("Predicciones para el conjunto de Test:\n", y\_pred)

>>> print("Etiquetas originales de este conjunto:\n", y\_test)

Predicciones para el conjunto de Test:

[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0

2]

Etiquetas originales de este conjjuto:

[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0

1]

Se ve que coinciden todos salvo el último. Podemos medir el éxito calculando la fracción de clasificaciones bien hechas (calculamos el promedio de "1 si está bien, 0 si está mal"):

>>> print(y\_pred == y\_test)

>>> print("Test set score: {:.2f}".format(np.mean(y\_pred == y\_test)))

[ True True True True True True True True True True True True

True True True True True True True True True True True True

True True True True True True True True True True True True

True False]

Test set score: 0.97

O, directamente, usando el método score que ya viene en el clasificador:

>>> print("Test set score: {:.2f}".format(knn.score(X\_test, y\_test)))

Test set score: 0.97

**Pasando en limpio todo**

Lo que hicimos hasta ahora fue:

1) Separar los datos en dos conjuntos: train y test.

2) Sefinir un clasificador knn y entrenarlo con los datos de training.

3) Evaluar el clasificador con los datos de testing.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

iris\_dataset['data'], iris\_dataset['target'])

knn = KNeighborsClassifier(n\_neighbors = 1)

knn.fit(X\_train, y\_train)

print("Test set score: {:.2f}".format(knn.score(X\_test, y\_test)))

Observá que en este último fragmento de código el split en test y train es aleatorio, y va a dar resultados (scores) diferentes cada vez que lo corramos.

**Ejercicios:**

**Ejercicio 12.11:**

Leé sobre los [clasificadores basados en arboles de decisión](https://es.wikipedia.org/wiki/Aprendizaje_basado_en_%C3%A1rboles_de_decisi%C3%B3n) y luego usá el objeto clasificador clf (definido a continuación) como se usó knn en el ejemplo anterior (es decir, entrená el clasificador sobre el conjunto train y evaluálo sobre el conjunto test). Tanto knn como clf son clasificadores y heredan los métodos "fit", "predict" y "score" de forma que su uso es casi idéntico. Ventajas del polimorfismo, del que hablamos antes (ver [Ejercicio 9.7](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/03_Herencia.md#ejercicio-97-polimorfismo-en-acci%C3%B3n)). ¿Qué clasificador dió mejores resultados?

from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()

**Ejercicio 12.12:**

La comparación anterior de los dos clasificadores puede resultar injusta ya que está basada en *una* partición del conjunto de datos en test y train que podría darle ventaja a uno u otro clasificador, arbitrariamente.

Para evitar esto, repetí 100 veces lo siguiente y calculá el promedio de los scores:

a) Partición del conjunto original en test y train aleatoriamente (sin fijar la semilla).

b) Entrenamiento de ambos modelos (knn y clf) con el conjunto train resultante.

c) Evaluación de ambos clasifcadores (score) con el conjunto test resultante.

¿Te animás a agregar también un clasificador de *[Random Forest](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html)*?

Imprimí el promedio de los scores obtenidos y guardá el código en el archivo clasificadores.py para entregar.

**12.5 Cierre de la clase de Ordenamiento**

En la clase de hoy te pedimos que entregues los siguientes archivos:

* El archivo burbujeo.py del [Ejercicio 12.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/02_Ordenamiento_sencillo.md#ejercicio-122-burbujeo).
* El archivo comparaciones\_ordenamiento.py del [Ejercicio 12.7](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/03_Divide_and_Conquer.md#ejercicio-127).
* El archivo time\_ordenamiento.py del [Ejercicio 12.8](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/03_Divide_and_Conquer.md#ejercicio-128).
* El archivo iris\_seaborn.py del [Ejercicio 12.10](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/04_introduccion_al_AA.md#ejercicio-1210-seaborn).
* Opcionalmente, el archivo clasificadores.py del [Ejercicio 12.12](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/12_Ordenamiento/04_introduccion_al_AA.md#ejercicio-1212).

Como de costumbre, completá por favor [el formulario](https://docs.google.com/forms/d/1s6zFYwZxgGih7auaLAtdanbzlNIxu14S0G5sGx2jVfg) asociado a la clase y adjuntá los archivos correspondientes.

Gracias!