Midterm 1, Summer 2021: A New Hope

Version 1.0.1: Added more reference solutions (master copy; no changes in the exam text or code).

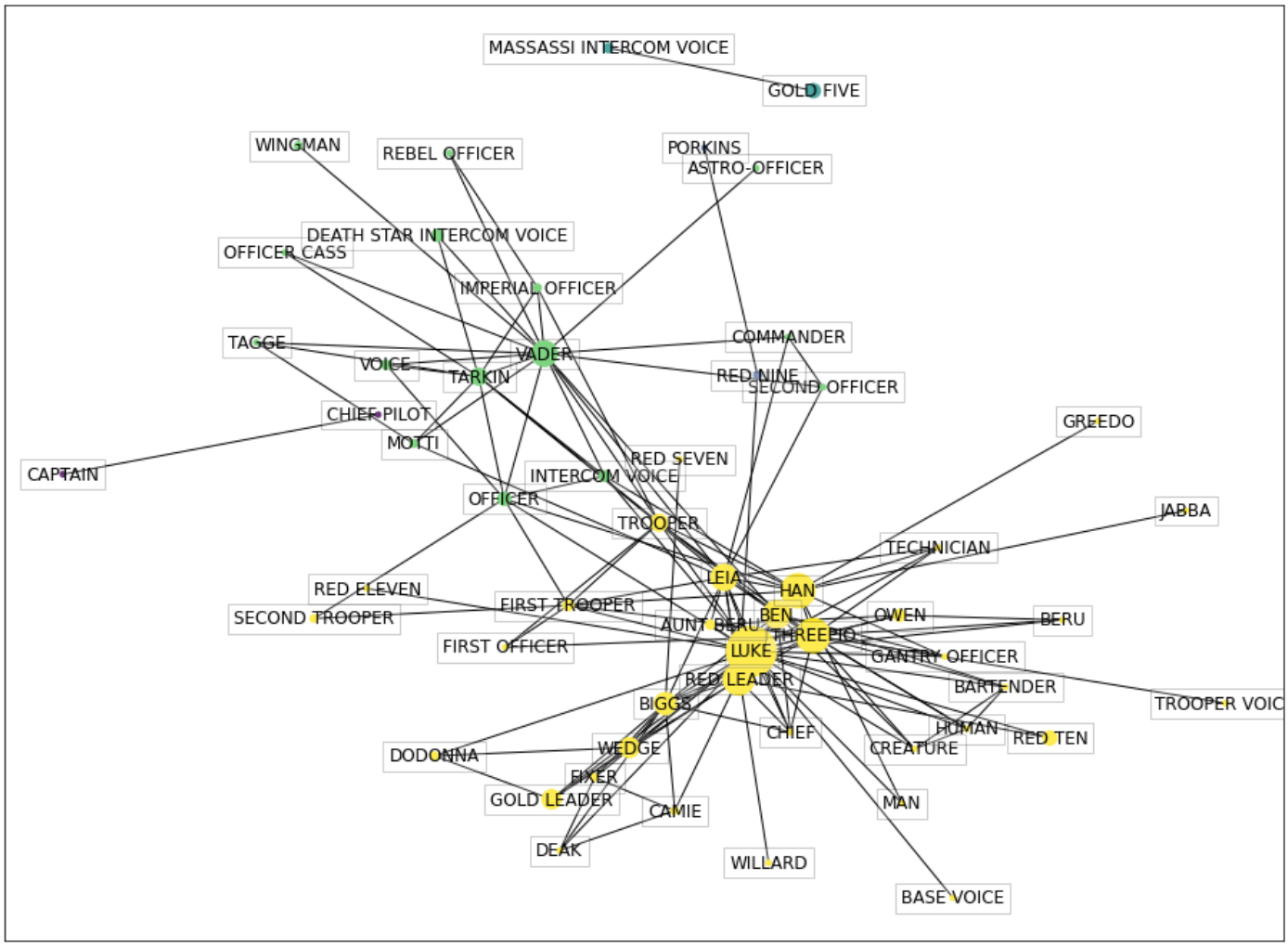
For other preliminaries and pointers, refer back to the Midterm 1: Release notes and updates (post @165) (https://piazza.com/class/knt87nry42w65k?cid=165) on Piazza.

- **Topics covered**: This problem assesses your knowledge of basic Python, including nested data structures, algorithmic problem solving, and string processing.
- Exercises: **Eight (8)**, numbered 0-7
- Time limit: **3.5 hours** (210 minutes)
- Points: **All scores are capped at 15 points** (the *"100% threshold"*) with no extra credit. So, if you get 13 points, your final score will be 13/15 ~ 86% ("middle B" equivalent).

Good luck!

Overview: Graph clustering

In this problem, you will analyze the screenplay (movie script) for the film, Star Wars: A New Hope (https://imsdb.com/scripts/Star-Wars-A-New-Hope.html). We are providing a semi-structured plaintext version, and you will use your Python skills to extract information about the characters that appear in the film and visualize their relationships through an automated analysis known as *graph clustering* (https://www.csc2.ncsu.edu/faculty/nfsamato/practical-graph-mining-with-R/slides/pdf/Graph_Cluster_Analysis.pdf). From it, you will generate a picture like this one:



You are looking at a visual representation of a **graph** or **network**. The boxes and circles show **vertices** and represent characters in the movie. The line segments that connect a pair of vertices are **edges**, which indicate that two characters speak to each other during the film. The colors show the clusters that the algorithm discovers. If you have seen the movie before, then you'll recognize that the biggest groups (yellow and green) reflect the two main "sides" in the film's conflict (yellow = "good guys," green = "bad guys").

Your main tasks. We'll use off-the-shelf graph clustering and visualization code. But to use it, you'll need to take the raw screenplay and generate the right inputs.

Your high-level workflow in this problem is as follows:

1. **Load** the raw screenplay text and **understand** the screenplay format *(brains only, no coding!)*
2. Break up the raw text into **scenes**, including a few cleaning steps
3. **Construct a graph** (social network) of which characters appear together in which scenes
4. Run a **graph clustering** to discover a "grouping" of the characters, and visualize the results

To get started, please run the following code cell to load some tools that we'll need later.

In [1]:
```
### BEGIN HIDDEN TESTS
%load_ext autoreload
%autoreload 2

global_overwrite = False
### END HIDDEN TESTS

from testing_tools import get_mem_usage_str, pretty_print
from testing_tools import load_raw_screenplay, pretty_print_element_apply

get_mem_usage_str()
```

```
Opening pickle from './resource/asnlib/publicdata/scene_starts.pickle' ...
Opening pickle from './resource/asnlib/publicdata/scene_start_diffs.pickle'
...
Opening pickle from './resource/asnlib/publicdata/scene_elements.pickle' ...
Opening pickle from './resource/asnlib/publicdata/scenes.pickle' ...
Opening pickle from './resource/asnlib/publicdata/characters_by_scene.pickle'
...
Opening pickle from './resource/asnlib/publicdata/character_co_occurrences.pi
ckle' ...
Opening pickle from './resource/asnlib/publicdata/largest_communities.pickle'
...
Opening pickle from './resource/asnlib/publicdata/settings_tin.pickle' ...
Opening pickle from './resource/asnlib/publicdata/spoken_lines_tin.pickle'
...
```

Out[1]: '47.4 MiB'

Part A: Understand the screenplay dataset (Exercise 0 — no coding!)

The first step in any data analysis task is simply to understand what you've got.

Exercise 0 (3 points, but no coding!)

Carefully read this section, which will explain what the data is. Be sure to run all its code cells, taking note of any functions we provide, which may help you later.

> The test cell at the end of Part A just prints "Passed!" and gives you 3 **free** points. However, you **must** submit the notebook to get these points.

The screenplay is stored as a plaintext file. The raw form we provide you initially is nothing more than a Python list of strings. Run the cell below, which loads that list into a variable named `raw_screenplay`, and displays the first 100 elements of the list.

In [2]:
```
raw_screenplay = load_raw_screenplay()

print("\nHere are the first 100 lines (list elements) of the screenplay (`raw_
screenplay`):\n")
pretty_print(raw_screenplay[:100])
```

Here are the first 100 lines (list elements) of the screenplay (`raw_screenplay`):

```
['STAR WARS',
 '',
 'Episode IV',
 '',
 'A NEW HOPE',
 '',
 'From the',
 'JOURNAL OF THE WHILLS',
 '',
 'by',
 'George Lucas',
 '',
 'Revised Fourth Draft',
 'January 15, 1976',
 '',
 'LUCASFILM LTD.',
 '',
 '',
 '',
 'A long time ago, in a galaxy far, far, away...',
 '',
 'A vast sea of stars serves as the backdrop for the main title.',
 'War drums echo through the heavens as a rollup slowly crawls',
 'into infinity.',
 '',
 'It is a period of civil war. Rebel spaceships,',
 'striking from a hidden base, have won their first',
 'victory against the evil Galactic Empire.',
 '',
 'During the battle, Rebel spies managed to steal',
 "secret plans to the Empire's ultimate weapon, the",
 'Death Star, an armored space station with enough',
 'power to destroy an entire planet.',
 '',
 "Pursued by the Empire's sinister agents, Princess",
 'Leia races home aboard her starship, custodian of',
 'the stolen plans that can save her people and',
 'restore freedom to the galaxy...',
 '',
 'The awesome yellow planet of Tatooine emerges from a total',
 'eclipse, her two moons glowing against the darkness. A tiny',
 'silver spacecraft, a Rebel Blockade Runner firing lasers',
 'from the back of the ship, races through space. It is pursed',
 'by a giant Imperial Stardestroyer. Hundreds of deadly',
 'laserbolts streak from the Imperial Stardestroyer, causing',
 'the main solar fin of the Rebel craft to disintegrate.',
 '',
 'INT. REBEL BLOCKADE RUNNER - MAIN PASSAGEWAY',
 '',
 'An explosion rocks the ship as two robots, Artoo-Detoo (R2-',
 'D2) and See-Threepio (C-3PO) struggle to make their way',
 'through the shaking, bouncing passageway. Both robots are',
```

```
 'old and battered. Artoo is a short, claw-armed tripod. His',
 'face is a mass of computer lights surrounding a radar eye.',
 'Threepio, on the other hand, is a tall, slender robot of',
 'human proportions. He has a gleaming bronze-like metallic',
 'surface of an Art Deco design.',
 '',
 'Another blast shakes them as they struggle along their way.',
 '',
 'THREEPIO',
 "Did you hear that? They've shut down",
 "the main reactor. We'll be destroyed",
 'for sure. This is madness!',
 '',
 'Rebel troopers rush past the robots and take up positions in',
 'the main passageway. They aim their weapons toward the door.',
 '',
 'THREEPIO',
 "We're doomed!",
 '',
 'The little R2 unit makes a series of electronic sounds that',
 'only another robot could understand.',
 '',
 'THREEPIO',
 "There'll be no escape for the Princess",
 'this time.',
 '',
 'Artoo continues making beeping sounds. Tension mounts as',
 'loud metallic latches clank and the scream of heavy equipment',
 'are heard moving around the outside hull of the ship.',
 '',
 'THREEPIO',
 "What's that?",
 '',
 'EXT. SPACECRAFT IN SPACE',
 '',
 'The Imperial craft has easily overtaken the Rebel Blockade',
 'Runner. The smaller Rebel ship is being drawn into the',
 'underside dock of the giant Imperial starship.',
 '',
 'INT. REBEL BLOCKADE RUNNER',
 '',
 'The nervous Rebel troopers aim their weapons. Suddenly a',
 'tremendous blast opens up a hole in the main passageway and',
 'a score of fearsome armored spacesuited stormtroopers make',
 'their way into the smoke-filled corridor.',
 '',
 'In a few minutes the entire passageway is ablaze with',
 'laserfire. The deadly bolts ricochet in wild random patterns']
```

Observe that each element of this list is a string. You may assume that no string has leading and trailing whitespace, as you can see above. Furthermore, some elements are blank (`''`), which will be important later. In fact, here is a function that can check if an element is a blank.

In [3]:
```
def is_blank(element):
    return element == ''

# Demo:
pretty_print_element_apply(raw_screenplay[0:3], is_blank)
```

```
Element 0:   is_blank('STAR WARS') == False
Element 1:   is_blank('') == True
Element 2:   is_blank('Episode IV') == False
```

Also observe that some elements consist of text in all caps. Here is a handy function to detect a *non-blank*, *all-caps* line:

In [4]:
```
def is_caps(element):
    return not is_blank(element) and element == element.upper()

pretty_print_element_apply(raw_screenplay[0:3], is_caps)
```

```
Element 0:   is_caps('STAR WARS') == True
Element 1:   is_caps('') == False
Element 2:   is_caps('Episode IV') == False
```

Overall structure: scenes, lines, and notes

The screenplay is a sequence of **scenes**, each of which contains some mixture of **lines of spoken dialogue** and **scene notes.** These components are explained below.

Scenes. Here is a fragment that shows typical examples of scenes. (Each line is a printed string from `raw_screenplay` with its index shown to the left, e.g., `raw_screenplay[6895] == 'GOLD FIVE'` .)

```
6893:    INT. MASSASSI OUTPOST - WAR ROOM
6894:
6895:    GOLD FIVE
6896:    (over speaker)
6897:    I'd say about twenty guns. Some on
6898:    the surface, some on the towers.
6899:
6900:    Leia, Threepio, and the technicians view the projected target
6901:    screen, as red and blue target lights glow. The red target
6902:    near the center blinks on and off.
6903:
6904:    MASSASSI INTERCOM VOICE
6905:    (over speaker)
6906:    Death Star will be in range in five
6907:    minutes.
6908:
6909:    EXT. SURFACE OF THE DEATH STAR
6910:
6911:    The three Y-wing fighters race toward camera and zoom overhead
6912:    through a hail of laserfire.
6913:
6914:    INT. GOLD LEADER'S Y-WING - COCKPIT
6915:
6916:    Gold Leader pulls his computer targeting device down in front
         ... (and so on) ...
```

Scene settings (or just **settings**): You can tell when a scene starts and ends by looking for *scene settings*. The setting is a description of where the scene takes place. It is *always* written in all caps and starts with either `'INT.'` (for "interior") or `'EXT.'` ("exterior"). Here, the three settings are `'INT. MASSASSI OUTPOST - WAR ROOM'` , `'EXT. SURFACE OF THE DEATH STAR'` , and `"INT. GOLD LEADER'S Y-WING - COCKPIT"` .

There is one tricky thing about settings: sometimes they may span multiple lines. For example, in the following scene,

```
409:    EXT. TATOOINE - ANCHORHEAD - SETTLEMENT - POWER STATION -
410:    DAY
411:
412:    The group stumbles out into the stifling desert sun. Camie
413:    and The Fixer complain and are forced to shade their eyes.
414:    Luke has his binoculars out scanning the heavens.
415:
```

the setting is the two-element slice, `raw_screenplay[409:411]` . But when that happens, the setting will consist of consecutive non-blank lines, again in all caps.

Scene bodies: The body of a scene is the sequence of all the elements that start at a setting and go until the next setting. So the scene set in `'INT. MASSASSI OUTPOST - WAR ROOM'` above starts at element 6893 and ends at 6908. Per the Python slicing convention, the slice corresponding to the first scene is `raw_screenplay[6893:6909]` , i.e., the stopping value is 6909 = 6908 + 1.

Lines of spoken dialogue (or just **lines**): A scene *may* include *lines of spoken dialogue*, or just *lines*, which is what the **characters** of a scene speak.

For example, in the scene set in `'INT. MASSASSI OUTPOST - WAR ROOM'`, there are two characters, one named `'GOLD FIVE'` and another named `'MASSASSI INTERCOM VOICE'`. Character names are *always* written in all caps.

The character's *lines* are all the elements immediately after their name until the first blank line. For example, the lines spoken by `'GOLD FIVE'` start at 6896 (`'(over speaker)'`) and end at 6898 (`'the surface, some on the towers.'`). Therefore, we say GOLD FIVE's lines in this case are the slice, `raw_screenplay[6896:6899]` (again, the stopping value is the end plus one).

Scene notes (or just "notes"). Anything else that is neither a scene-start nor a line is a *note.*

For example, look between `'GOLD FIVE'` and `'MASSASSI INTERCOM VOICE'` above. There is a three-element note,

```
6900:    Leia, Threepio, and the technicians view the projected target
6901:    screen, as red and blue target lights glow. The red target
6902:    near the center blinks on and off.
```

And the scene set in `'EXT. SURFACE OF THE DEATH STAR'` has no lines, only the note,

```
6911:    The three Y-wing fighters race toward camera and zoom overhead
6912:    through a hail of laserfire.
```

A "freebie" test cell. With that lengthy set up, you should know everything you need to know to start processing the script, to convert it into a more structured (analysis-friendly) form. But first, go ahead and submit this notebook to get your free points for reading this far.

In [5]:
```
# Test cell: `mt1_ex0_freebie` (3 points)

print("\n(Passed!)")
```

```
(Passed!)
```

Part B: Scene extractor (Exercises 1-3)

For your first task, let's build some code to extract all elements of a scene. We'll break down this task into three steps (Exercises 1-3):

- Exercise 1 (1 point): Find the index of the starting element ("scene starts") of every scene.
- Exercise 2 (2 points): Determine the number of elements that belong to each scene.
- Exercise 3 (3 points): Create a new nested data structure that groups elements by scene.

Exercise 1: Find scene starts (`find_scene_starts` ; 1 point)

Suppose you are given a raw screenplay, `screenplay` . Complete the function,

```
def find_scene_starts(screenplay):
    ...
```

so that it returns a list of **indices** of where each scene begins.

For example, suppose `screenplay` is the following list:

```
screenplay = [
    'Han draws his laser pistol and pops off a couple of shots',
    'which force the stormtroopers to dive for safety. The',
    'pirateship engines whine as Han hits the release button that',
    'slams the overhead entry shut.',
    '',
    'INT. MILLENNIUM FALCON',
    '',
    'HAN',
    'Chewie, get us out of here!',
    '',
    'The group straps in for take off.',
    '',
    'THREEPIO',
    "Oh, my. I'd forgotten how much I",
    'hate space travel.',
    '',
    'EXT. TATOOINE - MOS EISLEY - STREETS',
    '',
    'The half-dozen stormtroopers at a check point hear the general',
    'alarm and look to the sky as the huge starship rises above',
    'the dingy slum dwellings and quickly disappears into the',
    'morning sky.',
    '',
    'INT. MILLENNIUM FALCON - COCKPIT',
    '',
    "Han climbs into the pilot's chair next to Chewbacca, who",
    'chatters away as he points to something on the radar scope.',
    ''
]
```

Then `find_scene_starts(screenplay)` should return the list `[5, 16, 23]` .

> *Note:* If `screenplay` has *no* scenes in it, your function should return an empty list.

In [6]:
```
def find_scene_starts(screenplay):
    ### BEGIN SOLUTION
    return find_scene_starts__v0(screenplay)

# Helper function
def is_scene_start(element):
    return is_caps(element) and element[:4] in ['INT.', 'EXT.']

# Method 0: Basic loop, the way you'd write it in many other languages
def find_scene_starts__v0(screenplay):
    starts = []
    for i in range(len(screenplay)):
        if is_scene_start(screenplay[i]):
            starts.append(i)
        i += 1
    return starts

# Method 1: List comprehension + `enumerate`
def find_scene_starts__v1(screenplay):
    return [i for i, e in enumerate(screenplay) if is_scene_start(e)]

    ### END SOLUTION
```

In [7]:
```
# Demo cell:
demo_screenplay = [
    'Han draws his laser pistol and pops off a couple of shots',
    'which force the stormtroopers to dive for safety. The',
    'pirateship engines whine as Han hits the release button that',
    'slams the overhead entry shut.',
    '',
    'INT. MILLENNIUM FALCON',
    '',
    'HAN',
    'Chewie, get us out of here!',
    '',
    'The group straps in for take off.',
    '',
    'THREEPIO',
    "Oh, my. I'd forgotten how much I",
    'hate space travel.',
    '',
    'EXT. TATOOINE - MOS EISLEY - STREETS',
    '',
    'The half-dozen stormtroopers at a check point hear the general',
    'alarm and look to the sky as the huge starship rises above',
    'the dingy slum dwellings and quickly disappears into the',
    'morning sky.',
    '',
    'INT. MILLENNIUM FALCON - COCKPIT',
    '',
    "Han climbs into the pilot's chair next to Chewbacca, who",
    'chatters away as he points to something on the radar scope.',
    ''
]
find_scene_starts(demo_screenplay)
```

Out[7]:
```
[5, 16, 23]
```

In [8]:
```
# Test cell: `mt1_ex1_find_scene_starts` (1 point)

### BEGIN HIDDEN TESTS
def mt1_ex1__gen_soln__(fn_base="scene_starts", fn_ext="pickle", overwrite=Fal
se):
    from testing_tools import file_exists, save_pickle
    fn = f"{fn_base}.{fn_ext}"
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping...")
    else: # not file_exists(fn) or overwrite
        print(f"'{fn}' does not exist or needs to be overwritten; generatin
g...")
        scene_starts = find_scene_starts(raw_screenplay)
        save_pickle(scene_starts, fn)

!date
mt1_ex1__gen_soln__(overwrite=False or global_overwrite)
!date
### END HIDDEN TESTS

from testing_tools import mt1_ex1__check
print("Testing...")
for trial in range(250):
    mt1_ex1__check(find_scene_starts)

find_scene_starts__passed = True
print(get_mem_usage_str())
print("\n(Passed!)")
```

```
Thu Jun 17 11:32:57 EDT 2021
'scene_starts.pickle' exists; skipping...
Thu Jun 17 11:32:57 EDT 2021
Testing...
49.2 MiB

(Passed!)
```

Sample results for Exercise 1, `find_scene_starts => scene_starts`

If you had a working solution to Exercise 1, then in principle you could use it to find the scene starts for all scenes in `raw_screenplay`. We have precomputed this result for you; run the cell below create a variable named `scene_starts`, which holds this result.

> Note: **Read and run this cell even if you skipped or otherwise did not complete Exercise 1.**

In [9]:
```
from testing_tools import mt1_scene_starts as scene_starts

print("\nSample results: First few scene starts:\n")
for i, i_start in enumerate(scene_starts[:3]):
    print(f"==> Scene {i} starts at element {i_start} with the setting, '{raw_
screenplay[i_start]}'")
```

```
Sample results: First few scene starts:

==> Scene 0 starts at element 47 with the setting, 'INT. REBEL BLOCKADE RUNNE
R - MAIN PASSAGEWAY'
==> Scene 1 starts at element 85 with the setting, 'EXT. SPACECRAFT IN SPACE'
==> Scene 2 starts at element 91 with the setting, 'INT. REBEL BLOCKADE RUNNE
R'
```

Exercise 2: Measuring consecutive differences (2 points)

Given an ordered list of scene starts, you can calculate the number of elements in each scene by looking at the *difference* between consecutive indices.

For example, suppose there are three scene starts, given in the list `starts = [47, 85, 91]`. Then you know the first scene consists of 85-47=38 elements and the second scene has 91-85=6 elements. For the last scene, you would need to know the total number of elements. Suppose that is `num_elems=115`. Then the last scene would have length 115-91=24 elements.

Complete the function,

```
def diffs(starts, num_elems):
    ...
```

so that it computes these *consecutive differences*, returning them in a list. Per the preceding example, `diffs([47, 85, 91], 115)` would return `[38, 6, 24]`.

There are two extra requirements.

1. In the function definition below, `num_elems` takes on a *default value* of `None`. Your solution should check whether `num_elems` is `None`; if so, then you should ignore the last start. That is, `diffs([47, 85, 91])` or `diffs([47, 85, 91], None)` should both return `[38, 6]`, since you don't have enough information to compute a difference for the last input.
2. If `starts` is an empty list, then you should return an empty list, *regardless* of the value of `num_elems`.

> *Note:* You may assume that `starts` is sorted.

In [10]:
```
def diffs(starts, num_elems=None):
    assert (num_elems is None) or (not starts) or (num_elems >= max(starts))
    ### BEGIN SOLUTION
    return diffs__v1(starts, num_elems)

# == Method 0: Basic loop (the style of other languages) ==
def diffs__v0(starts, num_elems=None):
    deltas = []
    for i in range(len(starts)-1):
        a, b = starts[i], starts[i+1]
        deltas.append(b - a)
    if starts and num_elems is not None:
        deltas.append(num_elems - starts[-1])
    return deltas

# == Method 1: A little tighter (more "Pythonic") ==
def zip_consecutive_pairs(x):
    """Given `x`, generates `(x[i], x[i+1])` pairs."""
    return zip(x[:-1], x[1:])

def calc_deltas(x):
    """Given `x`, returns `x[i+1] - x[i]` differences ("deltas")."""
    return [b - a for a, b in zip_consecutive_pairs(x)]

def diffs__v1(starts, num_elems=None):
    deltas = calc_deltas(starts)
    if starts and num_elems is not None:
        deltas.append(num_elems - starts[-1])
    return deltas
    ### END SOLUTION
```

In [11]:
```
# Demo cell:
print(diffs([47, 85, 91], 115))  # answer: [38, 6, 24]
print(diffs([47, 85, 91]))       # answer: [38, 6]
print(diffs([47, 85, 91], None)) # answer: [38, 6]
print(diffs([]))                 # answer: []
print(diffs([91], 115))          # answer: [24]
print(diffs([]), 115)            # answer: []
```

```
[38, 6, 24]
[38, 6]
[38, 6]
[]
[24]
[] 115
```

In [12]:
```
# Test cell: `mt1_ex2_diffs` (2 points)

### BEGIN HIDDEN TESTS
def mt1_ex2__gen_soln__(fn_base="scene_start_diffs", fn_ext="pickle", overwrit
e=False):
    from testing_tools import file_exists, save_pickle
    fn = f"{fn_base}.{fn_ext}"
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping...")
    else: # not file_exists(fn) or overwrite
        print(f"'{fn}' does not exist or needs to be overwritten; generatin
g...")
        scene_start_diffs = diffs(scene_starts, len(raw_screenplay))
        save_pickle(scene_start_diffs, fn)

!date
mt1_ex2__gen_soln__(overwrite=False or global_overwrite)
!date
### END HIDDEN TESTS

from testing_tools import mt1_ex2__check
print("Testing...")
for trial in range(250):
    mt1_ex2__check(diffs)

diffs__passed = True
print(get_mem_usage_str())
print("\n(Passed!)")
```

```
Thu Jun 17 11:32:57 EDT 2021
'scene_start_diffs.pickle' exists; skipping...
Thu Jun 17 11:32:58 EDT 2021
Testing...
49.3 MiB

(Passed!)
```

Sample results for Exercise 2, `diffs => scene_lengths`

If you had a working solution to Exercise 2, then in principle you could use it to find the number of elements in each scene for all scenes in `raw_screenplay`. We have precomputed this result for you; run the cell below create a variable named `scene_lengths`, which holds this result.

> Note: **Read and run this cell even if you skipped or otherwise did not complete Exercise 2.**

In [13]:
```
from testing_tools import mt1_scene_start_diffs as scene_lengths

print("\nSample results: First few scene starts and their lengths:\n")
for i, (i_start, i_length) in enumerate(zip(scene_starts[:3], scene_lengths[:
3])):
    print(f"==> Scene {i}, '{raw_screenplay[i_start]}': Starts at {i_start} wi
th {i_length} elements.")
```

```
Sample results: First few scene starts and their lengths:

==> Scene 0, 'INT. REBEL BLOCKADE RUNNER - MAIN PASSAGEWAY': Starts at 47 wit
h 38 elements.
==> Scene 1, 'EXT. SPACECRAFT IN SPACE': Starts at 85 with 6 elements.
==> Scene 2, 'INT. REBEL BLOCKADE RUNNER': Starts at 91 with 24 elements.
```

Exercise 3: Group elements by scene (3 points)

Suppose you are given

- a screenplay (`screenplay` , a list of elements);
- the index of the start of each scene (`starts`);
- and the length of each scene (`lengths`).

Complete the function,

```
def group_elements_by_scene(screenplay, starts, lengths):
    ...
```

so that it returns a Python list of Python dictionaries, where

- the outer list has one entry per scene;
- the dictionary has two key-value pairs:
  - `setting` , which holds the setting of the scene, and
  - `elements` , which holds all the elements that belong to that scene.

There are **two additional requirements**:

1. If there are any blank elements after the setting or at the end of a scene, these should be **removed.**
2. If a **setting spans multiple elements**, then **those elements should be joined into one string** with the elements separated by a single space character.

For example, recall the `demo_screenplay` from Exercise 1:

In [14]:
```
# Run me:
pretty_print(demo_screenplay)
```

```
['Han draws his laser pistol and pops off a couple of shots',
 'which force the stormtroopers to dive for safety. The',
 'pirateship engines whine as Han hits the release button that',
 'slams the overhead entry shut.',
 '',
 'INT. MILLENNIUM FALCON',
 '',
 'HAN',
 'Chewie, get us out of here!',
 '',
 'The group straps in for take off.',
 '',
 'THREEPIO',
 "Oh, my. I'd forgotten how much I",
 'hate space travel.',
 '',
 'EXT. TATOOINE - MOS EISLEY - STREETS',
 '',
 'The half-dozen stormtroopers at a check point hear the general',
 'alarm and look to the sky as the huge starship rises above',
 'the dingy slum dwellings and quickly disappears into the',
 'morning sky.',
 '',
 'INT. MILLENNIUM FALCON - COCKPIT',
 '',
 "Han climbs into the pilot's chair next to Chewbacca, who",
 'chatters away as he points to something on the radar scope.',
 '']
```

In this case, `starts = [5, 16, 23]` and `lengths = [11, 7, 5]`. Thus, `group_elements_by_scene(demo_screenplay, starts, lengths)` would return the following list of three dictionaries:

```
group_elements_by_scene(demo_screenplay, starts, lengths) == \
[ # Outer list
    # Scene 0:
    {'setting' : 'INT. MILLENNIUM FALCON',
     'elements': [
        # No leading blank elements
        'HAN',
        'Chewie, get us out of here!',
        '',
        'The group straps in for take off.',
        '',
        'THREEPIO',
        "Oh, my. I'd forgotten how much I",
        'hate space travel.'
         # No trailing blank elements
     ]},
    # Scene 1:
    {'setting' : 'EXT. TATOOINE - MOS EISLEY - STREETS',
     'elements': [
        'The half-dozen stormtroopers at a check point hear the general',
        'alarm and look to the sky as the huge starship rises above',
        'the dingy slum dwellings and quickly disappears into the',
        'morning sky.'
     ]},
    # Scene 2:
    {'setting' : 'INT. MILLENNIUM FALCON - COCKPIT',
     'elements': [
        "Han climbs into the pilot's chair next to Chewbacca, who",
        'chatters away as he points to something on the radar scope.'
     ]}
]
```

And in case of settings that span more than one element, e.g.,

In [15]:
```
# Run me:
demo_screenplay2 = [
    'EXT. TATOOINE - ANCHORHEAD - SETTLEMENT - POWER STATION -',
    'DAY',
    '',
    'The group stumbles out into the stifling desert sun. Camie',
    'and The Fixer complain and are forced to shade their eyes.',
    'Luke has his binoculars out scanning the heavens.'
]
```

your function would return:

```
group_elements_by_scene(demo_screenplay2, [0], [6]) == \
[{  # Multi-line setting: lines are combined:
    'setting' : 'EXT. TATOOINE - ANCHORHEAD - SETTLEMENT - POWER STATION - DAY',
    # Elements processed as before:
    'elements': ['The group stumbles out into the stifling desert sun. Camie',
                 'and The Fixer complain and are forced to shade their eyes.',
                 'Luke has his binoculars out scanning the heavens.']}]
```

Note 0: You may assume that `len(starts) == len(lengths)` .

Note 1: If `screenplay` or `starts` is empty, your function should return an empty list.

Note 2: You may assume that `starts` is sorted.

Note 2: You may assume that scenes do not overlap. That is, for the `i` -th scene, `starts[i]+lengths[i] <= starts[i+1]` .

In [16]:
```
def group_elements_by_scene(screenplay, starts, lengths):
    assert len(starts) == len(lengths)
    assert len(starts) <= len(screenplay) or not starts \
           or all([0 <= i <= i+m <= j for i, m, j in zip(starts[:-1], lengths
[:-1], starts[1:])])
    ### BEGIN SOLUTION
    return group_elements_by_scene__v0(screenplay, starts, lengths)

# === Method 0: In the style of other languages ===
def group_elements_by_scene__v0(screenplay, starts, lengths):
    scenes = []
    for i in range(len(starts)): # i == scene number or index
        # `k` iterates over elements of the scene
        k_start = starts[i]
        k_stop = k_start + lengths[i]

        # Build the setting
        setting = ''
        while k_start < k_stop and is_caps(screenplay[k_start]):
            setting += (' ' if setting else '') + screenplay[k_start]
            k_start += 1

        # Build the body
        while k_start < k_stop and is_blank(screenplay[k_start]): # skip leadi
ng blanks
            k_start += 1
        while k_stop > k_start and is_blank(screenplay[k_stop-1]): # remove tr
ailing blanks
            k_stop -= 1
        body = screenplay[k_start:k_stop]
        scenes.append({'setting': setting, 'elements': body})
    return scenes

# === Method 1: Auxiliary function + more Pythonic list operations ===
def grab_elements(screenplay, i, m):
    """Given scene elements from `screenplay[i:m]`, extract the setting and bo
dy."""
    setting = screenplay[i]
    elements = screenplay[i+1:i+m]
    while elements and not is_blank(elements[0]): # multi-line settings
        setting += ' ' + elements.pop(0)
    while elements and is_blank(elements[0]): # remove leading blanks
        elements.pop(0)
    while elements and is_blank(elements[-1]): # remove trailing blanks
        elements.pop()
    return {'setting': setting, 'elements': elements}

def group_elements_by_scene__v1(screenplay, starts, lengths):
    return [grab_elements(screenplay, i, m) for i, m in zip(starts, lengths)]
    ### END SOLUTION
```

In [17]:
```
# Demo cell 0:
group_elements_by_scene(demo_screenplay, [5, 16, 23], [11, 7, 5])
```

Out[17]:
```
[{'setting': 'INT. MILLENNIUM FALCON',
  'elements': ['HAN',
   'Chewie, get us out of here!',
   '',
   'The group straps in for take off.',
   '',
   'THREEPIO',
   "Oh, my. I'd forgotten how much I",
   'hate space travel.']},
 {'setting': 'EXT. TATOOINE - MOS EISLEY - STREETS',
  'elements': ['The half-dozen stormtroopers at a check point hear the genera
l',
   'alarm and look to the sky as the huge starship rises above',
   'the dingy slum dwellings and quickly disappears into the',
   'morning sky.']},
 {'setting': 'INT. MILLENNIUM FALCON - COCKPIT',
  'elements': ["Han climbs into the pilot's chair next to Chewbacca, who",
   'chatters away as he points to something on the radar scope.']}]
```

In [18]:
```
# Demo cell 1:
group_elements_by_scene(demo_screenplay2, [0], [6])
```

Out[18]:
```
[{'setting': 'EXT. TATOOINE - ANCHORHEAD - SETTLEMENT - POWER STATION - DAY',
  'elements': ['The group stumbles out into the stifling desert sun. Camie',
   'and The Fixer complain and are forced to shade their eyes.',
   'Luke has his binoculars out scanning the heavens.']}]
```

In [19]:
```
# Test cell: `mt1_ex3_group_elements_by_scene` (3 points)

### BEGIN HIDDEN TESTS
def mt1_ex3__gen_soln__(fn_base="scene_elements", fn_ext="pickle", overwrite=F
alse):
    from testing_tools import file_exists, save_pickle
    fn = f"{fn_base}.{fn_ext}"
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping...")
    else: # not file_exists(fn) or overwrite
        print(f"'{fn}' does not exist or needs to be overwritten; generatin
g...")
        scene_elements = group_elements_by_scene(raw_screenplay, scene_starts,
scene_lengths)
        save_pickle(scene_elements, fn)

!date
mt1_ex3__gen_soln__(overwrite=False or global_overwrite)
!date
### END HIDDEN TESTS

from testing_tools import mt1_ex3__check
print("Testing...")
for trial in range(250):
    mt1_ex3__check(group_elements_by_scene)

group_elements_by_scene__passed = True
print(get_mem_usage_str())
print("\n(Passed!)")
```

```
Thu Jun 17 11:32:58 EDT 2021
'scene_elements.pickle' exists; skipping...
Thu Jun 17 11:32:58 EDT 2021
Testing...
49.4 MiB

(Passed!)
```

Sample results for Exercise 3, `group_elements_by_scene` => `scene_elements`

If you had a working solution to Exercise 3, then in principle you could use it to group all the elements by scene for all scenes in `raw_screenplay`. We have precomputed this result for you; run the cell below create a variable named `scene_elements`, which holds this result.

> Note: **Read and run this cell even if you skipped or otherwise did not complete Exercise 3.**

In [20]:
```
from testing_tools import mt1_scene_elements as scene_elements

print("\nSample results:")
for i, scene in enumerate(scene_elements[:3]):
    print(f"\n>>> Scene {i} <<<")
    pretty_print(scene)
```

```
Sample results:

>>> Scene 0 <<<
{'setting': 'INT. REBEL BLOCKADE RUNNER - MAIN PASSAGEWAY',
 'elements': ['An explosion rocks the ship as two robots, Artoo-Detoo (R2-',
              'D2) and See-Threepio (C-3PO) struggle to make their way',
              'through the shaking, bouncing passageway. Both robots are',
              'old and battered. Artoo is a short, claw-armed tripod. His',
              'face is a mass of computer lights surrounding a radar eye.',
              'Threepio, on the other hand, is a tall, slender robot of',
              'human proportions. He has a gleaming bronze-like metallic',
              'surface of an Art Deco design.',
              '',
              'Another blast shakes them as they struggle along their way.',
              '',
              'THREEPIO',
              "Did you hear that? They've shut down",
              "the main reactor. We'll be destroyed",
              'for sure. This is madness!',
              '',
              'Rebel troopers rush past the robots and take up positions in',
              'the main passageway. They aim their weapons toward the door.',
              '',
              'THREEPIO',
              "We're doomed!",
              '',
              'The little R2 unit makes a series of electronic sounds that',
              'only another robot could understand.',
              '',
              'THREEPIO',
              "There'll be no escape for the Princess",
              'this time.',
              '',
              'Artoo continues making beeping sounds. Tension mounts as',
              'loud metallic latches clank and the scream of heavy equipmen
t',
              'are heard moving around the outside hull of the ship.',
              '',
              'THREEPIO',
              "What's that?"]}

>>> Scene 1 <<<
{'setting': 'EXT. SPACECRAFT IN SPACE',
 'elements': ['The Imperial craft has easily overtaken the Rebel Blockade',
              'Runner. The smaller Rebel ship is being drawn into the',
              'underside dock of the giant Imperial starship.']}

>>> Scene 2 <<<
{'setting': 'INT. REBEL BLOCKADE RUNNER',
 'elements': ['The nervous Rebel troopers aim their weapons. Suddenly a',
              'tremendous blast opens up a hole in the main passageway and',
              'a score of fearsome armored spacesuited stormtroopers make',
              'their way into the smoke-filled corridor.',
              '',
              'In a few minutes the entire passageway is ablaze with',
```

```
                    'laserfire. The deadly bolts ricochet in wild random patterns',
                    'creating huge explosions. Stormtroopers scatter and duck',
                    'behind storage lockers. Laserbolts hit several Rebel soldier
s',
                    'who scream and stagger through the smoke, holding shattered',
                    'arms and faces.',
                    '',
                    'An explosion hits near the robots.',
                    '',
                    'THREEPIO',
                    'I should have known better than to',
                    'trust the logic of a half-sized',
                    'thermocapsulary dehousing assister...',
                    '',
                    'Artoo counters with an angry rebuttal as the battle rages',
                    'around the two hapless robots.']}
```

Part C: Scene cleaning and summarization (Exercises 4-6)

Having extracted the basic structure of a screenplay, let's clean the scenes for later analysis. There will be three tasks:

- Exercise 4 (3 points): "Compressing" the scene, to make it easier to distinguish lines and notes.
- Exercise 5 (2 points): Extracting the names of speaking characters that appear in each scene.
- Exercise 6 (2 points): Tabulating character co-occurrences, in particular, counting how many times two characters have speaking lines in the same scene.

Exercise 4: Scene "compression" (`compress_scene` ; 3 points)

Suppose you are given the elements of a scene in a dictionary, such as the following:

In [21]:
```
demo_scene = scene_elements[179]
assert isinstance(demo_scene, dict) and demo_scene['setting'] == 'INT. MILLENN
IUM FALCON - GUNPORTS'

pretty_print(demo_scene)
```

```
{'setting': 'INT. MILLENNIUM FALCON - GUNPORTS',
 'elements': ['Another TIE fighter moves in on the pirateship and Luke,',
              'smiling, fires the laser cannon at it, scoring a spectacular',
              'direct hit.',
              '',
              'LUKE',
              'Got him! I got him!',
              '',
              'Han turns and gives Luke a victory wave which Luke gleefully',
              'returns.',
              '',
              'HAN',
              "Great kid! Don't get cocky.",
              '',
              'Han turns back to his laser cannon.']}
```

Observe that this example has three notes (`'Another TIE fighter ...'` , `'Han turns and gives ...'` , and `'Han turns back ...'`) and two spoken lines (one by `'LUKE'` and the other by `'HAN'`). Furthermore, recall that the blank-string elements serve to separate notes and lines.

Your task. Complete the function,

```
def compress_scene(scene):
    ...
```

so that it does the following:

1. Joins the text of **each** note or spoken line into a single string, using a single space to join the strings.
2. Converts **each** note or line into a **dictionary** where
   - notes are formatted as `{'speaker': None, 'text': '...'}` ; and
   - lines are formatted as `{'speaker': 'SPEAKER NAME', 'text': '...'}` .
3. Returns a Python list of these notes and lines, where notes and lines appear in the same order as they do in the scene.

For example, for the `demo_scene` above, `compress_scene(demo_scene)` would return the list:

```
[
    {'speaker': None,
     'text': 'Another TIE fighter moves in on the pirateship and Luke, smiling, fir
es the laser cannon at it, scoring a spectacular direct hit.'},
    {'speaker': 'LUKE', 'text': 'Got him! I got him!'},
    {'speaker': None,
     'text': 'Han turns and gives Luke a victory wave which Luke gleefully return
s.'},
    {'speaker': 'HAN', 'text': "Great kid! Don't get cocky."}
]
```

> *Note 0:* If a scene has an empty `scene['elements']` list, your function should return an empty list.
>
> *Note 1:* You may assume that there are no leading or trailing blank elements, i.e., `scene['elements']` will never have first and last elements that are blank.
>
> *Note 2:* You may assume that `scene['elements']` never has two or more consecutive occurrences of blank elements. That is, you might have `['xxx', '', 'xxx', '']` but never `['xxx', '', '', 'xxx']` .
>
> *Note 3:* Although the example does not show it, a character's line may span multiple consecutive non-blank elements.

In [22]:
```
def compress_scene(scene):
    elements = scene['elements']
    assert not elements \
           or (not is_blank(elements[0])
               and not is_blank(elements[-1])
               and all(is_blank(a) + is_blank(b) < 2 for a, b in zip(elements
[:-1], elements[1:])))

    ### BEGIN SOLUTION
    return compress_scene__v1(elements)

# Helper functions
def is_character(e_current, e_next):
    return is_caps(e_current) and not is_blank(e_next)

def new_item():
    return {'speaker': None, 'text': None}

def initialize_speaker(item, e):
    item['speaker'] = e
    item['text'] = ''

def append_text(item, e):
    if item['text']: # not blank - need to add a separating space
        item['text'] += ' ' + e
    else:
        item['text'] = e

# == Method 0 ==
def compress_scene__v0(elements):
    elements = elements + [''] # Append a dummy blank
    i, m = 0, len(elements) # current element index, max element index
    all_items = [] # all lines and notes, returned
    current = new_item()
    while i < len(elements):
        e = elements[i]
        if is_blank(e): # end of current item — recall dummy blank, too
            all_items.append(current)
            current = new_item()
        elif is_character(e, elements[i+1]):
            initialize_speaker(current, e)
        else: # not blank and not a character's name
            append_text(current, e)
        i += 1
    return all_items

# == Method 1 ==
def detect_type(e, e_next):
    if is_blank(e):
        return 'blank'
    elif is_character(e, e_next):
        return 'character'
    else:
        return 'text'
```

```
def compress_scene__v1(elements):
    elements = elements + ['']
    types = [detect_type(e, e_next) for e, e_next in zip(elements[:-1], elemen
ts[1:])]
    all_items = []
    for e, e_type in zip(elements, types):
        if not all_items or e_type == 'blank':
            all_items.append(new_item())
        current_item = all_items[-1]
        if e_type == 'character':
            initialize_speaker(current_item, e)
        elif e_type == 'text':
            append_text(current_item, e)
    return all_items
    ### END SOLUTION
```

In [23]:
```
# Demo cell
compress_scene(demo_scene)
```

Out[23]:
```
[{'speaker': None,
  'text': 'Another TIE fighter moves in on the pirateship and Luke, smiling,
fires the laser cannon at it, scoring a spectacular direct hit.'},
 {'speaker': 'LUKE', 'text': 'Got him! I got him!'},
 {'speaker': None,
  'text': 'Han turns and gives Luke a victory wave which Luke gleefully retur
ns.'},
 {'speaker': 'HAN', 'text': "Great kid! Don't get cocky."},
 {'speaker': None, 'text': 'Han turns back to his laser cannon.'}]
```

In [24]:
```
# Test cell: `mt1_ex4_compress_scene` (3 points)

### BEGIN HIDDEN TESTS
def mt1_ex4__gen_soln__(fn_base="scenes", fn_ext="pickle", overwrite=False):
    from testing_tools import file_exists, save_pickle
    fn = f"{fn_base}.{fn_ext}"
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping...")
    else: # not file_exists(fn) or overwrite
        print(f"'{fn}' does not exist or needs to be overwritten; generatin
g...")

        # Test different versions
        zz0 = [compress_scene__v0(s['elements']) for s in scene_elements]
        zz1 = [compress_scene__v1(s['elements']) for s in scene_elements]
        assert all([a0 == a1 for a0, a1 in zip(zz0, zz1)])

        scenes = [compress_scene(s) for s in scene_elements]
        save_pickle(scenes, fn)


!date
mt1_ex4__gen_soln__(overwrite=False or global_overwrite)
!date
### END HIDDEN TESTS

from testing_tools import mt1_ex4__check
print("Testing...")
for trial in range(250):
    mt1_ex4__check(compress_scene)

compress_scene__passed = True
print(get_mem_usage_str())
print("\n(Passed!)")
```

```
Thu Jun 17 11:32:58 EDT 2021
'scenes.pickle' exists; skipping...
Thu Jun 17 11:32:59 EDT 2021
Testing...
49.5 MiB

(Passed!)
```

Sample results for Exercise 4, `compress_scene => scenes`

If you had a working solution to Exercise 4, then in principle you could use it to compress every scene in `scene_elements`. We have precomputed this result for you; run the cell below create a variable named `scenes`, which holds this result.

> Note: **Read and run this cell even if you skipped or otherwise did not complete Exercise 4.**

In [25]:
```
from testing_tools import mt1_scenes as scenes

print("\nSample results:")
for i, scene in enumerate(scenes[:3]):
    print(f"\n>>> Scene {i} <<<")
    pretty_print(scene)
```

```
Sample results:

>>> Scene 0 <<<
[{'speaker': None,
  'text': 'An explosion rocks the ship as two robots, Artoo-Detoo (R2- D2) an
d '
          'See-Threepio (C-3PO) struggle to make their way through the '
          'shaking, bouncing passageway. Both robots are old and battered. '
          'Artoo is a short, claw-armed tripod. His face is a mass of compute
r '
          'lights surrounding a radar eye. Threepio, on the other hand, is a
'
          'tall, slender robot of human proportions. He has a gleaming '
          'bronze-like metallic surface of an Art Deco design.'},
 {'speaker': None,
  'text': 'Another blast shakes them as they struggle along their way.'},
 {'speaker': 'THREEPIO',
  'text': "Did you hear that? They've shut down the main reactor. We'll be "
          'destroyed for sure. This is madness!'},
 {'speaker': None,
  'text': 'Rebel troopers rush past the robots and take up positions in the '
          'main passageway. They aim their weapons toward the door.'},
 {'speaker': 'THREEPIO', 'text': "We're doomed!"},
 {'speaker': None,
  'text': 'The little R2 unit makes a series of electronic sounds that only '
          'another robot could understand.'},
 {'speaker': 'THREEPIO',
  'text': "There'll be no escape for the Princess this time."},
 {'speaker': None,
  'text': 'Artoo continues making beeping sounds. Tension mounts as loud '
          'metallic latches clank and the scream of heavy equipment are heard
'
          'moving around the outside hull of the ship.'},
 {'speaker': 'THREEPIO', 'text': "What's that?"}]

>>> Scene 1 <<<
[{'speaker': None,
  'text': 'The Imperial craft has easily overtaken the Rebel Blockade Runner.
'
          'The smaller Rebel ship is being drawn into the underside dock of '
          'the giant Imperial starship.'}]

>>> Scene 2 <<<
[{'speaker': None,
  'text': 'The nervous Rebel troopers aim their weapons. Suddenly a tremendou
s '
          'blast opens up a hole in the main passageway and a score of '
          'fearsome armored spacesuited stormtroopers make their way into the
'
          'smoke-filled corridor.'},
 {'speaker': None,
  'text': 'In a few minutes the entire passageway is ablaze with laserfire. '
          'The deadly bolts ricochet in wild random patterns creating huge '
          'explosions. Stormtroopers scatter and duck behind storage lockers.
'
```

```
                'Laserbolts hit several Rebel soldiers who scream and stagger '
                'through the smoke, holding shattered arms and faces.'},
 {'speaker': None, 'text': 'An explosion hits near the robots.'},
 {'speaker': 'THREEPIO',
  'text': 'I should have known better than to trust the logic of a half-sized
'
                'thermocapsulary dehousing assister...'},
 {'speaker': None,
  'text': 'Artoo counters with an angry rebuttal as the battle rages around '
                'the two hapless robots.'}]
```

Important digression: Character voices

For some lines, a character speaks but does not appear visually in the scene. For example, here is an excerpt of scene 143 (`scenes[143]`):

```
[
    {'speaker': None, 'text': 'Solo, Chewie, Luke, and Leia t ...'},
    {'speaker': 'HAN', 'text': "Now's our chance! Go!"},
    {'speaker': None, 'text': 'They start for the Millennium  ...'},
    {'speaker': None, 'text': 'The old Jedi Knight looks over ...'},
    {'speaker': None, 'text': 'Vader brings his sword down, c ...'},
    {'speaker': 'LUKE', 'text': 'No!'},
    {'speaker': None, 'text': 'The stormtroopers turn toward  ...'},
    {'speaker': 'HAN', 'text': '(to Luke) Come on!'},
    {'speaker': 'LEIA', 'text': 'Come on! Luke, its too late!'},
    {'speaker': 'HAN', 'text': 'Blast the door! Kid!'},
    {'speaker': None, 'text': 'Luke fires his pistol at the d ...'},
    {'speaker': "BEN'S VOICE", 'text': 'Run, Luke! Run!'},
    {'speaker': None, 'text': 'Luke looks around to see where ...'}
]
```

There are four different speakers: `'HAN'` , `'LUKE'` , `'LEIA'` , and `"BEN'S VOICE"` . That is, the character `'BEN'` speaks but is only heard in this scene, not seen.

You'll need this fact in the next exercise.

Exercise 5: Extracting character sets (`extract_characters` ; 2 points)

Suppose you are given `scenes` , a Python list of compressed scenes. Complete the function,

```
def extract_characters(scenes):
    ...
```

so that it does the following:

1. For each compressed scene, determine which characters have spoken lines.
2. Return a Python list of Python sets, where each entry of the list corresponds to a scene, and each set is a Python set of all speakers.

However, **there is one more requirement**: if any character is in the scene by voice only (e.g., `"BEN'S VOICE"` rather than `'BEN'`), store the name in the set **without the `"'S VOICE"` suffix.**

For example, suppose `scenes` is as follows:

In [26]:
```
demo_scenes = \
[
    [ # (Compressed) scene 0
        {'speaker': None, 'text': 'Threepio and Artoo-Detoo are i ...'},
        {'speaker': 'THREEPIO', 'text': "Come on, Artoo, we're going!"},
        {'speaker': None, 'text': 'Threepio ducks out of sight as ...'}
    ],
    [ # Scene 1
        {'speaker': None, 'text': 'Solo, Chewie, Luke, and Leia t ...'},
        {'speaker': 'HAN', 'text': "Now's our chance! Go!"},
        {'speaker': None, 'text': 'They start for the Millennium  ...'},
        {'speaker': None, 'text': 'The old Jedi Knight looks over ...'},
        {'speaker': None, 'text': 'Vader brings his sword down, c ...'},
        {'speaker': 'LUKE', 'text': 'No!'},
        {'speaker': None, 'text': 'The stormtroopers turn toward  ...'},
        {'speaker': 'HAN', 'text': '(to Luke) Come on!'},
        {'speaker': 'LEIA', 'text': 'Come on! Luke, its too late!'},
        {'speaker': 'HAN', 'text': 'Blast the door! Kid!'},
        {'speaker': None, 'text': 'Luke fires his pistol at the d ...'},
        {'speaker': "BEN'S VOICE", 'text': 'Run, Luke! Run!'},
        {'speaker': None, 'text': 'Luke looks around to see where ...'}
    ],
    [ # Scene 2
        {'speaker': None, 'text': 'Han pulls back on the controls ...'},
        {'speaker': 'HAN', 'text': 'I hope the old man got that tr ...'},
        {'speaker': None, 'text': 'Chewbacca growls in agreement.'}
    ]
]
```

Then `extract_characters(scenes)` would return:

```
[
    {'THREEPIO'},
    {'HAN', 'LUKE', 'LEIA', 'BEN'}, # <<< 'BEN', **not** "BEN'S VOICE"
    {'HAN'}
]
```

> *Note:* Use an empty set for a scene with no spoken lines.

In [27]:
```
def extract_characters(scenes):
    ### BEGIN SOLUTION
    return extract_characters__v0(scenes)

def clean_speaker(item): # A helper function
    s = item['speaker']
    return s if s is None or s[-8:] != "'S VOICE" else s[:-8]

# Version 0: Basic solution
def extract_characters__v0(scenes):
    characters_by_scene = []
    for scene in scenes:
        speakers = set()
        for item in scene:
            speaker = clean_speaker(item)
            if speaker is not None:
                speakers.add(speaker)
        characters_by_scene.append(speakers)
    return characters_by_scene

# Version 1: Advanced solution, using the new 'walrus operator' of Python 3.8+
def extract_characters__v1(scenes):
    def get_speakers(scene):
        return {speaker for item in scene if (speaker := clean_speaker(item))}
    return [get_speakers(scene) for scene in scenes]
    ### END SOLUTION
```

In [28]:
```
# Demo:
extract_characters(demo_scenes)
```

Out[28]:
```
[{'THREEPIO'}, {'BEN', 'HAN', 'LEIA', 'LUKE'}, {'HAN'}]
```

In [29]:
```
# Test cell: `mt1_ex5_extract_characters` (2 points)

### BEGIN HIDDEN TESTS
def mt1_ex5__gen_soln__(fn_base="characters_by_scene", fn_ext="pickle", overwr
ite=False):
    from testing_tools import file_exists, save_pickle
    fn = f"{fn_base}.{fn_ext}"
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping...")
    else: # not file_exists(fn) or overwrite
        print(f"'{fn}' does not exist or needs to be overwritten; generatin
g...")

        # Compare different versions
        zz0 = extract_characters__v0(scenes)
        zz1 = extract_characters__v1(scenes)
        assert all([a0 == a1 for a0, a1 in zip(zz0, zz1)])

        characters_by_scene = extract_characters(scenes)
        save_pickle(characters_by_scene, fn)


!date
mt1_ex5__gen_soln__(overwrite=False or global_overwrite)
!date
### END HIDDEN TESTS

from testing_tools import mt1_ex5__check
print("Testing...")
for trial in range(250):
    mt1_ex5__check(extract_characters)

extract_characters__passed = True
print(get_mem_usage_str())
print("\n(Passed!)")
```

```
Thu Jun 17 11:32:59 EDT 2021
'characters_by_scene.pickle' exists; skipping...
Thu Jun 17 11:32:59 EDT 2021
Testing...
49.5 MiB

(Passed!)
```

Sample results for Exercise 5, `extract_characters => scene_characters`

If you had a working solution to Exercise 5, then in principle you could use it to extract the characters for every scene of `scenes`. We have precomputed this result for you; run the cell below create a variable named `scene_characters`, which holds this result.

> Note: **Read and run this cell even if you skipped or otherwise did not complete Exercise 5.**

In [30]:
```
from testing_tools import mt1_characters_by_scene as scene_characters

print("\n==> Sample results, for the first 10 scenes:\n")
pretty_print(scene_characters[:10])
```

```
==> Sample results, for the first 10 scenes:

[{'THREEPIO'},
 set(),
 {'THREEPIO'},
 {'LUKE'},
 set(),
 set(),
 {'THREEPIO'},
 {'IMPERIAL OFFICER', 'REBEL OFFICER', 'VADER'},
 {'TROOPER'},
 {'THREEPIO'}]
```

Important digression: Immutable, or "frozen", sets

For the next exercise, you need a new (but simple!) Python concept, which is that of a frozen set (https://docs.python.org/3/library/stdtypes.html#frozenset). It's nothing more than a set that is immutable, i.e., it cannot be changed.

Recall how a Python set works:

In [31]:
```
demo_set = {1, 3, 5}     # Some odd integers
print(3 in demo_set)     # `True`
print(2 not in demo_set) # Also `True`
```

```
True
True
```

Per the Python Primer, also recall one property of sets, which is that they are *mutable*:

In [32]:
```
demo_set.update({2, 4, 6}) # Add some even integers to the set
print(demo_set)
```

```
{1, 2, 3, 4, 5, 6}
```

A mutuable set *cannot* be used as a dictionary key! In the code cell below, if you remove the comment on the second line of code, the cell will fail when executed.

In [33]:
```
demo_dict = {'some-key': 3.1415926536}
#demo_dict[demo_set] = 2.7182818285 # this statement fails with an "unhashable type" error
```

Therefore, if you want to use a set as a key, you need an immutable version. That's what a *frozen set* allows. At first glance, it behaves like a set:

In [34]:
```
demo_frozen_set = frozenset(demo_set) # "Freeze" a regular set
print("1.", type(demo_set))
print("2.", type(demo_frozen_set))
print("3.", demo_set == demo_frozen_set) # Similar!
```

```
1. <class 'set'>
2. <class 'frozenset'>
3. True
```

But if you try to *modify* it, you will get an error. Uncomment the line below if you want to see that behavior:

In [35]:
```
#demo_frozen_set.update({7, 8, 9}) # A frozen set does not have an `update` method
```

Nevertheless, you *can* use it as a dictionary key!

In [36]:
```
demo_dict[demo_frozen_set] = 2.7182818285 # Succeeds!
print(demo_dict)
```

```
{'some-key': 3.1415926536, frozenset({1, 2, 3, 4, 5, 6}): 2.7182818285}
```

You'll use this concept in the next exercise.

Exercise 6: Count character co-occurrences (`count_pairs` ; 2 points)

Suppose you are given `scene_characters` , a Python list of sets: each entry of the list corresponds to a scene, and the entry is a Python set holding the names of the speaking characters in that scene. Complete the function,

```
def count_pairs(scene_characters):
```

so that it does the following:

1. For each element of `scene_characters` , which is a regular Python set, generate all pairs of characters. Store each pair as a **frozen set**. We'll refer to these as "frozen pairs," below.
2. Count the number of scenes in which each frozen pair occurs.
3. Return a Python dictionary, where each key is a frozen pair and the value is its count.

For example, suppose there are four scenes, with the speakers in each scene given as follows:

In [37]:
```
# Run this cell
demo_scene_characters = [
    {'LUKE'},
    {'LUKE', 'BEN'},
    {'LUKE', 'BEN', 'THREEPIO'},
    {'LUKE', 'BEN', 'LEIA', 'THREEPIO'}
]
```

Here are the pairs in each scene:

- `{'LUKE'}` has no pairs, since it has only one character.
- `{'LUKE', 'BEN'}` is a pair. Since it's a set, `{'LUKE', 'BEN'} == {'BEN', 'LUKE'}`, i.e., the ordering does not matter.
- `{'LUKE', 'BEN', 'THREEPIO'}` has 3 pairs, which written as sets are `{'LUKE', 'BEN'}`, `{'LUKE', 'THREEPIO'}`, and `{'BEN', 'THREEPIO'}`.
- `{'LUKE', 'BEN', 'LEIA', 'THREEPIO'}` has 6 pairs, which written as sets are `{'LUKE', 'BEN'}`, `{'LUKE', 'LEIA'}`, `{'LUKE', 'THREEPIO'}`, `{'BEN', 'LEIA'}`, `{'BEN', 'THREEPIO'}`, `{'LEIA', 'THREEPIO'}`.

The counts of each pair across all scenes:

- `{'LUKE', 'BEN'}` appears in three scenes, so its count is 3.
- `{'LUKE', 'THREEPIO'}` appears in two scenes, so its count is 2. The same is true for `{'BEN', 'THREEPIO'}`.
- `{'LUKE', 'LEIA'}`, `{'BEN', 'LEIA'}`, and `{'LEIA', 'THREEPIO'}` only appear in the last scene, so their counts are all 1.

Thus, `count_pairs(demo_scene_characters)` would return the dictionary,

```
{frozenset({'LUKE', 'BEN'}): 3,
 frozenset({'LUKE', 'THREEPIO'}): 2,
 frozenset({'BEN', 'THREEPIO'}): 2,
 frozenset({'LUKE', 'LEIA'}): 1,
 frozenset({'BEN', 'LEIA'}): 1,
 frozenset({'THREEPIO', 'LEIA'}): 1
}
```

In [38]:
```
def count_pairs(scene_characters):
    ### BEGIN SOLUTION
    return count_pairs__v1(scene_characters)

# === Method 0: Notebook 2 with a twist ===
def count_pairs__v0(scene_characters):
    from itertools import combinations
    from collections import defaultdict
    pairs = defaultdict(int)
    for characters in scene_characters:
        for a, b in combinations(characters, 2):
            pairs[frozenset({a, b})] += 1
    return dict(pairs)

# === Method 1 (more advanced concept): Use a "generator" to iterate over pair
-sets ===
def all_pair_sets(x):
    """Yield each pair of `x` as a `frozenset`."""
    from itertools import combinations
    for a, b in combinations(x, 2):
        yield frozenset({a, b})

def count_pairs__v1(scene_characters):
    from collections import defaultdict
    pairs = defaultdict(int)
    for characters in scene_characters:
        for s in all_pair_sets(characters):
            pairs[s] += 1
    return dict(pairs)
    ### END SOLUTION
```

In [39]:
```
# Demo cell:
count_pairs(demo_scene_characters)
```

Out[39]:
```
{frozenset({'BEN', 'LUKE'}): 3,
 frozenset({'LUKE', 'THREEPIO'}): 2,
 frozenset({'BEN', 'THREEPIO'}): 2,
 frozenset({'LEIA', 'LUKE'}): 1,
 frozenset({'BEN', 'LEIA'}): 1,
 frozenset({'LEIA', 'THREEPIO'}): 1}
```

In [40]:
```
# Test cell: `mt1_ex6_count_pairs` (2 points)

### BEGIN HIDDEN TESTS
def mt1_ex6__gen_soln__(fn_base="character_co_occurrences", fn_ext="pickle", o
verwrite=False):
    from testing_tools import file_exists, save_pickle
    fn = f"{fn_base}.{fn_ext}"
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping...")
    else: # not file_exists(fn) or overwrite
        print(f"'{fn}' does not exist or needs to be overwritten; generatin
g...")

        # Compare different versions
        zz0 = extract_characters__v0(scenes)
        zz1 = extract_characters__v1(scenes)
        assert all([a0 == a1 for a0, a1 in zip(zz0, zz1)])

        character_co_occurrences = count_pairs(scene_characters)
        save_pickle(character_co_occurrences, fn)


!date
mt1_ex6__gen_soln__(overwrite=False or global_overwrite)
!date
### END HIDDEN TESTS

from testing_tools import mt1_ex6__check
print("Testing...")
for trial in range(250):
    mt1_ex6__check(count_pairs)

count_pairs__passed = True
print(get_mem_usage_str())
print("\n(Passed!)")
```

```
Thu Jun 17 11:33:00 EDT 2021
'character_co_occurrences.pickle' exists; skipping...
Thu Jun 17 11:33:00 EDT 2021
Testing...
49.6 MiB

(Passed!)
```

Sample results for Exercise 6, `count_pairs => character_co_occurrences`

If you had a working solution to Exercise 6, then in principle you could use it to count all character-pairs in `scene_characters`. We have precomputed this result for you; run the cell below create a variable named `character_co_occurrences`, which holds this result.

> Note: **Read and run this cell even if you skipped or otherwise did not complete Exercise 6.**

In [41]:
```
from testing_tools import mt1_character_co_occurrences as character_co_occurrences

print("\n==> Sample results:\n")
for p in ['BEN', 'HAN', 'LEIA', 'THREEPIO']:
    s = {p, 'LUKE'}
    print("* The pair", s, "appears", character_co_occurrences[frozenset(s)], "time(s).")
```

```
==> Sample results:

* The pair {'LUKE', 'BEN'} appears 21 time(s).
* The pair {'LUKE', 'HAN'} appears 26 time(s).
* The pair {'LEIA', 'LUKE'} appears 17 time(s).
* The pair {'LUKE', 'THREEPIO'} appears 18 time(s).
```

Part D: Scene analysis (Exercise 7, the last one!)

The co-occurrence counts give us a way to measure the "strength" of the relationship between two characters. Let's assume that if two characters appear together in more scenes, then the more important their relationship is. Given a measure of such relationships, a **graph clustering algorithm** can try to analyze the connections between characters and divide (or *cluster*) them into groups, where we believe that members of a group are more strongly connected to one another than they are to those outside their group.

We are providing some code to calculate this clustering and visualize the results. But in this last exercise, Exercise 7, you need to postprocess those results.

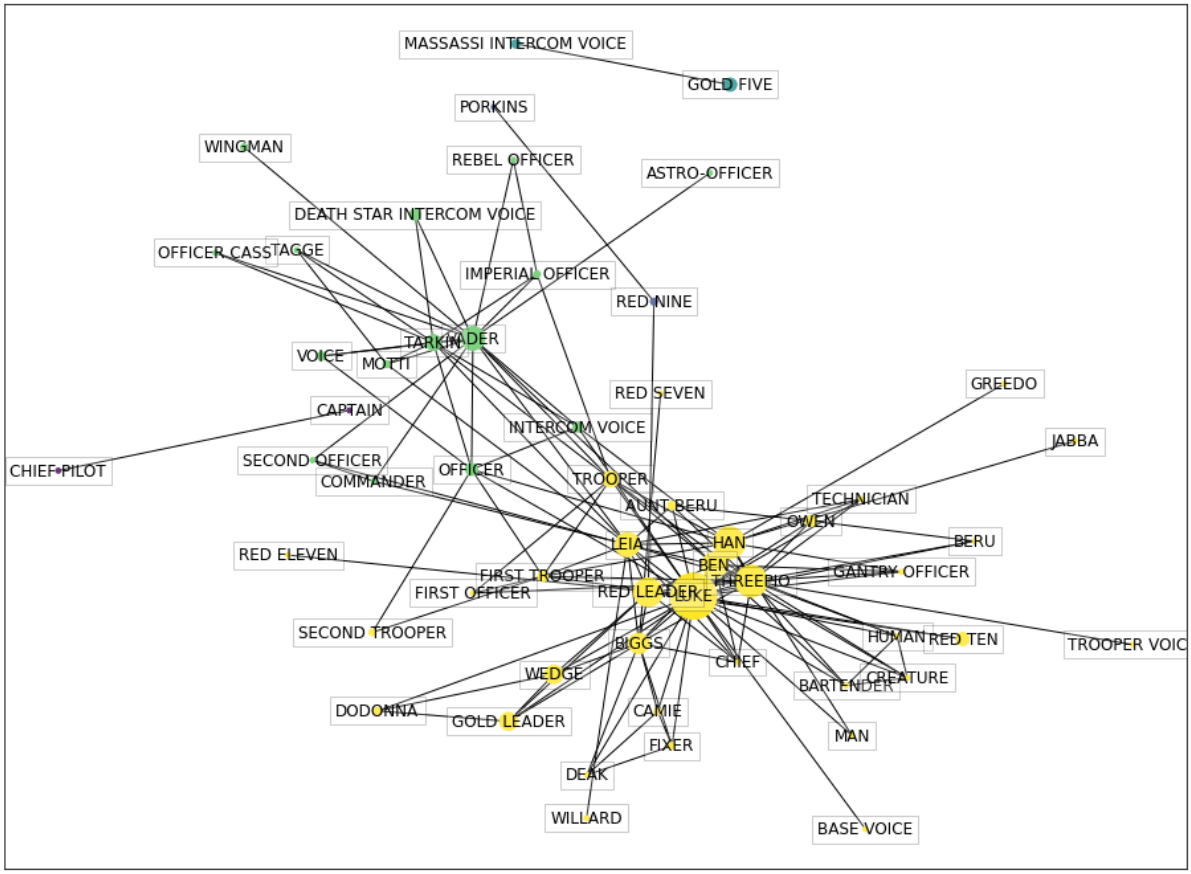
But first...

Digression: Graph clustering

We've provided some code for you that takes your co-occurrence count data and then uses the Python NetworkX (https://networkx.org/) module to automatically discover a grouping, or clustering, of the characters based on the strengths of their interrelationships. You'll learn more about *how* this technique works later in the semester, but for now, just run the code cell below to calculate the clusters.

In [42]:
```
from testing_tools import count_characters, cluster, viz_clustering

character_counts = count_characters(scene_characters)
character_graph, communities = cluster(character_co_occurrences)
viz_clustering(character_graph, communities, character_counts)
```



Each dot is a character, whose size increases with the number of spoken lines that character has. An edge indicates that a pair of characters have speaking lines in at least one scene together. The colors indicate the clusters. From it, you might be able to see, for example, that `'LUKE'`, `'HAN'`, and `'LEIA'` are a core part of one cluster, whereas `'VADER'`, `'TARKIN'`, and `'IMPERIAL OFFICER'` are part of a different cluster.

The visualization is nice, but we might want to take the raw clustering results and do something with those, too. In particular, the clustering produces a dictionary named `communities`: the keys are character names and the values are cluster labels, which are integer IDs. If you inspect the output below, you'll see the clustering found 5 clusters, numbered 0-4: cluster 0 contains `'CAPTAIN'` and `'CHIEF PILOT'`, cluster 1 contains `'PORKINS'` and `'RED NINE'`, and so on.

In [43]:
```
# Run me to see the cluster IDs for each character:
communities
```

Out[43]:
```
{'CHIEF PILOT': 0,
 'CAPTAIN': 0,
 'RED NINE': 1,
 'PORKINS': 1,
 'GOLD FIVE': 2,
 'MASSASSI INTERCOM VOICE': 2,
 'INTERCOM VOICE': 3,
 'MOTTI': 3,
 'IMPERIAL OFFICER': 3,
 'TARKIN': 3,
 'VOICE': 3,
 'WINGMAN': 3,
 'SECOND OFFICER': 3,
 'OFFICER CASS': 3,
 'OFFICER': 3,
 'ASTRO-OFFICER': 3,
 'TAGGE': 3,
 'DEATH STAR INTERCOM VOICE': 3,
 'REBEL OFFICER': 3,
 'COMMANDER': 3,
 'VADER': 3,
 'HAN': 4,
 'WEDGE': 4,
 'DEAK': 4,
 'DODONNA': 4,
 'LUKE': 4,
 'RED ELEVEN': 4,
 'HUMAN': 4,
 'MAN': 4,
 'BEN': 4,
 'TROOPER VOICE': 4,
 'LEIA': 4,
 'BERU': 4,
 'FIRST OFFICER': 4,
 'TROOPER': 4,
 'SECOND TROOPER': 4,
 'FIXER': 4,
 'GOLD LEADER': 4,
 'BIGGS': 4,
 'JABBA': 4,
 'RED SEVEN': 4,
 'OWEN': 4,
 'THREEPIO': 4,
 'GANTRY OFFICER': 4,
 'TECHNICIAN': 4,
 'FIRST TROOPER': 4,
 'GREEDO': 4,
 'BARTENDER': 4,
 'CHIEF': 4,
 'WILLARD': 4,
 'RED TEN': 4,
 'BASE VOICE': 4,
 'AUNT BERU': 4,
 'RED LEADER': 4,
```

```
            'CREATURE': 4,
            'CAMIE': 4}
```

Exercise 7: Find the k-largest communities (`k_largest_communities` ; 2 points)

Suppose you are given a dictionary that maps names to cluster labels, like `communities` , above. Complete the function,

```
def k_largest_communities(communities, k):
    ...
```

so that it finds and returns the `k` largest clusters. For example, if `k=3` , that means we want to know which 3 clusters are the largest.

There are some corner cases, so let's be more precise about what your function needs to do:

1. for each unique cluster label, it constructs a Python set holding all the names belonging to that cluster;
2. it ranks the clusters from largest to smallest by size (number of names);
3. it returns a **Python list** containing the `k` largest sets.

Important corner cases: In the case of ties, include all clusters with the same count. Therefore, your function might return more than `k` clusters. And if there are fewer than `k` distinct labels, then your function should return all clusters.

For example, suppose the given communities are as follows:

In [44]:
```
demo_comms = {
    #
    'INTERCOM VOICE': 0,
    'DODONNA': 0,
    'JABBA': 0,
    'HUMAN': 0,
    #
    'TARKIN': 1,
    'GOLD LEADER': 1,
    #
    'TROOPER': 2,
    'VOICE': 2,
    'GREEDO': 2,
    #
    'TECHNICIAN': 3,
    'RED NINE': 3,
    #
    'TROOPER VOICE': 4
}
```

Observe that there are 5 communities:

- Cluster 0 has 4 names: `{'INTERCOM VOICE', 'DODANNA', 'JABBA', 'HUMAN'}`.
- Cluster 1 has 2 names: `{'TARKIN', 'GOLD LEADER'}`
- Cluster 2 has 3 names: `{'TROOPER', 'VOICE', 'GREEDO'}`
- Cluster 3 has 2 names: `{'TECHNICIAN', 'RED NINE'}`
- Cluster 4 has 1 name: `{'TROOPER VOICE'}`.

Therefore:

- Suppose `k=1`, meaning we want the largest cluster. That is Cluster 0, so `k_largest_communities(demo_comms, k=1)` would return the 1-element list, `[{'INTERCOM VOICE', 'DODANNA', 'JABBA', 'HUMAN'}]`.
- Suppose `k=2`. The two largest cluster sizes are 4 (Cluster 0) and 3 (Cluster 2). Therefore, your code should return the 2-element list, `[{'INTERCOM VOICE', 'DODANNA', 'JABBA', 'HUMAN'}, {'TROOPER', 'VOICE', 'GREEDO'}]`.
- For `k=3`, the three largest cluster sizes are 4 (Cluster 0), 3 (Cluster 2), and 2 (Clusters 1 and 3). There is a two-way tie for third place. Therefore, your code should return a **4-element** list (even though `k=3`) with the sets for Clusters 0, 2, 1, and 3.
- For `k=4`, the four largest cluster sizes are the same as for `k=3` because of the tie. Therefore, your code should return the same result as `k=3`.
- For `k=5`, your code would return all five clusters, since there are only five clusters.

> *Note 0:* If `k=0`, your code should return an empty list. If `k` is *greater than* the number of clusters, your code should just return all the clusters.
>
> *Note 1:* The order in which your code returns clusters does not matter, as long as it includes them all.

In [45]:
```
def k_largest_communities(communities, k):
    assert k >= 0
    ### BEGIN SOLUTION
    return k_largest_communities__v1(communities, k)

# Helper function: Groups members of each community together.
# Let `g` be a cluster label and `S` be the set of its members.
# This function returns a dictionary mapping each `g` to its `S`.
def get_groups(communities):
    from collections import defaultdict
    groups = defaultdict(set)
    for character, group in communities.items():
        groups[group] |= {character} # alternative: `groups[group].add(charact
er)`
    return groups

# === Method 0: First create groups, then sort ===
def k_largest_communities__v0(communities, k):
    return get_top_groups(get_groups(communities), k)

# Given a bunch of `groups`, find the `k` largest.
def get_top_groups(groups, k):
    sorted_groups = sorted(list(groups.values()), key=lambda x: -len(x)) # dec
reasing-sort by size
    top, remaining = sorted_groups[:k], sorted_groups[k:] # split groups, igno
ring ties
    if top: # check for ties with the last element of `top`
        while remaining and len(remaining[0]) == len(top[-1]):
            top.append(remaining.pop(0))
    return top

# === Method 1: Variation ===
def k_largest_communities__v1(communities, k):
    comm_set_dict = get_groups(communities)
    members = list(comm_set_dict.values())
    lengths = sorted((len(m) for m in members), reverse=True)
    k = min([k, len(members)])
    return [] if k == 0 else sorted((m for m in members if len(m) >= lengths[k
-1]), key=len, reverse=True)
    ### END SOLUTION
```

In [46]:
```
# Demo cell:
print("Cluster labels:\n")
pretty_print(demo_comms)

for k in range(7):
    print(f"\n>>> k={k} <<<")
    pretty_print(k_largest_communities(demo_comms, k))
```

```
Cluster labels:

{'INTERCOM VOICE': 0,
 'DODONNA': 0,
 'JABBA': 0,
 'HUMAN': 0,
 'TARKIN': 1,
 'GOLD LEADER': 1,
 'TROOPER': 2,
 'VOICE': 2,
 'GREEDO': 2,
 'TECHNICIAN': 3,
 'RED NINE': 3,
 'TROOPER VOICE': 4}

>>> k=0 <<<
[]

>>> k=1 <<<
[{'INTERCOM VOICE', 'JABBA', 'HUMAN', 'DODONNA'}]

>>> k=2 <<<
[{'INTERCOM VOICE', 'JABBA', 'HUMAN', 'DODONNA'},
 {'GREEDO', 'VOICE', 'TROOPER'}]

>>> k=3 <<<
[{'INTERCOM VOICE', 'JABBA', 'HUMAN', 'DODONNA'},
 {'GREEDO', 'VOICE', 'TROOPER'},
 {'TARKIN', 'GOLD LEADER'},
 {'RED NINE', 'TECHNICIAN'}]

>>> k=4 <<<
[{'INTERCOM VOICE', 'JABBA', 'HUMAN', 'DODONNA'},
 {'GREEDO', 'VOICE', 'TROOPER'},
 {'TARKIN', 'GOLD LEADER'},
 {'RED NINE', 'TECHNICIAN'}]

>>> k=5 <<<
[{'INTERCOM VOICE', 'JABBA', 'HUMAN', 'DODONNA'},
 {'GREEDO', 'VOICE', 'TROOPER'},
 {'TARKIN', 'GOLD LEADER'},
 {'RED NINE', 'TECHNICIAN'},
 {'TROOPER VOICE'}]

>>> k=6 <<<
[{'INTERCOM VOICE', 'JABBA', 'HUMAN', 'DODONNA'},
 {'GREEDO', 'VOICE', 'TROOPER'},
 {'TARKIN', 'GOLD LEADER'},
 {'RED NINE', 'TECHNICIAN'},
 {'TROOPER VOICE'}]
```

In [47]:
```
# Test cell: `mt1_ex7_k_largest_communities` (2 points)

### BEGIN HIDDEN TESTS
def mt1_ex7__gen_soln__(fn_base="largest_communities", fn_ext="pickle", overwr
ite=False):
    from testing_tools import file_exists, save_pickle
    fn = f"{fn_base}.{fn_ext}"
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping...")
    else: # not file_exists(fn) or overwrite
        print(f"'{fn}' does not exist or needs to be overwritten; generatin
g...")
        largest_communities = k_largest_communities(communities, k=2)
        save_pickle(largest_communities, fn)

!date
mt1_ex7__gen_soln__(overwrite=False or global_overwrite)
!date
### END HIDDEN TESTS

from testing_tools import mt1_ex7__check
print("Testing...")
for trial in range(250):
    mt1_ex7__check(k_largest_communities)

k_largest_communities__passed = True
print(get_mem_usage_str())
print("\n(Passed!)")
```

```
Thu Jun 17 11:33:02 EDT 2021
'largest_communities.pickle' exists; skipping...
Thu Jun 17 11:33:02 EDT 2021
Testing...
112.8 MiB

(Passed!)
```

Sample results for Exercise 7, `k_largest_communities` => `largest_communities`

If you had a working solution to Exercise 7, then in principle you could use it to analyze the cluster results that yielded `communities`. We have precomputed this result for you in the case of `k=2`; run the cell below create a variable named `largest_communities`, which holds this result.

> Note: **Read and run this cell even if you skipped or otherwise did not complete Exercise 7.**

In [48]:
```
from testing_tools import mt1_largest_communities as largest_communities

print("\nThe two largest communities in the Star Wars network:\n")
pretty_print(largest_communities)
```

```
The two largest communities in the Star Wars network:

[{'AUNT BERU',
  'BARTENDER',
  'BASE VOICE',
  'BEN',
  'BERU',
  'BIGGS',
  'CAMIE',
  'CHIEF',
  'CREATURE',
  'DEAK',
  'DODONNA',
  'FIRST OFFICER',
  'FIRST TROOPER',
  'FIXER',
  'GANTRY OFFICER',
  'GOLD LEADER',
  'GREEDO',
  'HAN',
  'HUMAN',
  'JABBA',
  'LEIA',
  'LUKE',
  'MAN',
  'OWEN',
  'RED ELEVEN',
  'RED LEADER',
  'RED SEVEN',
  'RED TEN',
  'SECOND TROOPER',
  'TECHNICIAN',
  'THREEPIO',
  'TROOPER',
  'TROOPER VOICE',
  'WEDGE',
  'WILLARD'},
 {'ASTRO-OFFICER',
  'COMMANDER',
  'DEATH STAR INTERCOM VOICE',
  'IMPERIAL OFFICER',
  'INTERCOM VOICE',
  'MOTTI',
  'OFFICER',
  'OFFICER CASS',
  'REBEL OFFICER',
  'SECOND OFFICER',
  'TAGGE',
  'TARKIN',
  'VADER',
  'VOICE',
  'WINGMAN'}]
```

Fin!

You’ve reached the end of this part. Don’t forget to restart and run all cells again to make sure it’s all working when run in sequence; and make sure your work passes the submission process. Good luck!

If you enjoyed this problem, please thank one of your TAs, Albert Waldron, who came up with the idea and prototyped an initial version.