Investigating Texts and Calls

Problem Definition

- In this problem I need to complete 5 tasks, based on a fabricated files of **calls** and **texts**;
    - The problem is broken beforehand on 5 tasks that I need to solve and analyze;
- I will use Python to analyze and answer questions about the texts and calls contained in the datasets;
- I will perform runtime and space analysis of my solutions;

Steps to solve the problem

Pythonista's Guide to All problems in the Galaxy:

- Understand the Inputs;
- Understand the Outputs;
- Understand the connection between inputs and outputs;
- Create Initial Solution as a human;
- Create a simple mechanical solution;
- Analyse the performance;
- Create iterative solution if needed for a better performance;

Input Requirements

All telephone numbers are **10 or 11 numerical digits long**. Each telephone number starts with a code indicating the **location** and/or **type of the telephone number**. There are three different kinds of telephone numbers, each with a different format:

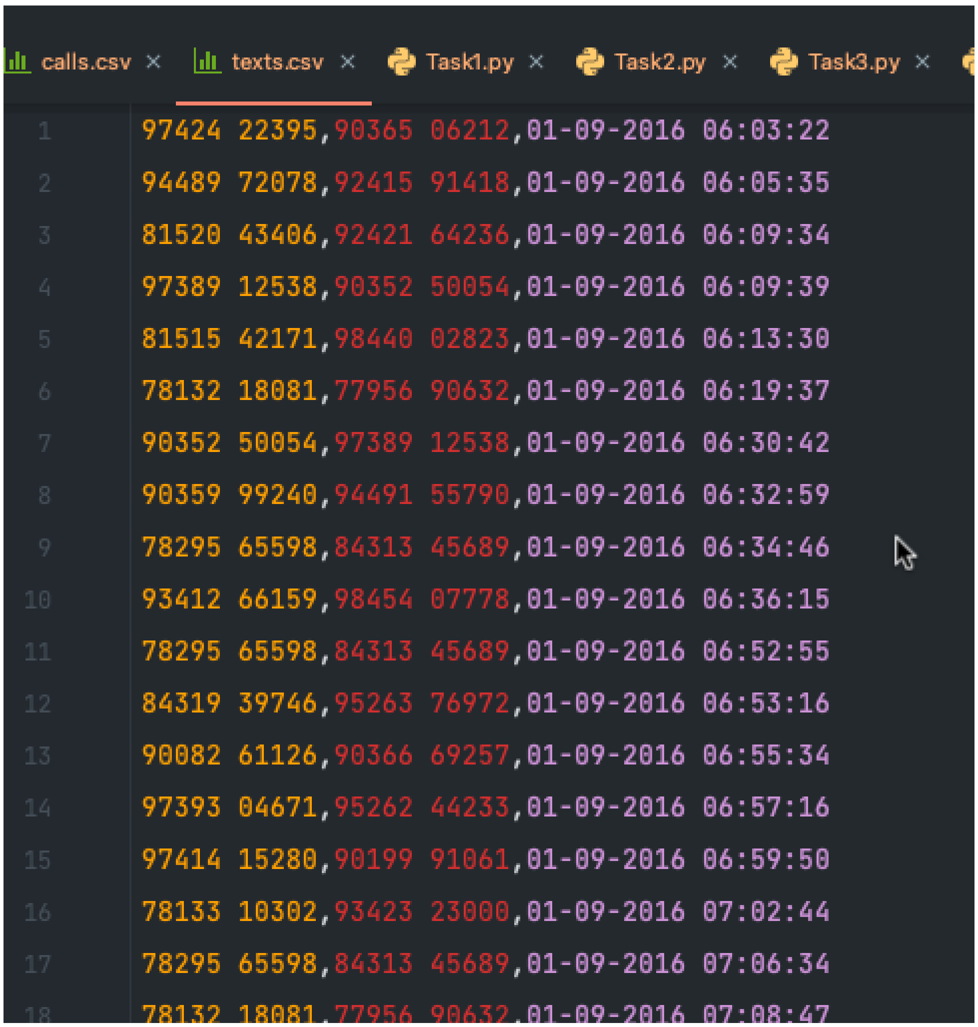
- ***Fixed lines*** - **start with an area code enclosed in brackets**. The **area codes vary in length but always begin with 0**. Example: "(**022**)40840621".
- ***Mobile numbers*** - have **no parentheses, but have a space in the middle of the number to help readability**. The **mobile code** of a mobile number **is its first four digits and they always start with 7, 8 or 9**. Example: "**9341**2 66159".

- ***Telemarketers*** - numbers have **no parentheses or space**, but **start with the code 140**. Example: "**140**2316533".

Messages File

Text data file (text.csv) in CSV format with following columns:

- sending telephone number (string)
- receiving telephone number (string)
- timestamp of text message (string)

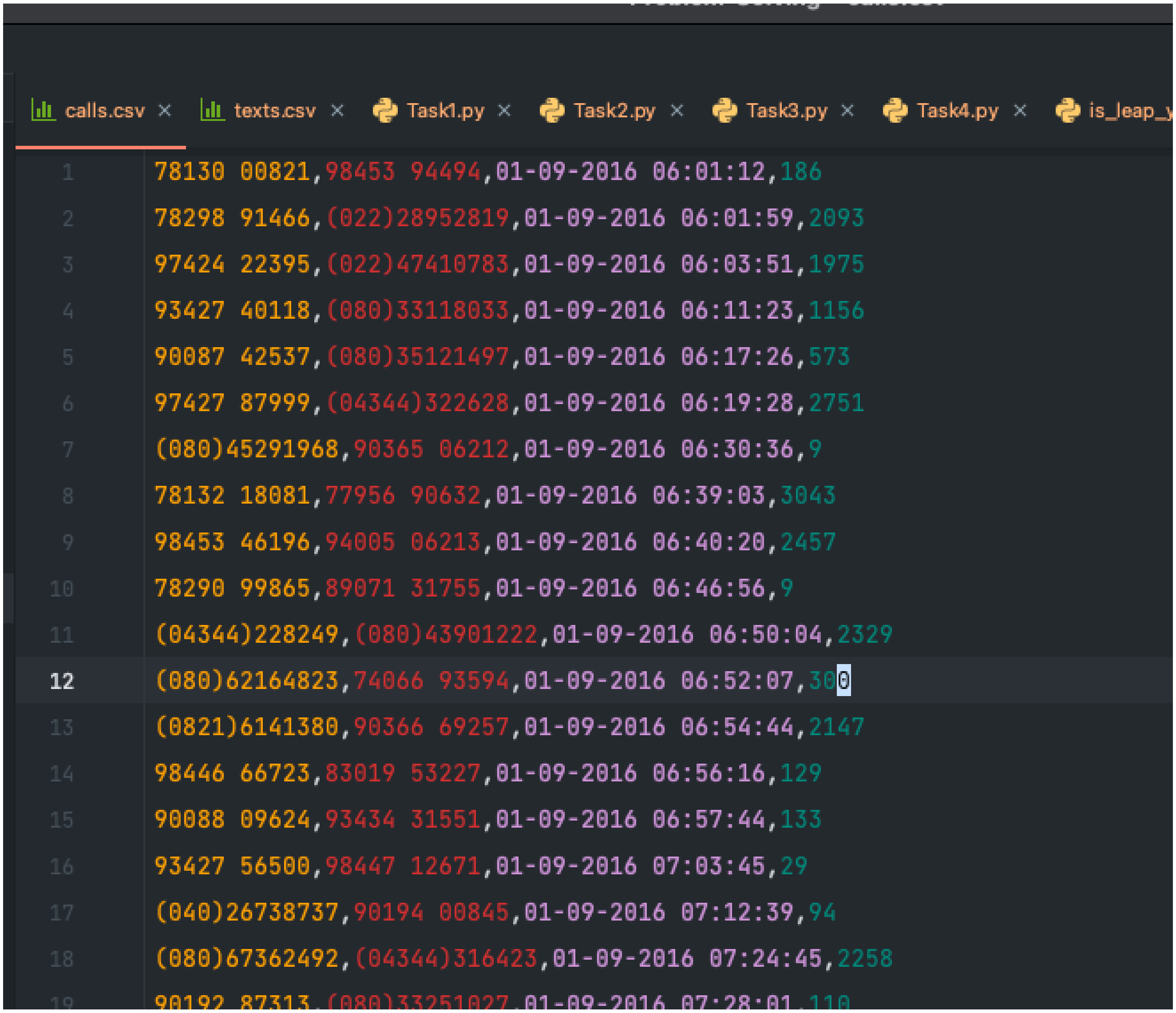



```
97424 22395,90365 06212,01-09-2016 06:03:22
94489 72078,92415 91418,01-09-2016 06:05:35
81520 43406,92421 64236,01-09-2016 06:09:34
97389 12538,90352 50054,01-09-2016 06:09:39
81515 42171,98440 02823,01-09-2016 06:13:30
78132 18081,77956 90632,01-09-2016 06:19:37
90352 50054,97389 12538,01-09-2016 06:30:42
90359 99240,94491 55790,01-09-2016 06:32:59
78295 65598,84313 45689,01-09-2016 06:34:46
93412 66159,98454 07778,01-09-2016 06:36:15
78295 65598,84313 45689,01-09-2016 06:52:55
84319 39746,95263 76972,01-09-2016 06:53:16
90082 61126,90366 69257,01-09-2016 06:55:34
97393 04671,95262 44233,01-09-2016 06:57:16
97414 15280,90199 91061,01-09-2016 06:59:50
78133 10302,93423 23000,01-09-2016 07:02:44
78295 65598,84313 45689,01-09-2016 07:06:34
78132 18081,77956 90632,01-09-2016 07:08:47
```

Phone Call File

Text data file (call.csv) in CSV format with following columns:

- calling telephone number (string)
- receiving telephone number (string)
- start timestamp of the call (string)
- duration of telephone call in seconds (string)



Space Complexity

In Python is less clear how to calculate the Space efficiency due to the underlying data structures for housekeeping. So I will borrow from C and C++ the following sizes: [1]

Type	Storage size
char	1 byte
bool	1 byte
int	4 bytes
float	4 bytes
double	8 bytes

When calculating the list Space complexity I'm not taking into consideration any of the metadata used by Python to store and housekeep the structures.

Also I'm not taking into consideration any of the environment or instructional space at that point.

Calls and Texts lists in Python are list of lists, example form:

```
calls = [
    ['78130 00821', '98453 94494', '01-09-2016 06:01:12', '186'],
    ['78298 91466', '(022)28952819', '01-09-2016 06:01:59', '2093']
    .
    .
    .
]

texts = [
    ['97424 22395', '90365 06212', '01-09-2016 06:03:22'],
    ['94489 72078', '92415 91418', '01-09-2016 06:05:35'],
    .
    .
    .
]
```

All input data stored in the CSV file is stored and represented as String.

Representing single Phone call record

Telephone numbers (calling and receiving) can be:

- 10 or 11 symbols (+2 additional for brackets), so I will calculate it as a worst-case 13 symbols.
- every string symbol is represented as character, and the string as a sequence of chars will take up to: $13\ Bytes$

Timestamp is represented with 19 character symbols or $19\ Bytes$

Duration - there is no limit on the phone duration, and I will assume as worst case scenario that the time duration is no longer than 9999 seconds, or 4 characters. This results in $4\ Bytes$

The final calculation for Space complexity based on that will be:
$O(2 \cdot 13\ Bytes + 19\ Bytes + 4\ Bytes) = 49\ Bytes$

Representing single Texts record

Telephone numbers (sending and receiving) are represented by worst-case 13 symbols record (same as the Phone call).

Timestamp is represented with 19 character symbols or $19\ Bytes$

Final calculation for single text record (constant size):
$O(2 \cdot 13\ Bytes + 19\ Bytes) = 45\ Bytes$

Storing Texts and Phone Calls

The final calculation for Space Complexity is:
`texts[]` list: $O(n \cdot 45\ Bytes)$
`calls[]` list: $O(n \cdot 49\ Bytes)$

Live Experiment

The experiment was conducted on MacBook Pro (Intel Processor), using:

- PyCharm Professional[2];
- Python 3.9[3];
- and Python Memory-Profiler module[4].

Rough calculations for texts list: it appears that most of the records are only mobile numbers (11 chars):
$11\ Bytes \cdot 2 + 19\ Bytes = 41\ Bytes \cdot 9072\ records = 371952\ Bytes$ or $0.354721\ Megabytes$

But actually the list takes around: $2.7\ MiB$

```
Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     9     16.1 MiB     16.1 MiB           1   @profile
    10                                         def read_texts():
    11                                             global f, reader, 
    12     16.1 MiB      0.0 MiB           1       with open('texts.c
    13     16.1 MiB      0.0 MiB           1           reader = csv.r
    14
```

```
    15                                         # List of list
    16     18.7 MiB      2.7 MiB           1           texts = list(re
    17
    18                                         # Access the fi
    19     18.7 MiB      0.0 MiB           1           first_record =
    20     18.7 MiB      0.0 MiB           1           income_number =
    21     18.7 MiB      0.0 MiB           1           answering_numbe
    22     18.7 MiB      0.0 MiB           1           timestamp = fir

    23
    24                                         # Output the re
    25     18.7 MiB      0.0 MiB           1           print(f'First r
```

I'm not sure why the result differs so much, even if single list record representation takes 2 times or 3 times the size of the strings, the result does not come close to the actual. Need `#further-investigation` , or during the Nanodegree I may receive the answers.

Problems to be solved

The problem (or project) is divided into 5 tasks, that are predefined by Udacity. Each task needs to solve a problem, and analyze the solution to the problem. At first glance over the project requirements I'm **not seeing any requirements for Space or Time complexities**.

However, the final solution needs to be checked against Udacity Rubric?, and submitted for review by Udacity reviewer.

In task 3 and Task 4, I can use built-in Python methods `sorted()` and `list.sort()`, which are implementations of Timsort[5] and Samplesort[6]. I can read further how to use these Python Built-in function on Python documentation[7]. In Python documentation I can also find more information and analyses for Time complexity[8] on the sorting algorithms used.

General Bio O Cheetsheet can be downloaded from Know Thy Complexities![9].

Task 0: Printing First and Last records

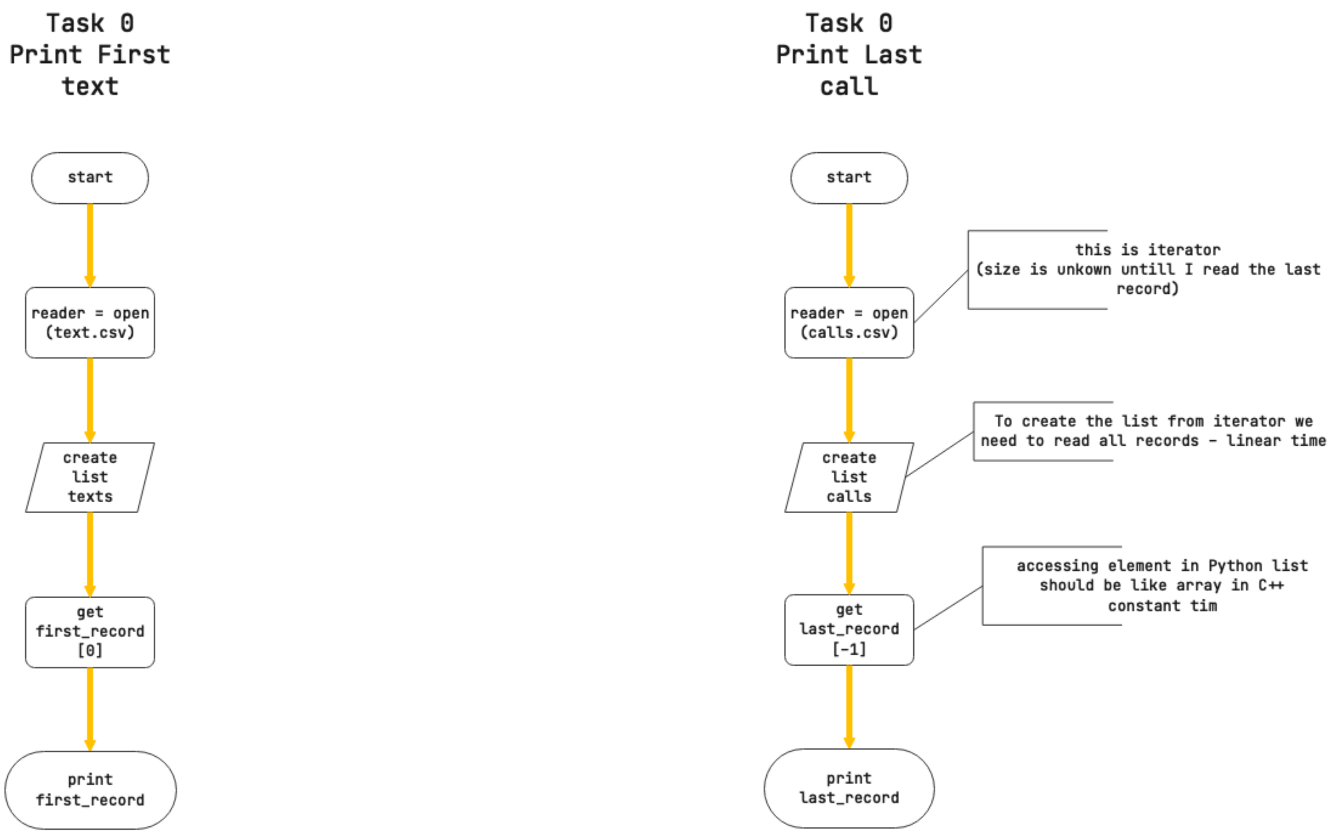
First task of the project, requires from me to:

- print first record from texts file;
- print last record from calls file;

Example of the output:

```
"First record of texts, <incoming number> texts <answering number> at
"Last record of calls, <incoming number> calls <answering number> at
```

Flowchart of the solution



Analyses on Printing First record from Texts

- using Python built-in function `open` to load file: $O(1)$ (**constant time**), this is actually a pointer to a system handle (C++);
- creating reader iterator: $O(1)$ (**constant time**);
- **(Operation: Insert)**[10] - Creating text record in Python list from iterator, for every input record: $O(n)$ (**linear time**);
- **(Operation: Get Item)**[10-1] - getting an item from a list has constant time complexity;
- output to console operation has constant time complexity;

The most demanding operation is the creation of texts list from iterator, which is linear iteration over CSV file.
Task 0 - Print first item from list: $O(n)$

Analyses on Printing Last record from Calls

Printing last record from a CSV is similar to printing the first record. There is one difference here, we are getting the **last item**. If we were talking for C++, I had to carefully analyze also the type of list implemented for the operation Time Complexity calculation. However, in Python **getting item from list** is **constant time operation**, without taking into consideration the place of the element that I seek.

Taking into consideration the above mentioned implementation specifics, I can conclude that the complexity for printing last record from CSV file is with the same complexity: Task 0 - Print last item from list: $O(n)$

Task 0 time complexity

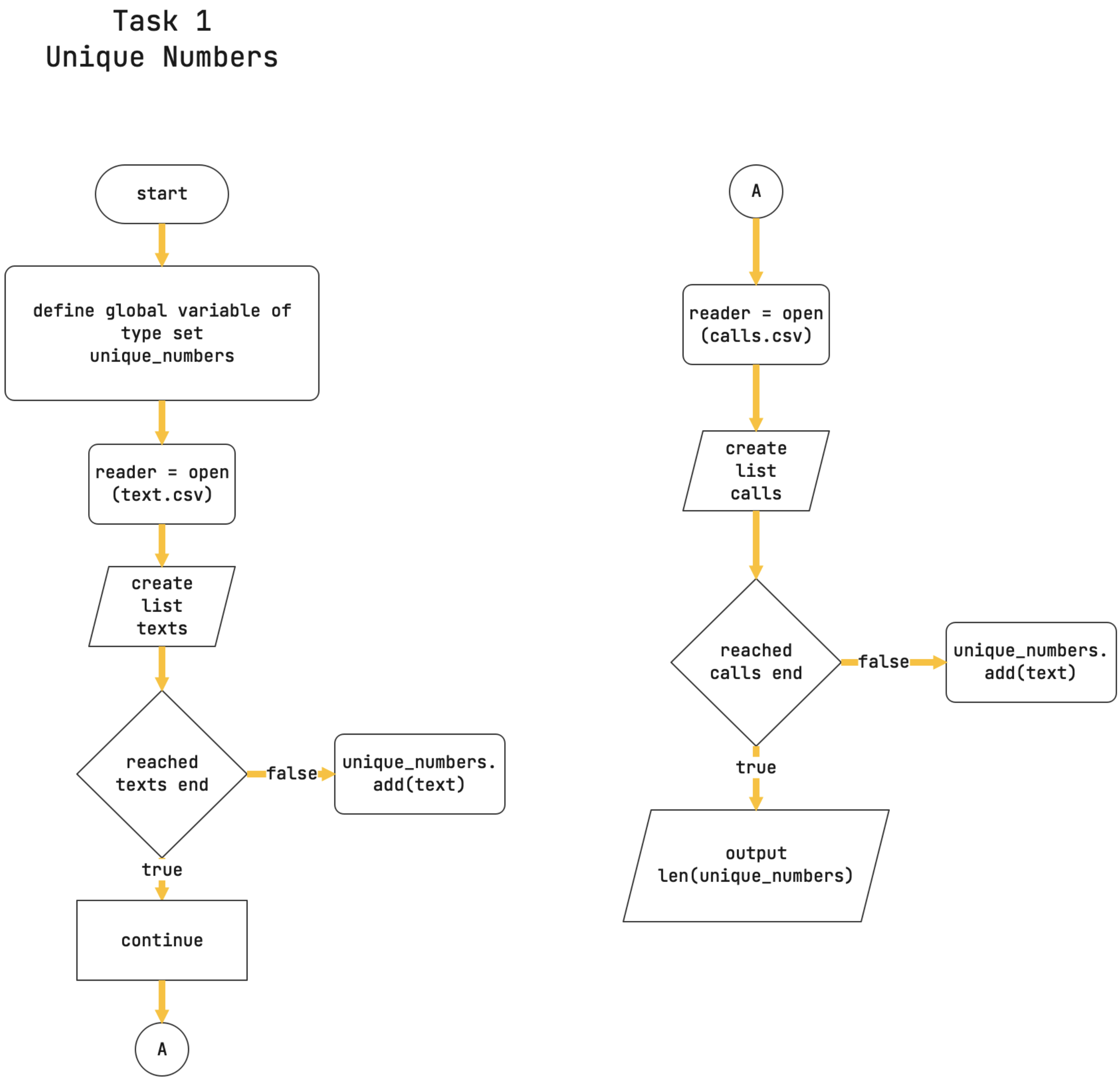
For the whole task (considered as a module) time complexity is:
$O(2 \cdot n)$

Task 1: Counting unique telephone numbers

For the second task I need to calculate how many unique numbers are the records. It is not clearly specified whether or not I need to count numbers from Phone file, or from Texts file, also it is not specified whether or not I need to count In or Out telephone numbers so I do assume by default that I need to calculate numbers from both `calls.csv` and and `texts.csv`. Columns used to count unique numbers:

- sending telephone number (texts.csv)
- receiving telephone number (texts.csv)
- calling telephone number (calls.csv)
- receiving telephone number (calls.csv)

Flowchart of the solution



Analysis on Counting unique numbers

To count the unique numbers as described in the Flowchart I need to iterate twice every input file, which translates to $O(2 \cdot n)$.

Once:

- Create global set variable (**Constant time**")
- Print output length of `unique_numbers` set (**Constant time**)

For every input file (texts and calls):

- Open python file (**Constant time**)
- Create Iterator (**Constant time**)
- Create list from reader iterator - $O(n)$
- For every input, add numbers (in/out) to global set $O(n)$

Task 1 time complexity

For the whole task (considered as a module) time complexity is:
$O(4 \cdot n)$

Task 2: Number with most time on phone

Initially I misunderstand the problem to be solved, so I need second pass on the problem.

Initially I considered:
In this task I need to find the longest Phone call that is stored in `calls.csv`.

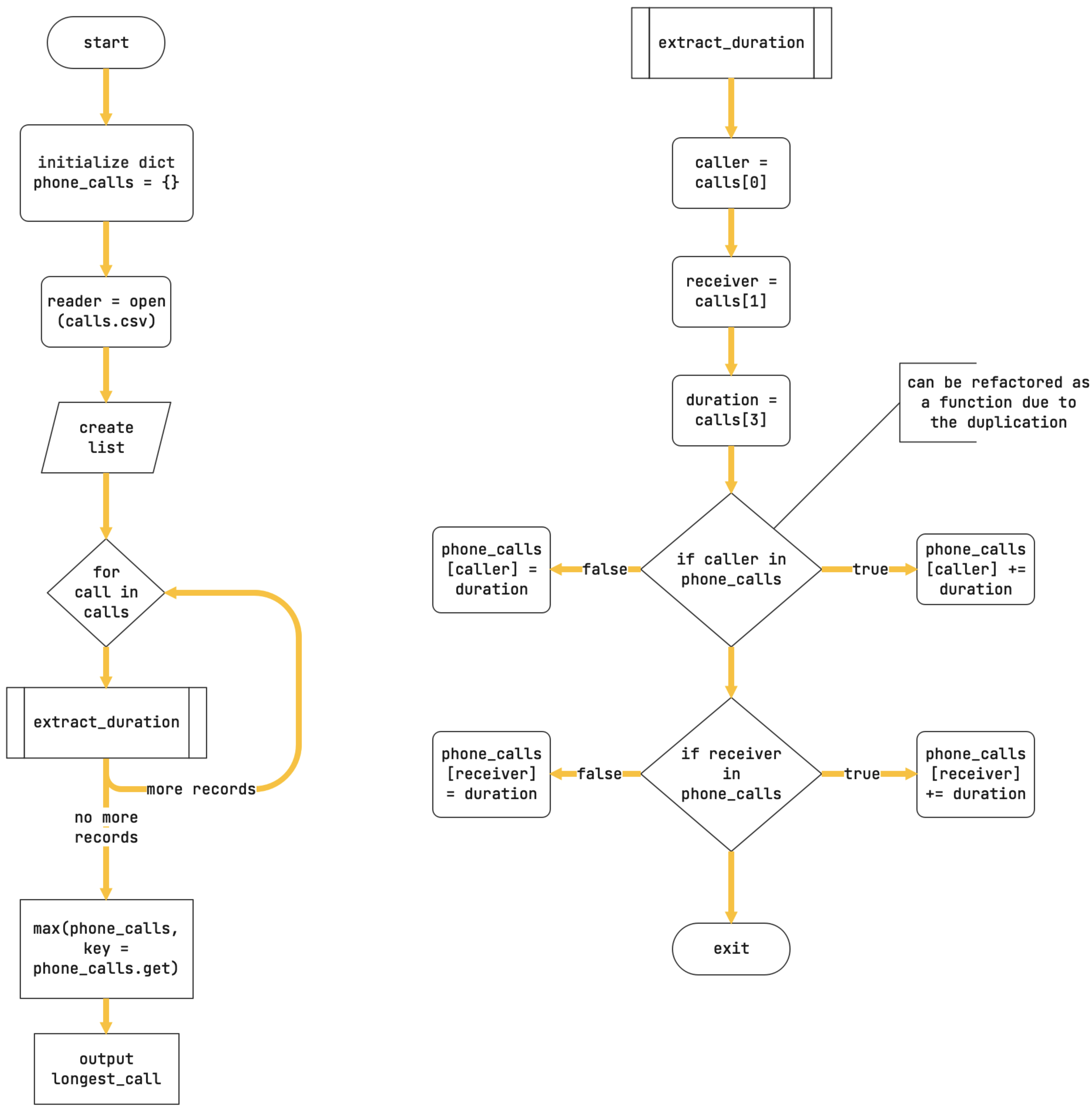
But the problem requires from me to find the phone number that spent most time on the phone during the whole month. Which means that I need to sum both incoming and outgoing call times, and then find the number with maximum amount spent on the phone.

Output should be presented in the form:
`"<telephone number> spent the longest time, <total time> seconds, on the phone during September 2016."`

Reviewer gave me a hint to use dictionary, that will hold in the same way as a set the phone number, but then I can easily add new field to store and increment the time spent with each next row from phone calls CSV file.

Flowchart

Task 2
Longest phone call



Analysis

This task looks like a simple one on the first reading, however there are tricky parts, that most probably will lead to complexities that are not very good. Starting with the list creation:

- As observed in the previous task creating list from input CSV file is **Linear operation** $O(n)$
- `extract_duration` uses lots of built-in function, but they will lead to complicated running time because of the operations performed ($O(n^2)$)
  - setting 3 variables from the input is **Constant time operation** (it is not very interesting);
  - comparison itself is a **Constant time operation**, but in my case I'm calling **dictionary** `in operator`, which compared to Hash tables should more more of a **Constant operation**. But taking a look at dictionaries in Python and time complexity for this operator I can see that worst-case runtime is $O(n)$[10-2] (strangely it acts like a normal search not key hash lookup);
  - based on the decision taken in the previous comparison I have add operation to a dictionary that happens always no mater the outcome of the previous step. Again taking a look at complexities[10-3] I can see that **dictionary Set operation** has worst case runtime **$O(n)**;
  - based on the observation I can conclude that my `extract_duration` **function** may have worst case **Quadratic time complexity:** $O(n^2)$ (based on numerous facts the runtime can be **Constant** $O(1)$, but I will take the worst-case);
- Iterating over a list is **Linear** $O(n)$
  - Inside the iteration I have a call to `extract_duration`, which I'm considering to have a **Quadratic** complexity added with the iteration operation I end up at **Cubic time complexity:** $O(n^3)$;
- Getting `max` value and key from dictionary has **Linear complexity** due to the fact that I need to iterate over all element to search and find the biggest value (considering that the dictionary is not ordered by values, and there are no further advanced implementation on-top that can speed up this find operation);

Task 2 time complexity

Taking into consideration the observations from above:
Absolutely complexity: $O(2 \cdot n + n^3)$;
Relative complexity: $O(n^3)$ (This is a running time that I should prefer to optimize and look for simpler solutions if possible, but at this stage I'm not going to look for improvements);

Task 3: Bangalore Call Statistics

At this stage I'm considering to create new function that get as input phone number and returns the type of the phone number. The best location of such function would be in a new module for re-usability, but I'm not sure still what is the Udacity Rubric, and whether or not I can pass a new module for this Project. I'm guessing that this will not be possible. So I will make a function inside the Task and unit test it against the task.

The task itself looks for certain call statistics.

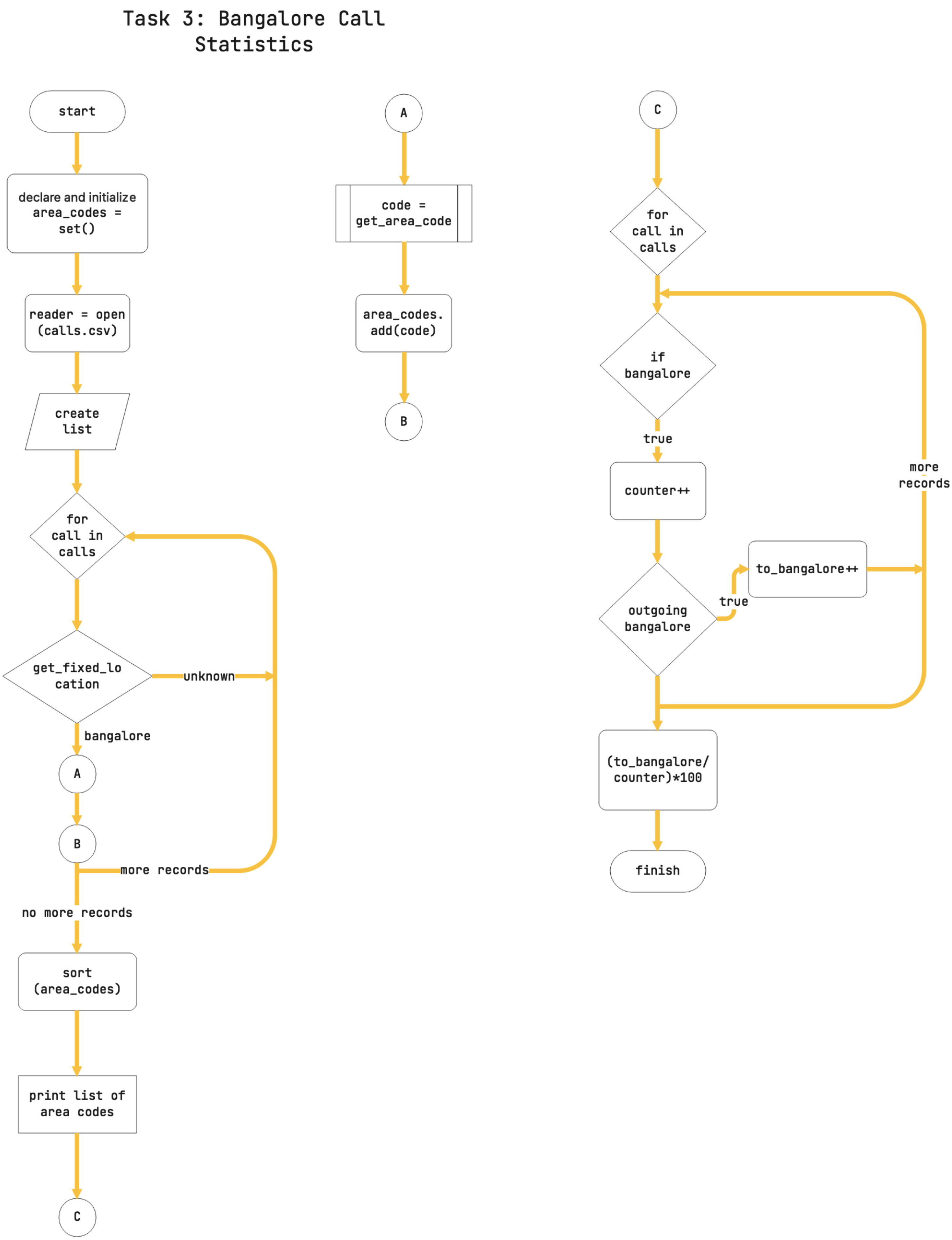
- Find all mobile phone prefixes and and area codes, for Bangalore numbers;
- The list of phone codes need to be printed out in Lexicographic Order [11] [12];
- How many calls from Bangalore are made to Bangalore (return as percent);

Helper functions that I will need for the current task:

- **get_telephone_type** - get telephone number as input, and returns: **fixed**, **mobile**, **telemarketers**;
- **get_area_code** - takes as input phone number, returns area code such as **(080)**;
- **get_fixed_location** - takes as input **fixed** type of telephone number, and returns location ex. **Bangalore**, for all other numbers (including fixed lines outside of Bangalore) return unknown of the moment;

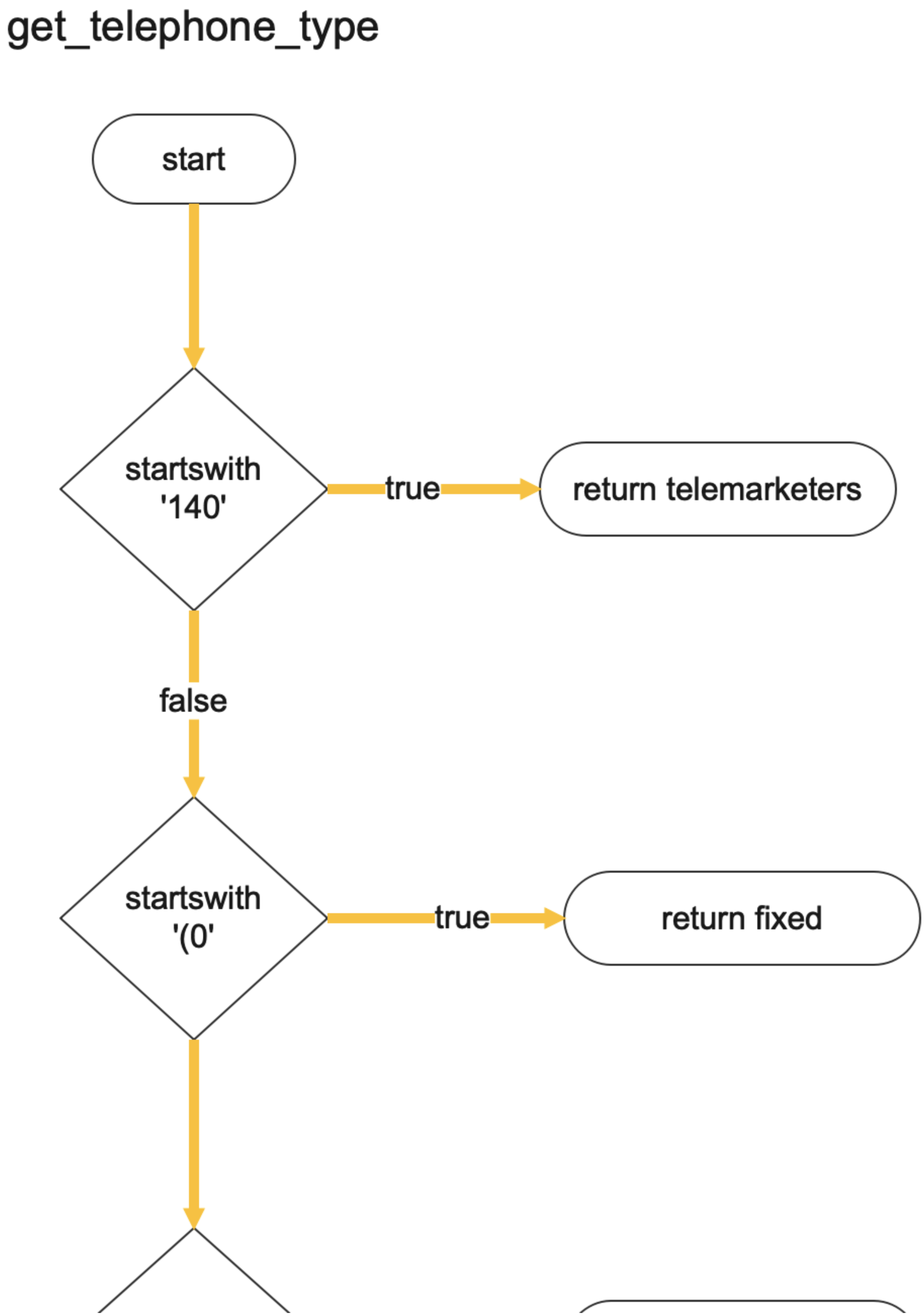
Flowchart

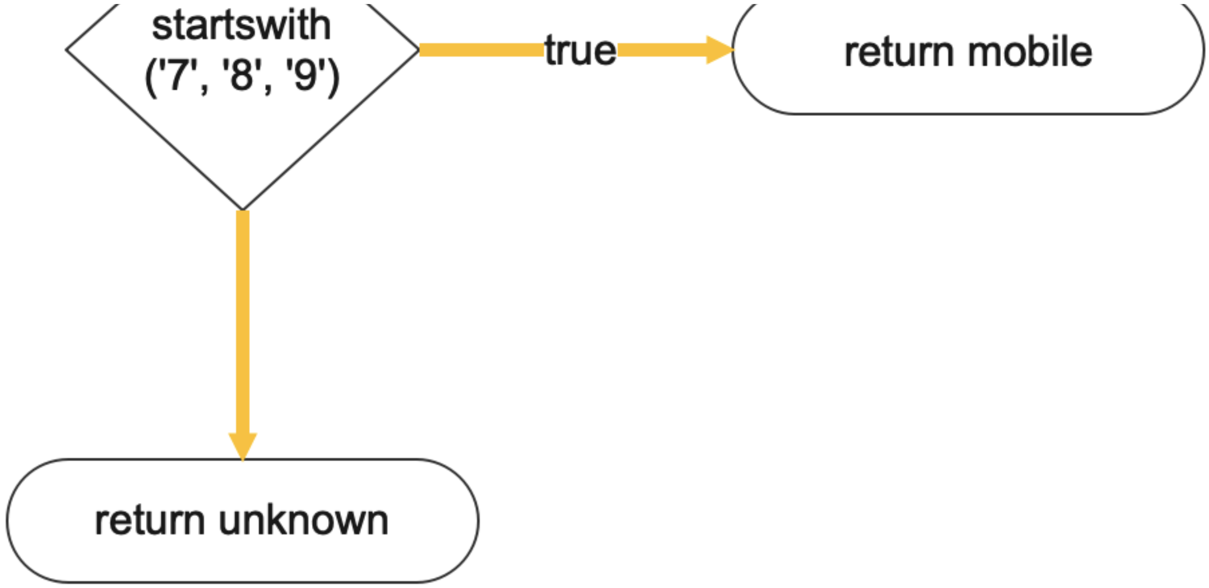
Task 3


Task 3: Bangalore Call Statistics
start
declare and initialize area_codes = set()
reader = open (calls.csv)
create list
for call in calls
get_fixed_location
unknown
bangalore
A
B
more records
no more records
sort (area_codes)
print list of area codes
C
A
code = get_area_code
area_codes.add(code)
B
C
for call in calls
if bangalore
true
counter++
outgoing bangalore
true
to_bangalore++
more records
(to_bangalore/ counter)*100
finish

Helper functions

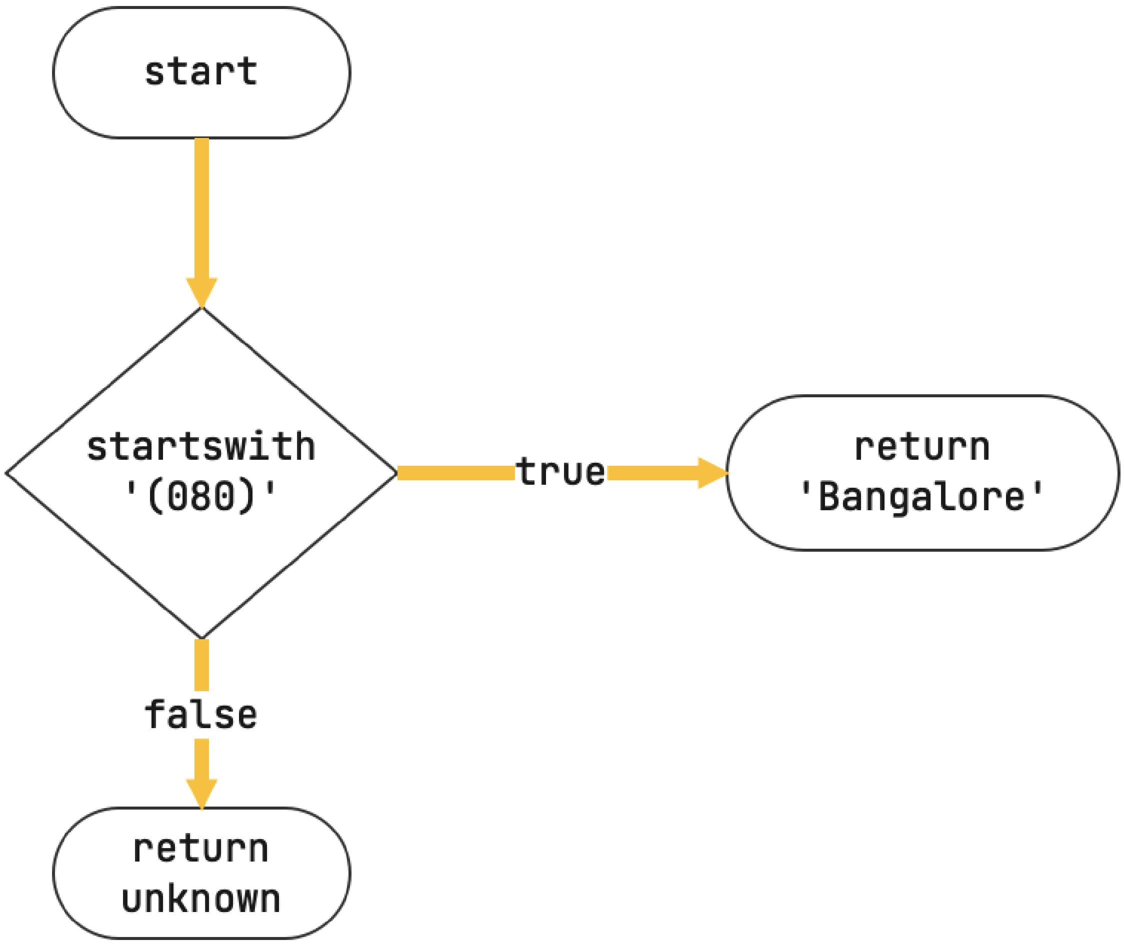
`get_telephone_type`



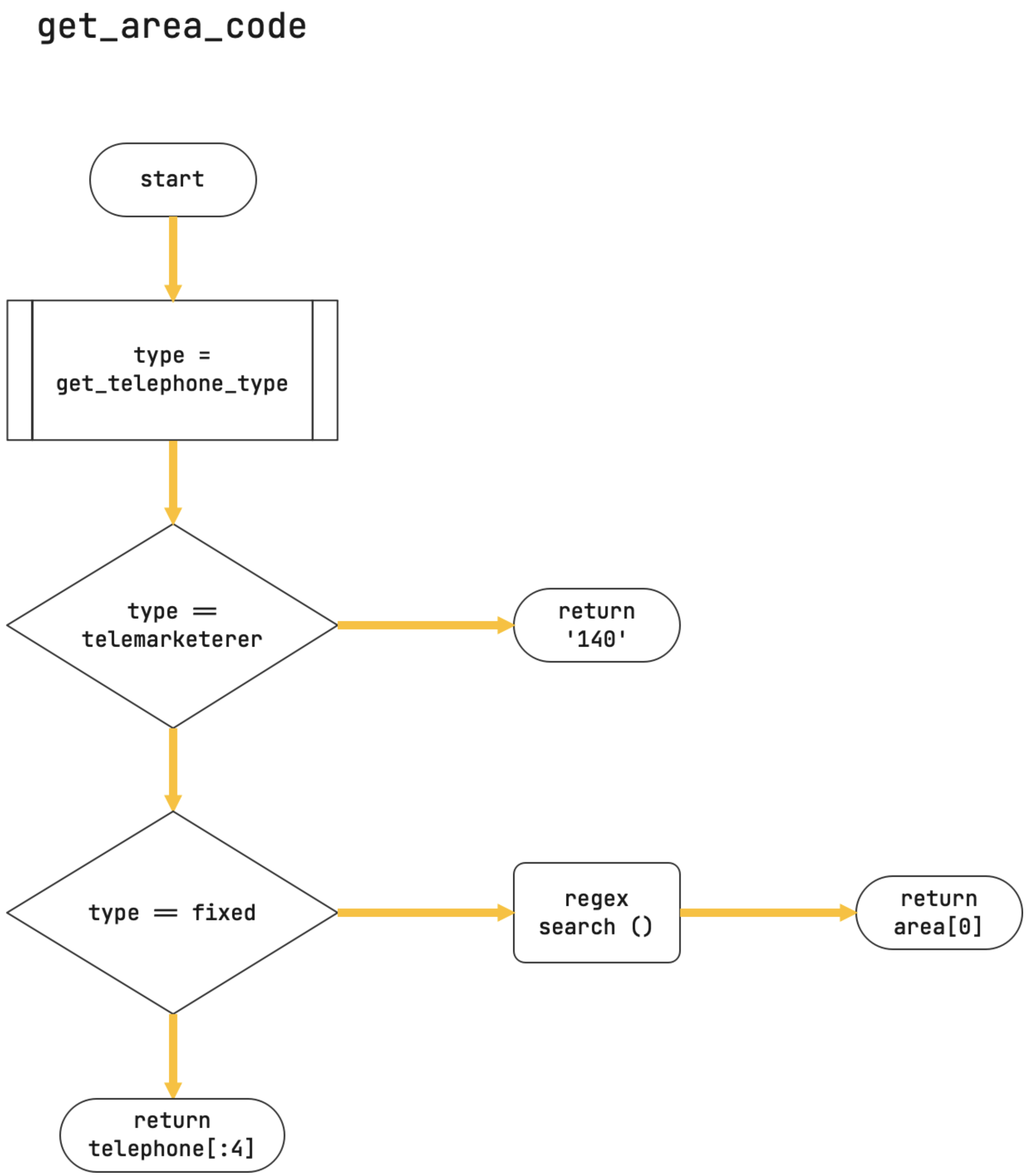

startswith
('7', '8', '9')
true
return mobile
return unknown

`get_fixed_location`

get_fixed_location



`get_area_code`



Analysis

Going from backward of the solution, I had to take a decision whether to extend in iterative manner the first solution, to produce the second one. However, as we are looking for simple mechanical solutions at first, and as we are looking for **Relative complexity** not **Absolute** I decided to go with two similar solution one after another. The overhead of going twice element by element in the Calls list is not so big (compared to relative complexity). Also at this stage I'm not seeing any restrictions on performance or memory.

Now getting into analyzing the **Helper functions**.

Helper function to find what telephone type we are dealing with: **get_telephone_type**:

- In this helper function we do have two comparisons with **str.startswith**. I did not manage to find how the function is implemented, but for the sake of this exercise I can guess that it compares symbol by symbol. In normal circumstances this translates to **Linear complexity** - $O(n)$, but in my scenarios I'm comparing fixed width strings every time, so I will consider the operation as a **Constant**;
- For mobile phones I'm performing search against tuple, which will translate in my case as a 3 times **Constant** operation, which is still **Constant**. There is alternative way to solve the problem by looking at the input as an array/list, and comparing the first character against either 7 or 8 or 9. However, due to the Pythonic way of solving problems I will prefer not the hacky brute way but the standard for the community, guessing that experienced developers will look up for `startswith` rather than hacky array solution;

Helper function that finds the area code **get_area_code**:

- First we have a **constant operation** call to **get_telephone_type**;
- Then we have couple comparisons which are also **Constant time operations**;
- For the fixed line numbers, as their size can vary I decided for first implementation go to with regex search **Linear operation**, I'm not trying to compare the whole string only bits of the string, but as I'm not looking specifically for the start of a string (even the I know it should be there), I will consider the operation as linear. $O(n)$ (**This may be a good candidate for future improvements**)

And finally we are getting into get fixed location function, which at the moment is used only to match Bangalore call codes.

- I will consider this helper function to be of **Constant time complexity -** $O(1)$

Now, I can focus on calculating the whole **Task 3 complexity**.

Solution A: On top of the previously calculated time complexity (in Tasks 0, 1 and 2) of $O(n)$ we have:

- adding item to **set** - **Linear time complexity** $O(n)$
- get_fixed_location is with **Constant time complexity**
- get_area_code has **Linear time complexity**

- **sorted (Timsort)** function call to get **lexicographical order of the set**, which has **Linearithmic** time according to the documentations $O(n \cdot log\, n)$ [5-1]

Calculating the final time complexity:

- Having an for loop that makes call to another **Linear time complexity** function translates to **Quadratic -** $O(n^2)$
- And comparing **Quadratic time** against **Linearithmic time**, I conclude[9-1] the final **Relative time complexity is:** $O(n^2)$

Solution B:

- as in looking for the percentage of calls from Bangalore to Bangalore I'm comparing twice against **get_fixed_location** the time complexity is still $O(n)$

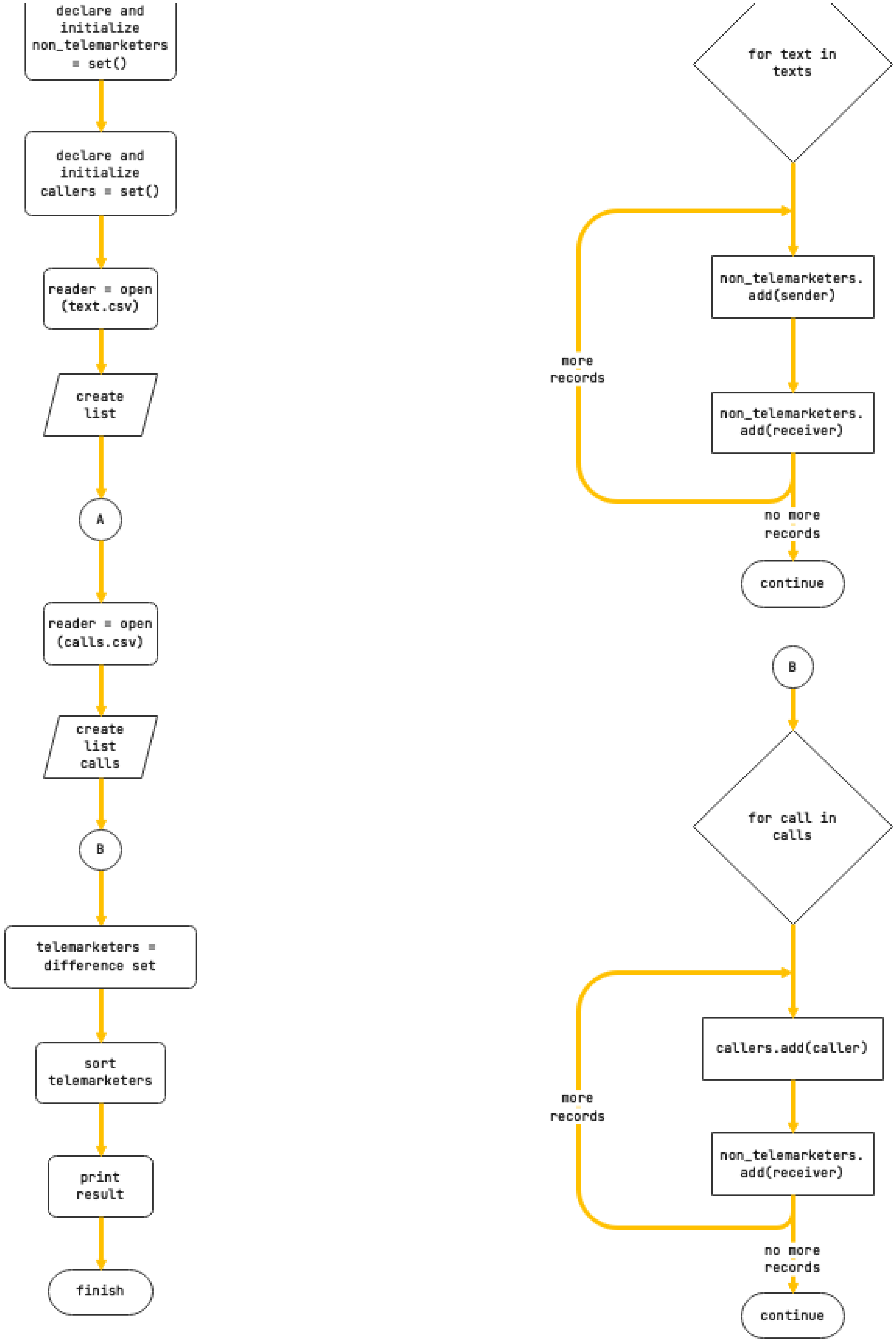
Task 3 time complexity

The final conclusion for the whole Task, is that the **relative time complexity** of my solution is: $O(n^2)$. (Of course if I was looking for performance optimizations this time is not good enough)

Task 4: Looking for Marketing telephones

I can reuse all helper functions from the previous task in order to finish faster Task 4. The job for me here is to find all telemarketer phones in both `texts.csv` and `calls.csv`. From the phone that I find in those lists then I need to create a set of possible phone numbers. The possible phone numbers I need to check against both files again but this time I need to search in the second columns, they **should not receive any incoming calls** or **should not receive any text messages**.

Flowchart



Analysis

Most of the code is reused from the previous Task (although the fact that is copy and paste). So I will analyze only bits and parts of the code.

- looping through both calls and texts: I have `for loop` for every element in the input list, which translated to $O(n)$. For every element I have **add operation** to a **set** which translates to another $O(n)$[10-4]. Time complexity for the iteration is: $O(n^2)$ (because the two **add operations** are one after another - they are not nested, I'm not calculating the complexity as **Cubic**);
- Looking for difference between two sets translates to $O(len(n))$[10-5], where n is the size of the set(`callers`). Again there is a slight chance this to produce a running complexity time **$O(n)**, nevertheless I already have slower part of my solution;

Task 4 time complexity

As we are having three operations to compare, two of which are with **Quadratic time** and one is with **Linear** the slower of them is **Quadratic**[9-2]. **Absolute complexity**: $O(2 \cdot n^2 + n)$;

For the whole module, time complexity is: **Quadratic Time -** $O(n^2)$

Bibliography and Resources

---

1. *Space Complexity Examples*. (n.d.). Udacity. Retrieved April 27, 2021, from https://classroom.udacity.com/nanodegrees/nd256/parts/f74fc064-524b-4ee8-8fb1-570b3c31a993/modules/3675395a-5de0-4f74-a4c1-eadafd6be9f9/lessons/b5ed8170-8fce-463a-aefc-64272cb3852e/concepts/09af2846-2d8a-4688-870e-89fb57e7c74c↩
2. JetBrains. (2021). *PyCharm Professional* (2021.1.1). JetBrains. https://www.jetbrains.com/pycharm/↩
3. Python Software Foundation. (2020). *Python 3.9.0* (3.9.0). Python Software Foundation. https://www.python.org/downloads/release/python-390/↩
4. fpedregosa. (2020). *memory-profiler 0.58.0* (0.58.0). Open Source. https://pypi.org/project/memory-profiler/↩
5. *Timsort Wikipedia*. (n.d.). Wikimedia Foundation. Retrieved April 27, 2021, from https://en.wikipedia.org/wiki/Timsort↩↩
6. *Samplesort Wikipedia*. (n.d.). Wikimedia Foundation. Retrieved April 27, 2021, from https://en.wikipedia.org/wiki/Samplesort↩
7. Python Software Foundation. (n.d.). *Sorting How To*. Python.Org. Retrieved April 27, 2021, from https://docs.python.org/3/howto/sorting.html↩
8. Python Software Foundation. (n.d.). *Comparison with Python's Samplesort Hybrid*. Python.Org. Retrieved April 27, 2021, from https://svn.python.org/projects/python/trunk/Objects/listsort.txt↩

9. *Know Thy Complexities! (Cheatsheet)*. (2020). https://www.bigocheatsheet.com/ ↩↩↩
10. *Python Wiki TimeComplexity*. (n.d.). https://wiki.python.org/moin/TimeComplexity↩↩↩↩↩↩
11. *Lexicographic Order*. (n.d.). Wikimedia Foundation. Retrieved May 2, 2021, from https://en.wikipedia.org/wiki/Lexicographic_order↩
12. Samual Sam. (2018). *Sort the words in lexicographical order in Python*. Tutorials Point. https://www.tutorialspoint.com/Sort-the-words-in-lexicographical-order-in-Python↩