

# 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:

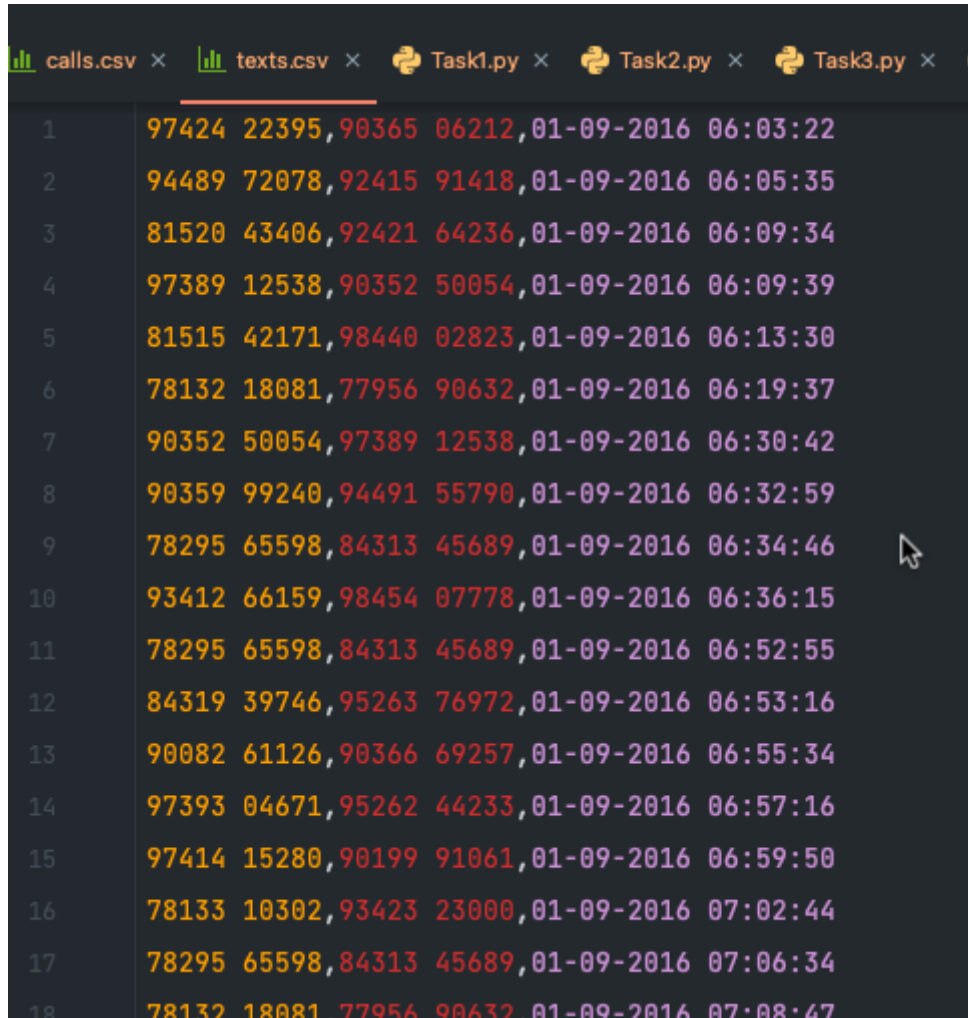
- **Fixed lines** - start with an area code enclosed in brackets. The area codes vary in length but always begin with 0. Example: "(022)40840621".
- **Mobile numbers** - have no parentheses, but have a space in the middle of the number to help readability. The mobile code of a mobile number is its first four digits and they always start with 7, 8 or 9. Example: "93412 66159".

- **Telemarketers** - numbers have **no parentheses or space**, but **start with the code 140**. Example: "1402316533".

## 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)



	calls.csv	texts.csv	Task1.py	Task2.py	Task3.py
1	97424	22395,90365	06212,01-09-2016	06:03:22	
2	94489	72078,92415	91418,01-09-2016	06:05:35	
3	81520	43406,92421	64236,01-09-2016	06:09:34	
4	97389	12538,90352	50054,01-09-2016	06:09:39	
5	81515	42171,98440	02823,01-09-2016	06:13:30	
6	78132	18081,77956	90632,01-09-2016	06:19:37	
7	90352	50054,97389	12538,01-09-2016	06:30:42	
8	90359	99240,94491	55790,01-09-2016	06:32:59	
9	78295	65598,84313	45689,01-09-2016	06:34:46	
10	93412	66159,98454	07778,01-09-2016	06:36:15	
11	78295	65598,84313	45689,01-09-2016	06:52:55	
12	84319	39746,95263	76972,01-09-2016	06:53:16	
13	90082	61126,90366	69257,01-09-2016	06:55:34	
14	97393	04671,95262	44233,01-09-2016	06:57:16	
15	97414	15280,90199	91061,01-09-2016	06:59:50	
16	78133	10302,93423	23000,01-09-2016	07:02:44	
17	78295	65598,84313	45689,01-09-2016	07:06:34	
18	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)

```
calls.csv x texts.csv x Task1.py x Task2.py x Task3.py x Task4.py x is_leap_y

1 78130 00821,98453 94494,01-09-2016 06:01:12,186
2 78298 91466,(022)28952819,01-09-2016 06:01:59,2093
3 97424 22395,(022)47410783,01-09-2016 06:03:51,1975
4 93427 40118,(080)33118033,01-09-2016 06:11:23,1156
5 90087 42537,(080)35121497,01-09-2016 06:17:26,573
6 97427 87999,(04344)322628,01-09-2016 06:19:28,2751
7 (080)45291968,90365 06212,01-09-2016 06:30:36,9
8 78132 18081,77956 90632,01-09-2016 06:39:03,3043
9 98453 46196,94005 06213,01-09-2016 06:40:20,2457
10 78290 99865,89071 31755,01-09-2016 06:46:56,9
11 (04344)228249,(080)43901222,01-09-2016 06:50:04,2329
12 (080)62164823,74066 93594,01-09-2016 06:52:07,300
13 (0821)6141380,90366 69257,01-09-2016 06:54:44,2147
14 98446 66723,83019 53227,01-09-2016 06:56:16,129
15 90088 09624,93434 31551,01-09-2016 06:57:44,133
16 93427 56500,98447 12671,01-09-2016 07:03:45,29
17 (040)26738737,90194 00845,01-09-2016 07:12:39,94
18 (080)67362492,(04344)316423,01-09-2016 07:24:45,2258
19 90192 87313,(080)33251027,01-09-2016 07:28:01,110
```

## 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:

$$O(2 \cdot 13 \text{ Bytes} + 19 \text{ Bytes} + 4 \text{ Bytes}) = 49 \text{ Bytes}$$

## 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):

$$O(2 \cdot 13 \text{ Bytes} + 19 \text{ Bytes}) = 45 \text{ Bytes}$$

## Storing Texts and Phone Calls

The final calculation for Space Complexity is:

**texts[]** list:  $O(n \cdot 45 \text{ Bytes})$

**calls[]** list:  $O(n \cdot 49 \text{ Bytes})$

## Live Experiment

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

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

Rough calculations for texts list: it appears that most of the records are only mobile numbers (11 chars):

$$11 \text{ Bytes} \cdot 2 + 19 \text{ Bytes} = 41 \text{ Bytes} \cdot 9072 \text{ records} = 371952 \text{ Bytes or } 0.354721 \text{ Megabytes}$$

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, t
12	16.1 MiB	0.0 MiB	1	with open('texts.csv', 'r') as f:
13	16.1 MiB	0.0 MiB	1	reader = csv.reader(f)
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 = fi

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<sup>[5]</sup> and Samplesort<sup>[6]</sup>. I can read further how to use these Python Built-in function on Python documentation<sup>[7]</sup>. In Python documentation I can also find more information and analyses for Time complexity<sup>[8]</sup> on the sorting algorithms used.

General Bio O Cheetsheet can be downloaded from [Know Thy Complexities!](#)<sup>[9]</sup>.

## Task 0: Printing First and Last records

First task of the project, requires from me to:

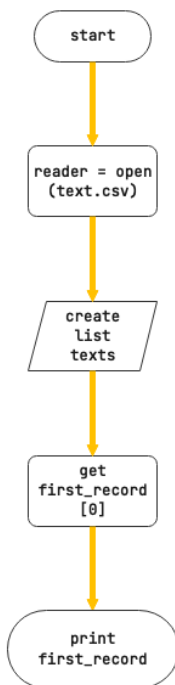
- print first record from texts file;
- print last record from calls file;

**Example of the output:**

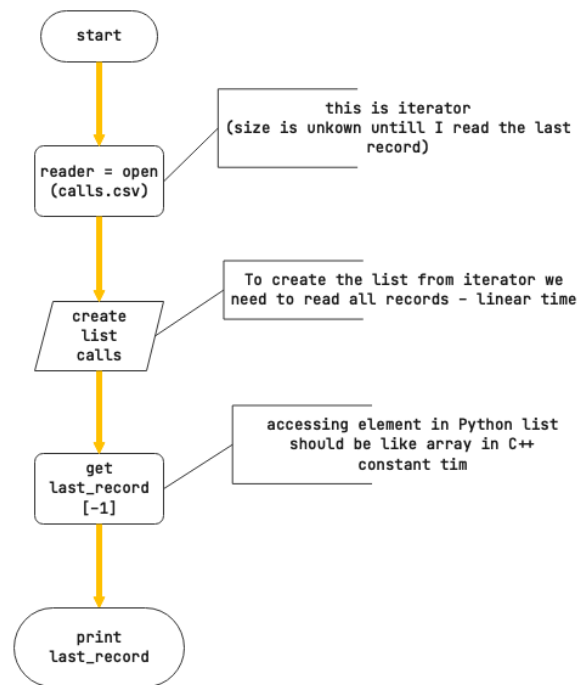
"First record of texts, <incoming number> texts <answering number> at  
"Last record of calls, <incoming number> calls <answering number> at

## Flowchart of the solution

### Task 0 Print First text



### Task 0 Print Last call



## 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)**<sup>[10]</sup> - Creating text record in Python list from iterator, for every input record:  $O(n)$  (**linear time**);
- **(Operation: Get Item)**<sup>[10-1]</sup> - 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 **texts.csv**. Columns used to count unique numbers:

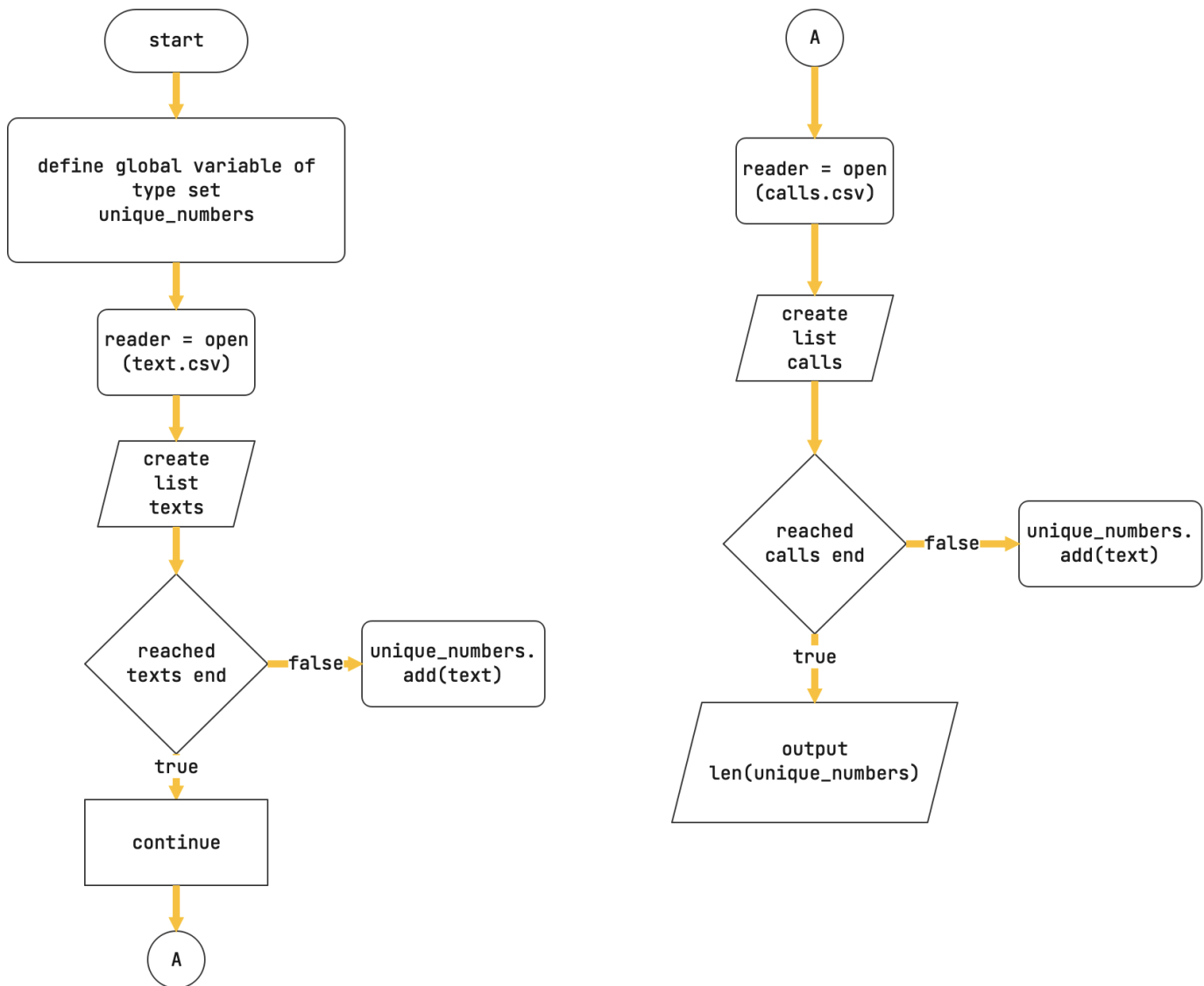
- sending telephone number (texts.csv)
- receiving telephone number (texts.csv)
- calling telephone number (calls.csv)
- receiving telephone number (calls.csv)

## Flowchart of the solution



## Task 1

### Unique Numbers



### 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: Longest Phone Call

In this task I need to find the longest Phone call that is stored in `calls.csv`, and output the following information:

"<telephone number> spent the longest time, <total time> seconds, on the phone during September 2016."

I can think of couple solutions:

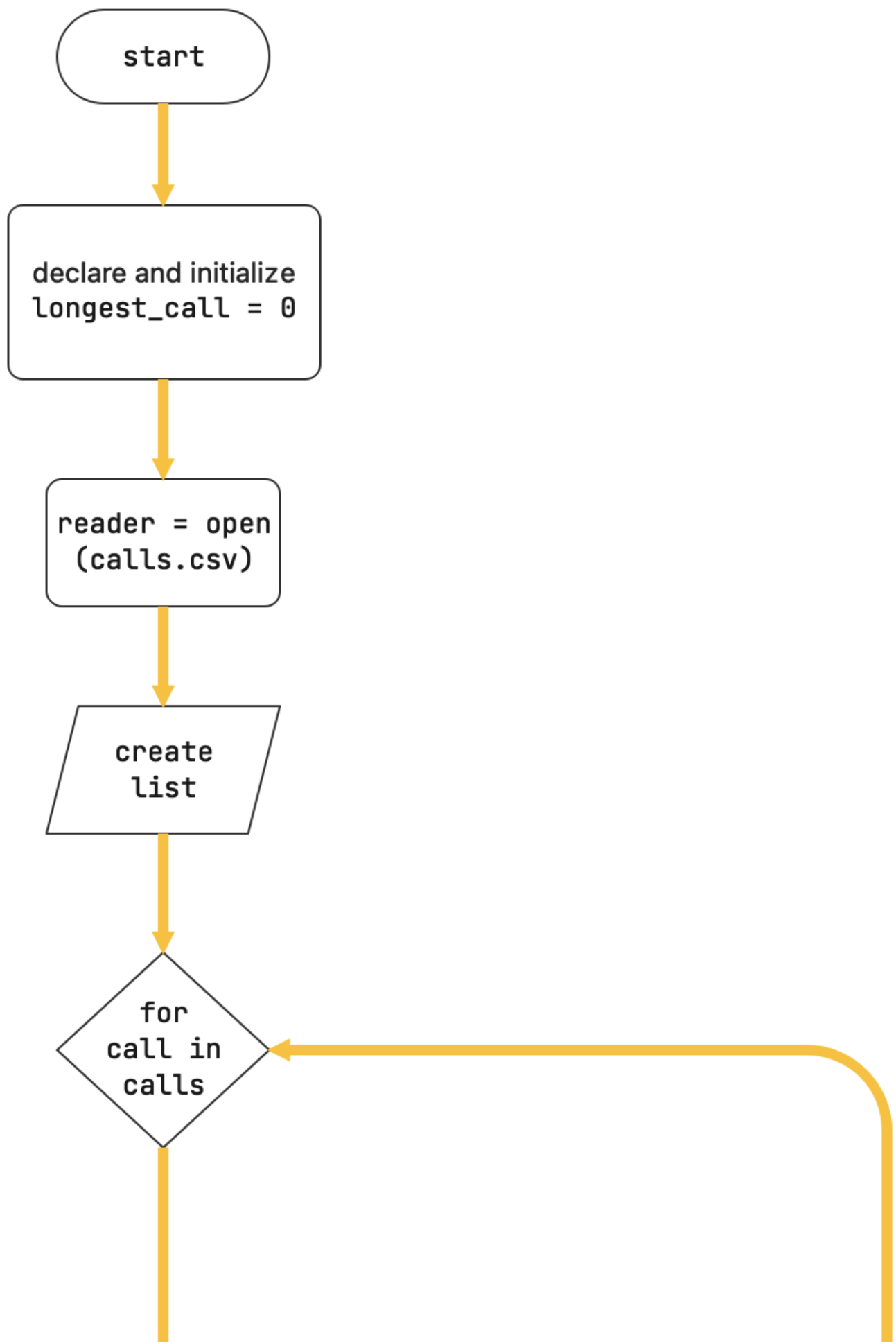
- Go for each record and compare the time duration;
- Sort the list by duration field and pick first/last record depending on the sorting criteria;

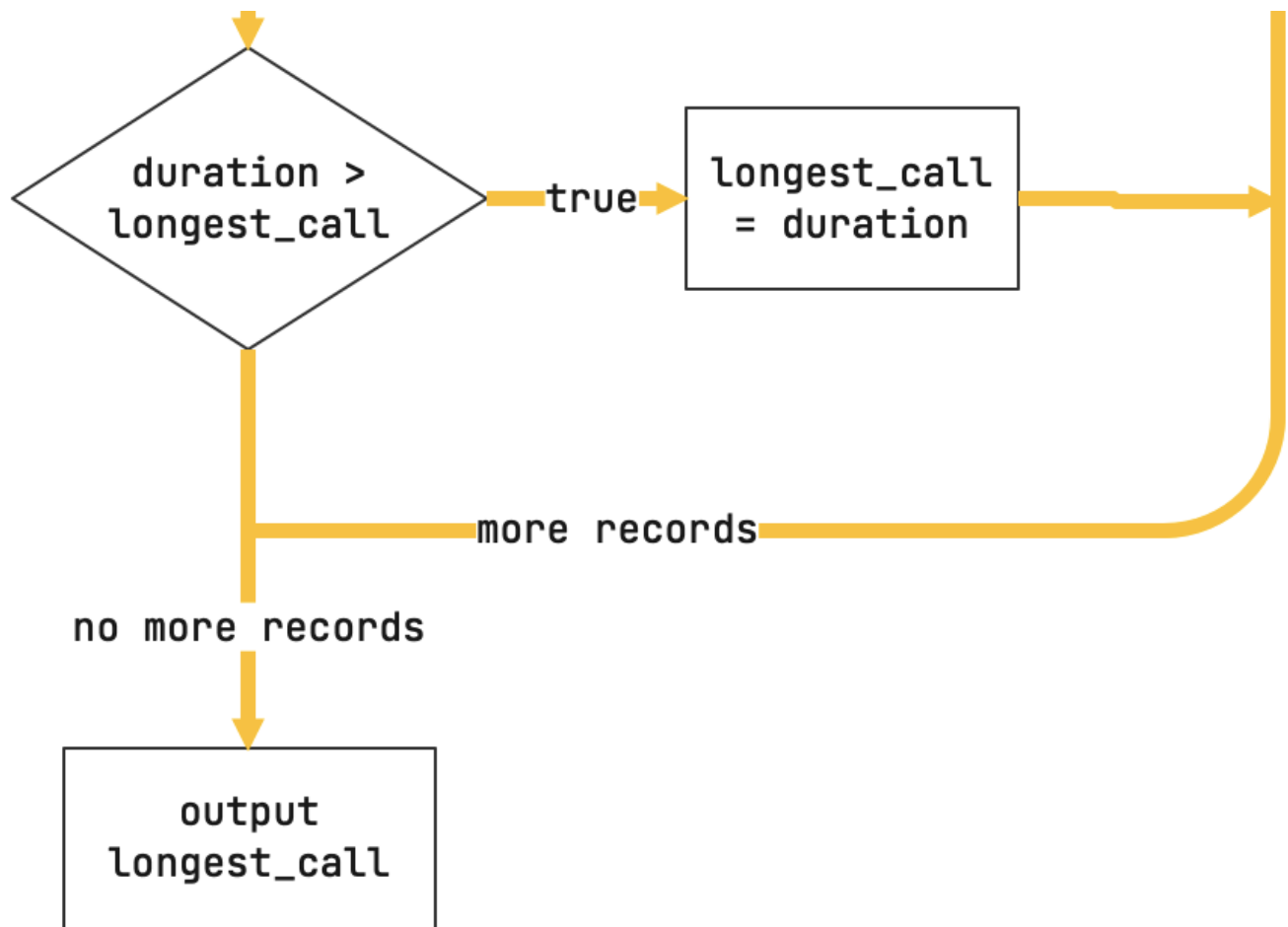
In the previous task [Task 1 Counting unique telephone numbers](#) I already have performed going record by record to search for unique numbers. This will easily give me again time complexity of  $O(2 \cdot n)$ , comparing with sorting methods that are outlined in the project `Timsort` and `Samplesort` I will get the same time complexity if not worse.

Based on that I will go with the linear search in the whole list. Also, I will not consider which record to choose if two or more call records are with the same duration. I will simply get the first record found.

## Flowchart

# Task 2 Longest phone call





## Analysis

Most of the operation that are happening in this task have been already discussed in the previous tasks. The difference here is in the for loop:

- Creating list from input csv file is **Linear operation**  $O(n)$
- Converting duration from string to integer is **Constant operation**
- Iterating over a list is **Linear**  $O(n)$
- If condition is by itself does have  $O(1)$  complexity, but it depends a lot on what operation is performed. In my case the operation of setting variable is constant time. So the whole operation will have **Constant time complexity**.

## #### Task 2 time complexity

**Absolutely complexity:**  $O(2 \cdot n)$

**Relative complexity:**  $O(n)$

## *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 area codes, for Bangalore numbers;
- The list of phone codes need to be printed out in [Lexicographic Order](#) <sup>[11]</sup> <sup>[12]</sup>;
- How many calls from Bangalore are made to Bangalore (return as percent);

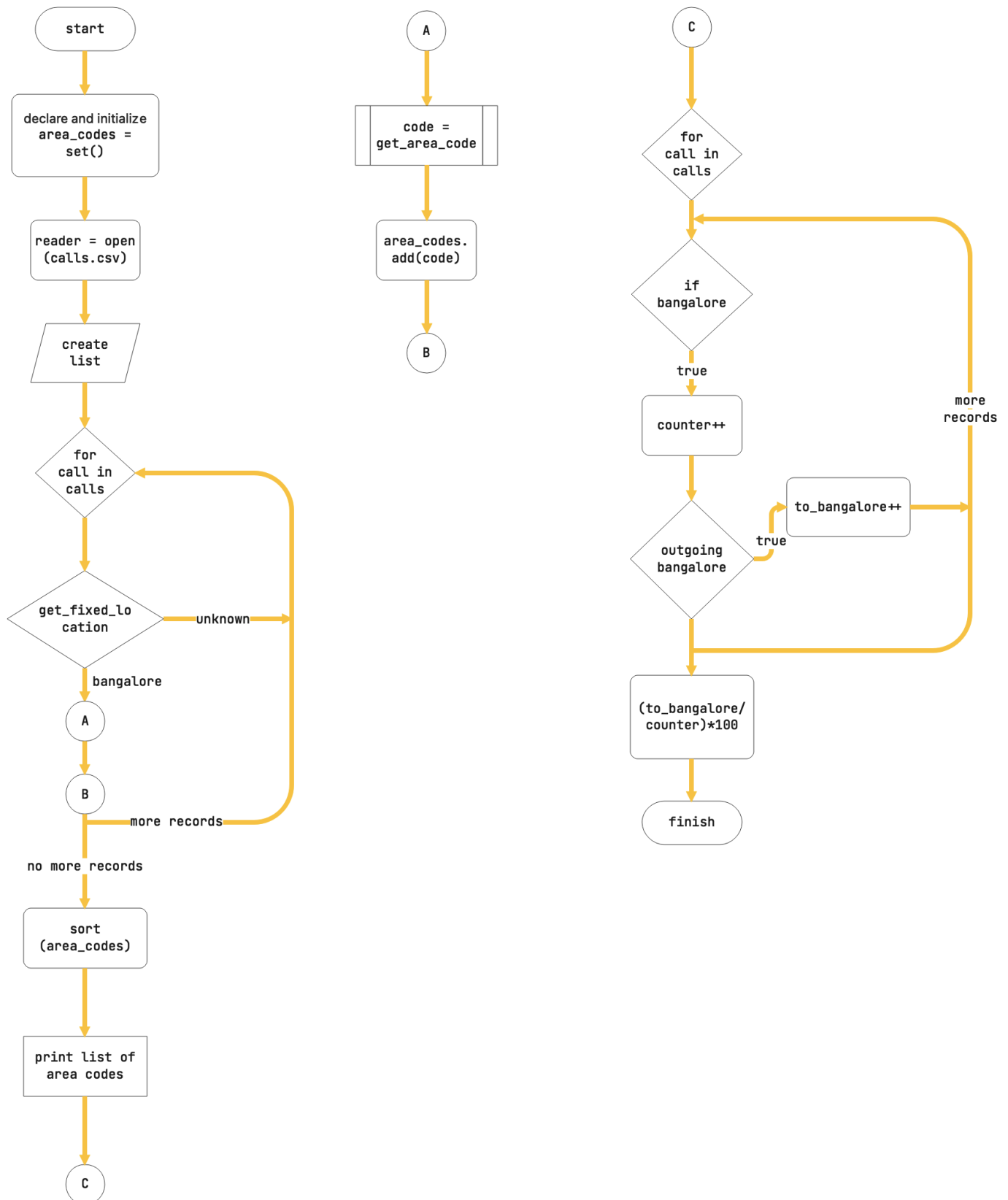
Helper functions that I will need for the current task:

- **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;

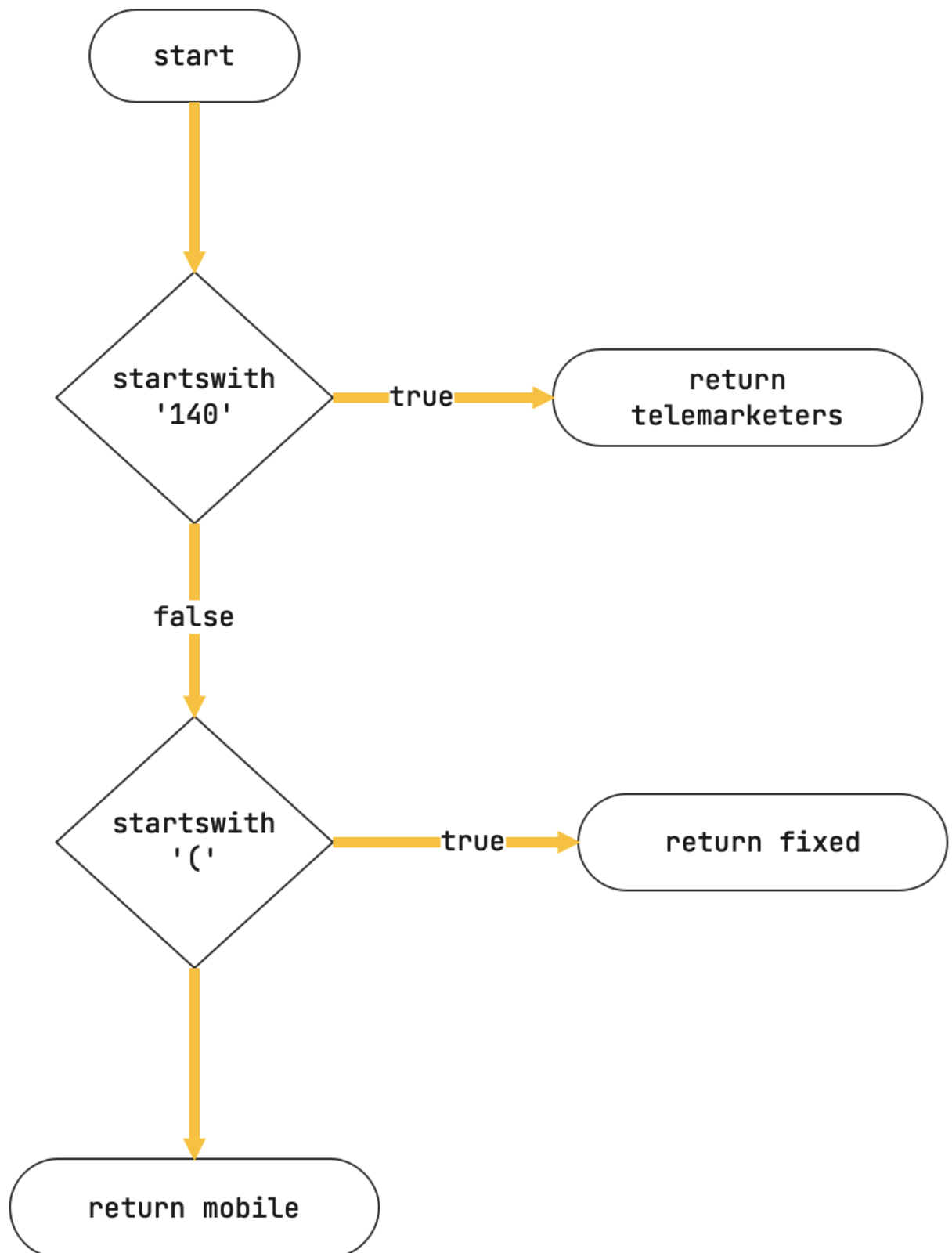
## Flowchart

## Task 3

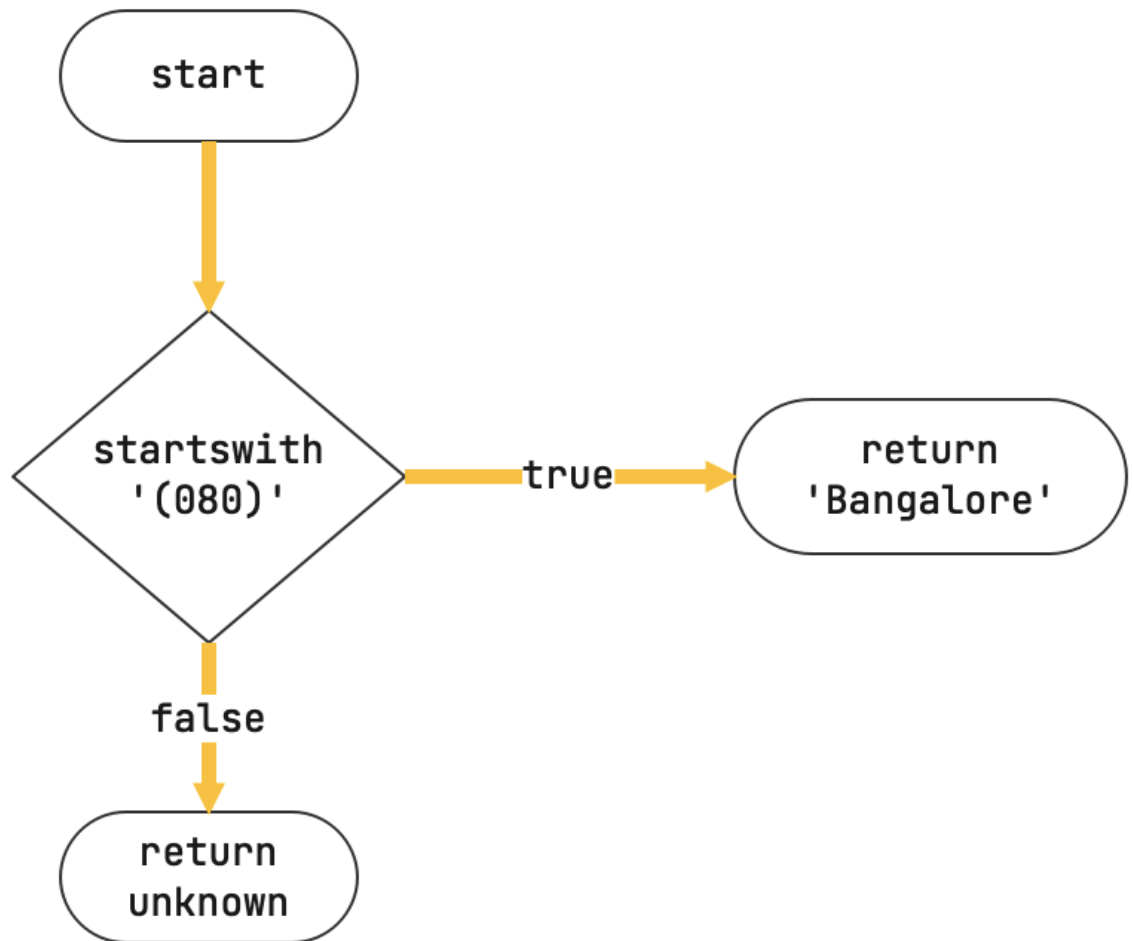
### Task 3: Bangalore Call Statistics



## get\_telephone\_type

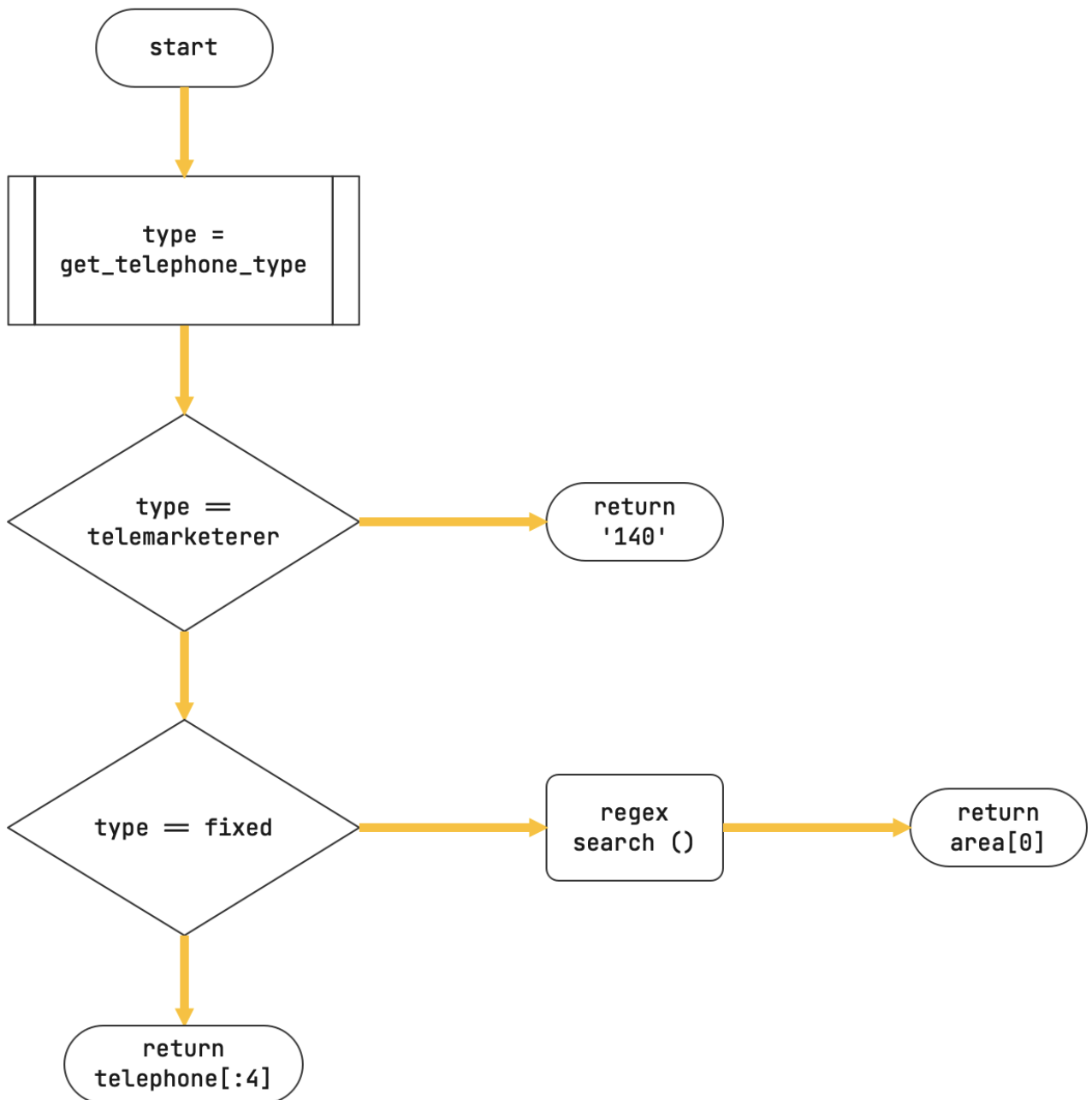


## get\_fixed\_location





## 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**.

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  $O(n \cdot \log n)$  <sup>[5-1]</sup>

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<sup>[9-1]</sup> 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**

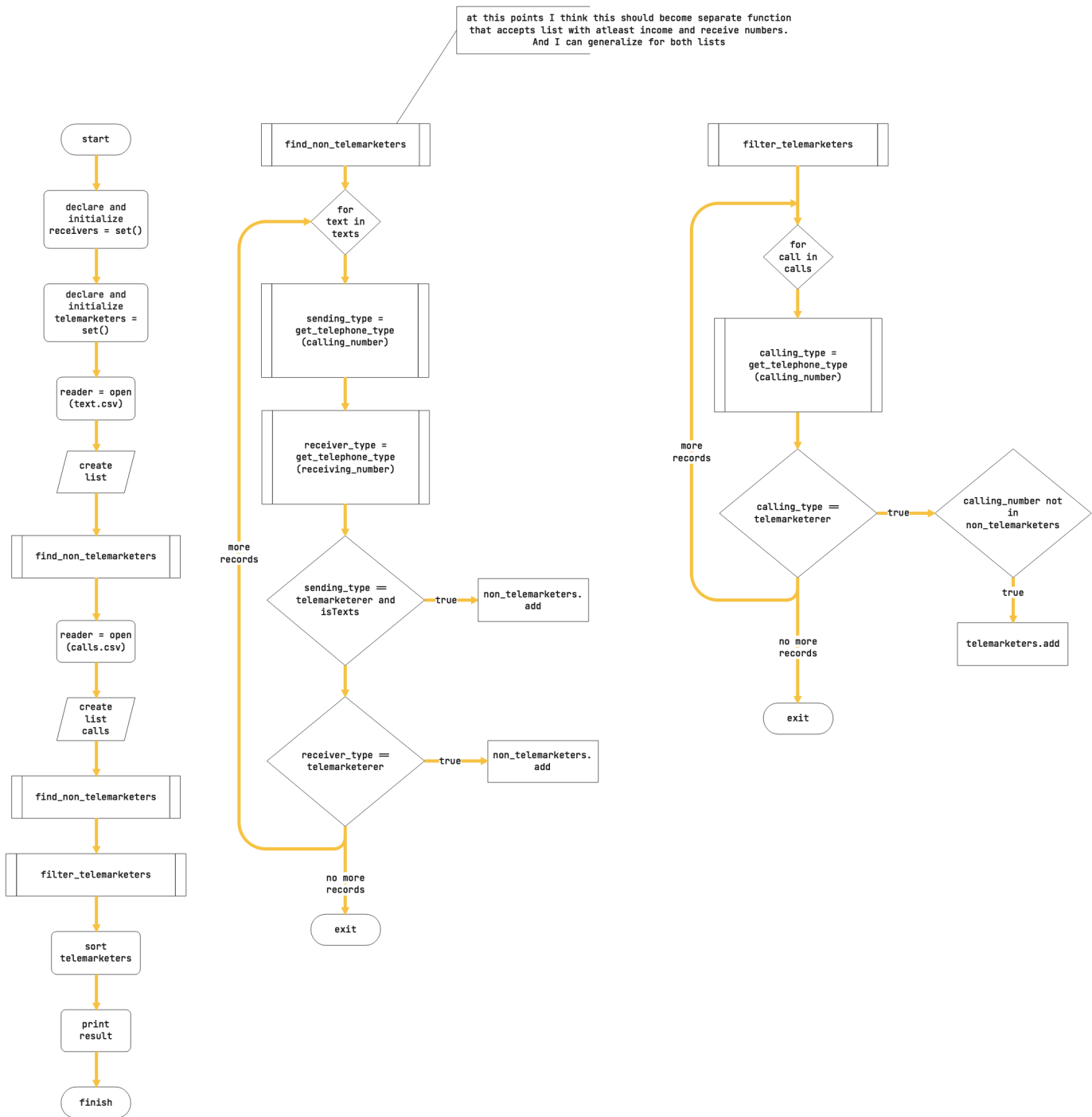
The final conclusion for the whole Task, is that the **relative time complexity** of my solution is:  $O(n^2)$ . (Of course if I was looking for performance optimizations this time is not good enough)

#### *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**

## Task 4: Looking for Marketing telephones



## 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.

- starting from the end, **sorted (Timsort)** has **Linearithmic** time according to the documentations  $O(n \cdot \log n)$ ;

- going to the new function from this module: `find_non_telemarketers` I have for loop for every element in the input list, which translated to  $O(n)$ . Then for every element that is `telemarketer` I have **add operation** to a **set** which translates to another  $O(n)$ . Even though not every element will be `telemarketer` there is a possibility in which all the numbers are `telemarketers`. Time complexity for the function is:  $O(n^2)$ ;
- for the same reason as above `filter_telemarketers` (the **set**) I conclude that the time complexity is again:  $O(n^2)$  (even though the chance is smaller than the previous to hit all records);

## Task 4 time complexity

As we are having three operations to compare, two of which are with **Quadratic time** and one is with **Linearithmic (Log-linear Time)** the slower of them is **Quadratic**<sup>[9-2]</sup>.

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](https://en.wikipedia.org/wiki/Lexicographic_order)↩
12. Samuel Sam. (2018). *Sort the words in lexicographical order in Python*. Tutorials Point. <https://www.tutorialspoint.com/Sort-the-words-in-lexicographical-order-in-Python>↩