**COM421**

**AE1 Report**

*By: Ismail Kefokeris*

1. **Justification of data structures and algorithms used**

**Data Structure: Dictionaries**

The data structure I had chosen for the task was the dictionary data structure due to its ability to simplistically store multiple values under one item (key, the key will be a modified version of the name with an addition of the number of keys with the same name, if there are none it will by default set it as value 0) in this case I would use it to store the name, establishment, description, and address into one single key, this will allow me to call the key and make use of the data values stored inside. Using a dictionary was the correct choice due it not consuming a large amount of memory but instead starts with a small chunk based on the number of initial slots it has and only when the dictionary starts to grow will it take additional memory, additionally the keys and values are not stored within the dictionary but instead are a reference to the position in memory which contains them. Dictionaries will also give me a huge performance boost in the program when it grows, for instance, if the program were to have hundreds of different POIs, we would be able to locate the POI desired extremely easily by referencing the specific key.

Task one entailed that I needed to allow the user to add new points of interest which included 4 values (name, establishment, description, and address). The program will take the “name” value and modifying it adding a unique number at the end of the name depending on the amount of previously existing keys similar to its name making it unique and setting it as the key of the dictionary and inside this, the other values will be stored including its original name.

Task two essentially states that I am required to allow the user to search for a specific POI and return the results to the user. The program will search for the name that the user would have inserted and go through all the keys (using a for loop) noting down the keys which contain the name (even if partially), this is done to allow the user to distinguish from other POI that may have a similar name. The method will present all the results found with their respective post-codes and will prompt the user to be more specific allowing the user to find the exact POI desired.

Task four states that the user must be able to search for a POI by name, the program allows the user to enter the exact name if known, this would be the value that is being used as the key (the name modified by adding a number on the end making it unique). This is easily because all POIs are stored inside a dictionary allowing me to simply check if the inputted name exists in the dictionary. The complexity for this is O(1) because it is constant time meaning it performance isn’t dependant on the size of the dictionary. For task four I had no need of a searching algorithm such as Binary search because of the data type chosen, dictionaries eliminates the need to use a searching algorithm if the key is known and would perform faster in most cases than a Binary search which has a time complexity of O(log n).

**Sorting Algorithm: Quick Sort**

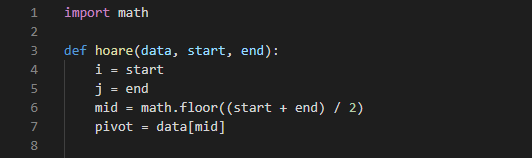
Task three specifies that I must display all POIs but they must all be sorted by name (alphabetically), to do this I had to decide between two sorting algorithms Quick Sort utilises what’s known as a pivot to choose the half point and divides the list into mini lists, this is performed recursively until the list is sorted making it an (O(n log(n))), in the worst-case scenario Quick Sort can be an (O(n^2)) and Bubble Sort which focuses on moving one value at a time (the larger value) meaning the algorithm will only amass to an (O(n^2)). Quick Sort was the obvious option in this scenario due to it being faster (on average) compared to Bubble Sort. Even in the worst-case scenarios, it will perform just as fast as a Bubble Sort algorithm.

Task five states that I must also allow the user to delete a specific POI when or if needed, to complete this task I simply utilise my previously built method search to narrow down and find the specific POI that is needed to delete and then I make use of the “del” function built into python to remove the specified dictionary. Alternatively, I could have used the “.pop()” function but I had no need to return the POI which had been deleted to the user which means the extra functions in “.pop()” was useless(in this scenario).

Task six says that I must allow the user to make enquiries about a specific POI, I have implemented this through my search method, once the user searches for a specific POI the method will iterate using a For loop through the keys looking for keys which match what the user has inputted if it finds more than one key the user will be prompted to input the correct key linking to the correct POI. The user will be asked after locating their specific POI if they wished to enquire on the selected establishment if so they were able to save their question in “deque” a double-ended queue, this allows us to remove items from both the front or end of the queue (in this scenario we only need to remove from the front). I have chosen a “deque” over a list because it is quicker in regards to appending and popping with time complexity of O(1) compared to a lists time complexity of O(n) this is because deques do not require any shifting operations and instead the starting node (pointer) will be linked (changed) to the new starting value in the queue, additional to its time complexity being faster I needed one item in the queue to be removed at a time.

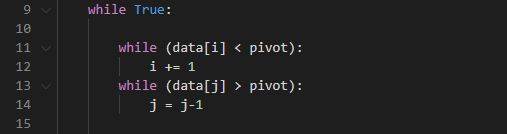
**2) Detail on algorithm (and data structure, if appropriate) implementation (this must be backed up with code, you cannot gain the credit for a writeup alone)**

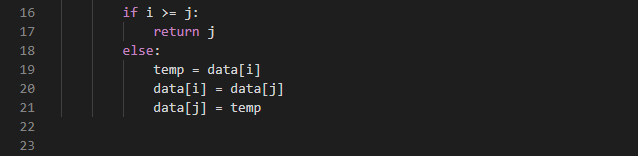
**Quicksort**: The quicksort algorithm is used in question three which stated, “The user must be able to display all POIs, sorted by name”. To implement the quicksort algorithm I started by creating a new python file which would contain the code required for quick sort to function, the backbone of my quicksort algorithm was making effective use of the Hoare Partitioning algorithm which is an algorithm developed by Tony Hoare, this algorithm starts with two points a start and endpoint (placed on opposite sides of any list presented) the algorithm also creates a variable which I have named mid which calculates the mid-point of any given list by dividing the (start + end) by two and using the math.floor function from the math module which will round any number it is given down to the nearest integer (whole number) giving us our index for the mid-point in the given list, we then assign the value which corresponds to the index and store it into a variable which I named pivot.



The Pivots role is to always keep track of the middle of two partitions allowing us to sort each side based on the pivot, left would be values less than (or equal to) the Pivot and right would be values bigger than (or equal to) the Pivot.

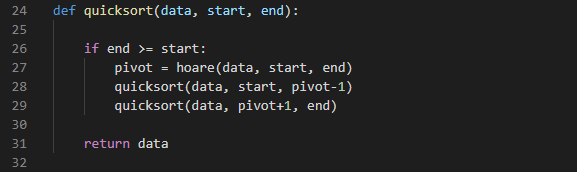
The next part when programming the Hoare algorithm was to “sort” the list, to do this we start by creating an endless loop which will continuously go through its conditions and test if they are true or not, we start by checking whether our value in the “i” (which begins at the start of the list) index is less than our Pivot, if so then “1” will be added to the variable “i” allowing it to move an index at a time through the list continuing until the conditions are met. Similarly, the program will check whether the value in the “j” index is bigger than the Pivot, if so then “1” will be subtracted from the variable “j” allowing it to move down the list until the specified conditions are met.



Once the program has completed these two while loops this means that either “i” and “j” are pointing to the Pivot or they have what’s known as a crossover (“i” which started on the left is now on the right of “j” and “j” which started on the right is now on the left of “i”) we can check this with our next line of code, on line 16 we check if “i” is bigger than or equal to “j” and if so then we will return “j” (either one but I have chosen “j”) setting “j” as the new Pivot variable (in our quicksort function). Else if this condition is not met then we will swap the values at index “i” with index “j” and index “j” with index “i” (essentially swapping the two variables, I do this by creating a temporary variable named “temp” to store the current value of “i” before replacing it with “j” allowing me to place the value of “i” into “j” later on.

Now we can move on to our quicksort method where all the recursion in this algorithm occurs. The function starts by allowing you to enter the data (list in need of sorting) the start point of the list and the endpoint (later will be assigned to either “i” or “j”), it first checks whether the end is bigger than or equal to start which is an important step for our recursion, this will allow the program to know when the Pivot is at the start or end of the list meaning that the values in the list have been sorted if the condition is not met the program will return the list.

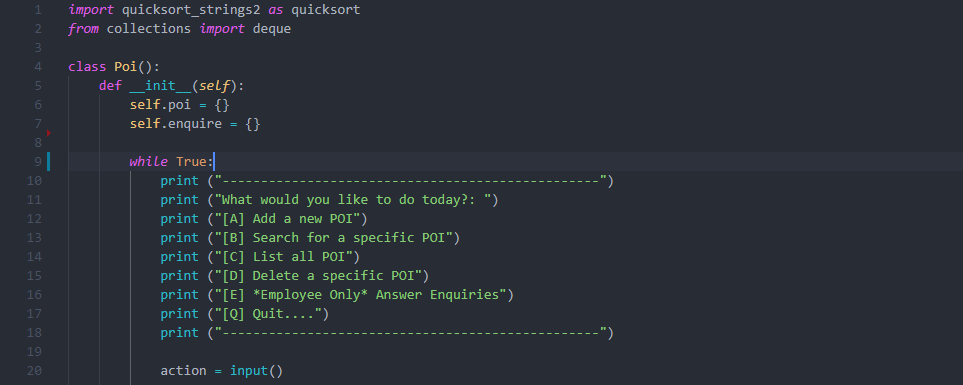
If the condition is met the program will then start running the algorithm and store the data returned from it inside a variable named Pivot, this variable will be used to recursively run the quicksort function, first, it will run with the pivot as the end but minus one from itself giving us the left side of the unsorted list (the left of the initial Pivot, we know the initial Pivot is in the right position because it would’ve been sorted the first time we ran the program). The program will repeat the steps until both sides (partitions) are fully sorted.



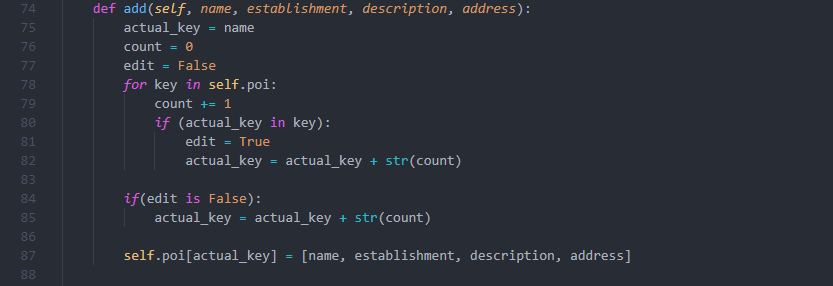
Now to use this algorithm in my POI program I had to import it as a module. I imported it as quicksort, and it works perfectly.

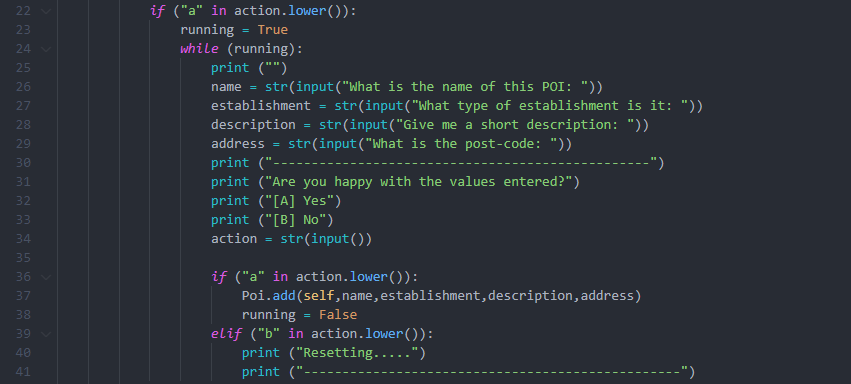


**3) How data structures and algorithms have been used in your application (this must be backed up with a working application, you cannot gain the credit for a writeup alone)**

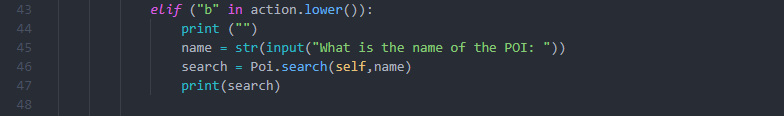
**Task one:**

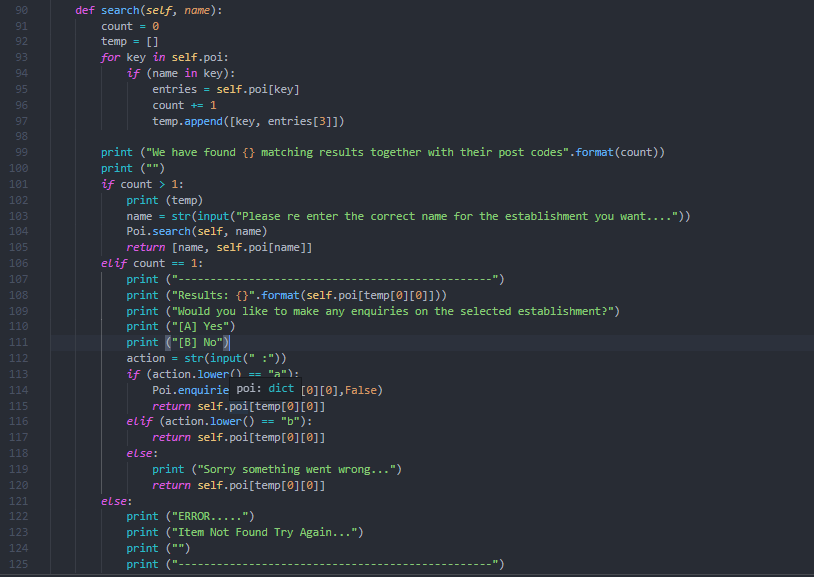
The program starts by creating two data structures (dictionaries), “self.poi” is the main dictionary which will contain all the POIs added by the user, “self.enquire” will contain questions from the users. First, the program initiates an infinite loop which will serve as a menu allowing the user to access the different sections of the program through the variable action with the input function.

Bellow will contain if and elif statements to read the users input and direct the program toward the appropriate option if the conditions are met if the user chooses “a” it will ask the user four questions (name, establishment, description, and address), once the user is happy with the values they will be directed to the “add” method inside our “Poi” class.

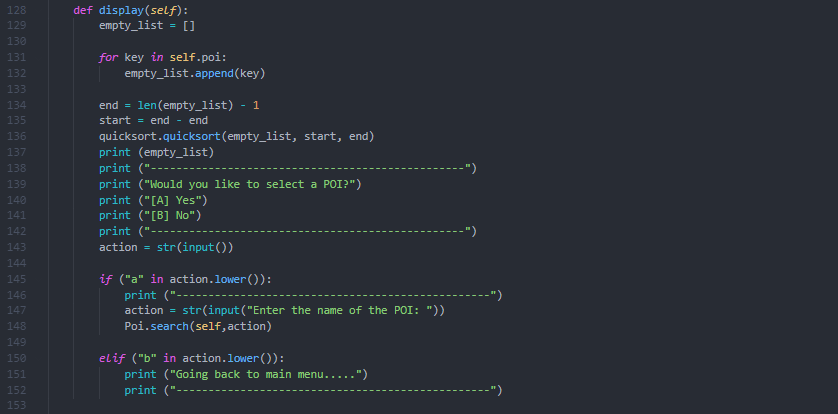
The add method simply takes the name of the establishment and checks the dictionary for keys which contains (roughly) the same name, it will add one to the “count” variable which at the end will be added to the name of the establishment creating a unique key to be stored inside the dictionary, on line 87 the program will store the establishment using the “actual\_key” as the key into the dictionary “self.poi”.

**Task two:**

The program also allows the user to search for a specific POI based on the name of the POI regardless of how many have the same name.

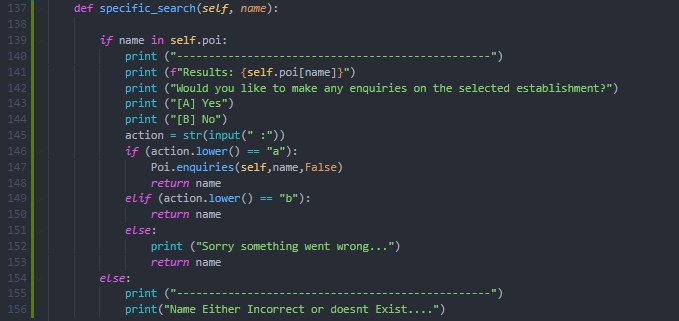
Once the user chooses option “b” they will be directed to input the name of the POI the program will then run the search method storing its return value inside a variable which can be printed later.

The search method will start by creating two variables, a “count” and a list called “temp” the program will iterate (using a for loop) through the dictionary “self.poi” storing every key and their value (index 3 which is the postcode/address) that match the “name” given by our user into “temp”. if there is more than one result (kept track by the “count” variable) the program will display all the establishments and their postcodes asking the user to re-enter the name exactly as it is saved in the dictionary. Once this is done the program will recall itself and search again, this time “count” should not exceed one allowing for the second statement to run (the elif), the user can now choose to submit an enquiry on the selected POI.

**Task three:**

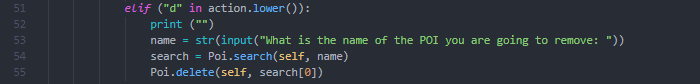
The program also allows the user to display all known POI in sorted order, first it stores all keys in a list called “empty\_list” which may pose some efficiency problems depending on the size of the Poi dictionary as it copies all keys into a list through iteration of the dictionary before sorting it, making this small operation before the sort O(n) in time complexity.

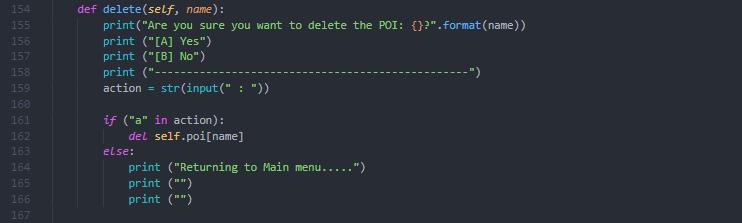
The program will figure out the start and end of the list and calls the quicksort function to start the sort were start and end are used as the “fingers” or “points” for the quicksort algorithm. Once the sorting is complete the list will be printed out and the user will be asked if they would like to select a POI, If that is the case we can easily call our search method from earlier, if not we will return to the main menu screen.

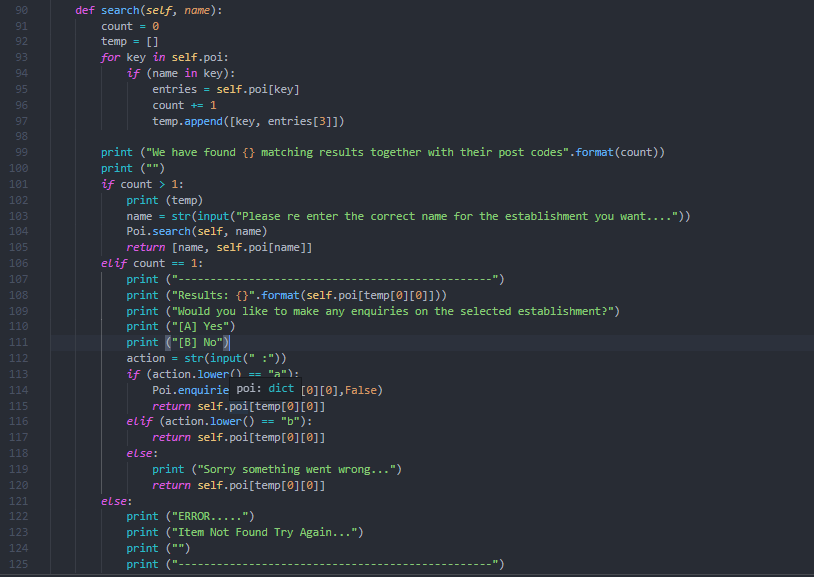
**Task four:**

Like Task two the program also allows the user to search for a specific POI but in task four the user must know the unique key which contains the POI they require.

It will simply use an If statement to determine whether the key is located in side the dictionary, if so it will relay the results and ask for enquiries, otherwise the program will show an error and go back to the main menu.

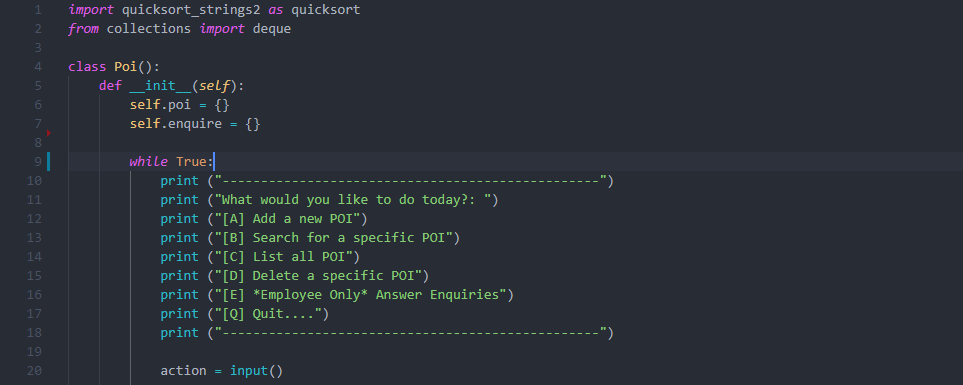
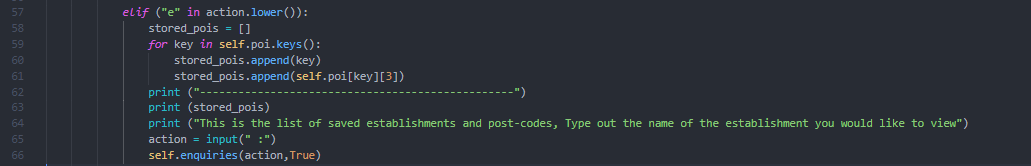
**Task five:**

If option D is selected from the main menu the program will allow the user to delete a POI of their choice, once the user inputs the name it will run through the search method to find the exact POI that the user wants to delete then stores it inside a variable called search allowing it to be called in the delete method. The delete method will double-check making sure the user is sure they want to delete the POI and if so the POI will simply be deleted by using the built-in python function “del” as dictionaries are not sorted data types but instead rely on the use of keys this function will provide us with a time complexity (on average) of O(1) as all we need is the key which we wish to delete (avoids the use of locating the correct index). Compared to lists time complexity (on average) of O(n).

**Task six:**

As mentioned earlier the program allows the user to ask questions (make enquiries on POIs), if the user decides they want to enquire about something it will take them to the enquire method in POI.



The method will first check whether the user accessing it is either an employee or not (this will be determined on which way the user enters the enquiries method), if the user is not an employee they will be asked to input their question, if a question had been asked previously on the same POI then the user's question will simply be “appended” into the queue and will be answered once all other questions asked before are answered. If there had been no previous questions asked the program will create a queue inside the dictionary “self.enquire” using the name of the POI as the key, then it will “append” the questions until answered.

If the user were to be an employee, they would enter the enquiries method using the main menu, if that option was selected they will be presented a list of stored POIs and their post-codes(for identification) to select from, once selected the program will then initiate the enquiries method using their selected POI and setting their employee status to True allowing them to access the employee section.

The employee section will print out the first question in the queue for the employee to answer and returns them to the main menu. If there were no questions available or some error were to occur the program will display the message “No questions Available” and return them to the main menu.