Project Idea Brief Overview:

I created an application, which suggests meals from different restaurants based on the mood(Hangry/On Diet) of the user and their budget or the max amount of calories respectively. The user can add their allergies and sort the results by their preferences (restaurant rating, distance, calories, price).

Tkinter Widgets:

- Labels
- Entry
- OptionMenu
- Radiobutton
- Buttons
- Message
- Listbox / Scrollbar
- Canvas

Algorithms:

- Generate Data Algorithm (generatedata.py)
- Read Data Algorithm (readdata.py)
- Unbounded Knapsack Problem Algorithm (ukp.py)
- Modified Unbounded Knapsack Problem Algorithm (ukp_calories.py)
- Quicksort Algorithm (quicksort.py)

Functions in Main.py:

- allergies(s)
- add_more()
- make_suggestions(event)
- ingredient_onselect(event)
- calculate_distance()
- new_activity()
- onselect(event)
- display_map()

Tkinter Widgets

Labels

I use labels to display texts, mainly questions to make the structure of my app more understandable for the user. For example, before the user gets to the buttons where they choose their mood, I use a Tkinter label to ask the user what's their mood. This way I tell them that this is the part of the application where they choose their mood. I use labels as sorting and adding allergies instructions.

Entry

I use the Entry widget three times in my code: for the input of the amount of money the user has, for the input of the number of calories the user wants to consume, and for the input of the ingredient that the user doesn't want to eat. I get the strings from these entries and use them in different functions.

OptionMenu

I use the OptionMenu widget as a pop-up menu for the ingredients list. Instead of typing their allergies, the user also can choose directly from the list of the ingredients.

Radiobutton

I use the Radiobutton widget two times: when the user chooses their mood and when the user chooses the sort criteria. Each Radiobutton is associated with a single variable and each button has a single value for that variable. I use these values to detect which button is chosen.

Buttons

I use three buttons in my application and each of them is associated with a command. I use the "Add" button to add an ingredient as an allergy and call the allergies(s) function. I use the "Show matched meals" button to call the new_activity() function, which displays the new window with matched meals. I use the "Display map" button to call the display_map() function.

Message

I use the message widget to display the message to the user when there's an error with chosen radio button matches, there is no such ingredient to add the allergies list, and there are no meals under the given amount of money or calories.

Listbox

I use the Listbox widget two times. First, when the user is typing the ingredient name in the entry, I display the suggested ingredients in a list box(listbox). This Listbox(listbox) is bound with ingredient_onselect(event) function. The second Listbox(mylist) is used when finding matching meals is done and I display the list of the restaurants. This Listbox(mylist) is bounded with onselect(event) function. I also use the Scrollbar widget with the second Listbox(myList).

Canvas

I use the Canvas widget in the display_map() function to display the map.

Algorithms

Generate data (generatedata.py)

In this code, I generate the following information: **number of restaurants** (I use Random module to generate a random integer between 20 - 50; cnt_restaurants), **restaurant names** (for this, I iterate for loop from 1 to cnt_restaurants (i), convert i to string and add it to string "Restaurant"), **restaurant location** (I use randint in range of 10 - 880 for x coordinate since the width of my app is 900 and in range of 10 - 470 for y coordinate since the height of my app is 500), restaurant rating (random float from 1 to 5), **number of restaurant menu items** (randint between 10 and 25; cnt_tems), for each item - **item price** (integer), **item calories** (integer), and **item ingredients** (list of strings). To generate an item ingredients list I have a general ingredients dictionary, then I randomly choose ingredients from each category of ingredients(meat, vegetables, dairy). I also create a dictionary called used {} to keep track of the ingredients which have already been added to the item ingredients, this way I prevent the cases where one item has the same ingredients several times. I write all this information in a text file, but before writing this data I write the string "NEW", this makes it easier to read the data.

Read Data Algorithm (readdata.py)

In this file, I read data from a text file and save it in a dictionary. As I already mentioned, I have a keyword "NEW", which informs me that the next few lines will give me data about a new restaurant. I keep track of the number/indexes of the lines, this helps me to use modulus and identify which feature of the restaurant I'm gonna read on this particular line. For example, line_cnt is 0 when I start reading info about a new restaurant, so if line_cnt ==

0, then this line will give me the restaurant's name. After the first 4 lines, I have item name, price, and calories on the odd lines, and I have ingredients on the even lines. I save this information in my dictionary (myData{}) and return this dictionary when I finish reading the whole file.

Unbounded Knapsack Problem Algorithm (ukp.py)

To count the maximum amount of calories I can get at a given price I decided to use the unbounded knapsack problem algorithm, which unlike the other interpretations of the knapsack problem algorithm gives me a chance to have several elements with the same amount of calories and same price. I have two functions in this file ukp and items. Function ukp takes 5 arguments: w(user's amount of money), weights(prices list), values(calories list), myData(dictionary of everything), item_indexes(indexes of items, which weren't deleted from myData). I use nested loops and go through all the weights from 1 to w and calculate the maximum value for each weight. The runtime of this function will be O(n*m), where n is a user's amount of money and m is the length of the items list(restaurant menu). I also have helper function items. Moreover, in this function, I keep track of the items that were used to get the maximum value for each weight. For this, I use a 2D list weight_indexes. This list contains information about from which weight we got to this particular weight and the index of the weight we added to get here. I call the helper function items() and give it this list, values list, final_values[w] (which is the maximum amount of values we can get with w), w, and item_indexes. This function returns the list of the items that were used to get the maximum value with w. In the worst-case scenario, the runtime of this function will be O(n), where n is the maximum value we can get.

Modified Unbounded Knapsack Problem Algorithm (ukp_calories.py)

This program is a lot like the ukp.py, but this time instead of finding the maximum value, I find the minimum value. The weights list the list of calories and the values list is the list of cost. This time main function ukp returns the minimum amount of money the user will need to get the given amount of calories and the list of items they should buy. The runtime of both functions will be the same as they were in the ukp.py.

Quicksort Algorithm (quicksort.py)

I use the quicksort algorithm to sort my final list of the restaurants, items, calories, and prices. I have 4 sorting criteria: By distance, by calories (this is only for hangry mood since the second one already gives us the fixed number of calories), by rating, by cost(this is only for the diet option since another one gives us a fixed amount of money). Sorting by ratings and calories requires a reverse sort (I mean the final list to be from maximum to minimum), meanwhile, the other two do not. To solve this problem, I had several ideas like writing if statements in the quicksort algorithm or returning the reversed list, but I decided to use a

math trick instead. As we compare the list items to the pivot value we have lines like these: **while myList[i][index] < pivot** and **while myList[j][index] > pivot**, if we want to sort from maximum to minimum the sign (<, >) changes to its opposite(>, <) respectively. So, I decided to create a variable called rev and set its default value to 1. If we want the reverse sort I set its value to -1. I give this variable to my quicksort function with other arguments and changed above mention lines to this: **while myList[i][index]*rev < pivot * rev.** If rev = 1, then everything will work like **while myList[i][index] < pivot** would work, but if rev = 1 then everything will be work like **while myList[i][index] > pivot.** Otherwise, my quicksort algorithm is a typical one. About the runtime: the worst-case runtime for quicksort is O(n^2), but it happens when the list is already sorted. Since I generate data randomly (although it doesn't guarantee that there won't be cases when the list is sorted) I felt positive for using quicksort because its average runtime is O(n log n).

Main.py Functions:

At the beginning of the Main.py, I call functions from generatedata.py and readdata.py and save all the information in the dictionary called myData.

allergies(s)

This is a function that takes a single string argument s. In this function, I go through all the items from all the restaurants and delete them if s is one of the ingredients. To avoid KeyErrors, I use a dictionary called empty. In this dictionary, the keys are string(restaurant number) + string(item number). The values of this dictionary are booleans, and I automatically assign True when I add a key. As, if key_string not in empty - works in O(1), the runtime of this function will be O(n*m), where n is a number of restaurants and m is an average number of items in the menu.

add_more()

This function is called when the button "Add" is pressed. In this function, I check if the user chose an ingredient from the pop-up menu. If yes, then I remove it from my ingredients list and call the **allergies(s)** function for the selected ingredient. Remove operation works in O(n). I also check if the entered string is empty or not. If it's not empty I check if the input string is in the ingredients list to remove it from the list, and then I call the **allergies(s)** function. Finding an element in the list takes O(n) and then removing it takes another O(n), so in the worst case, this part of my code works in O(n^2). If there is no such ingredient in my list I display the message about it. After changing my ingredients list, I assigned an updated list to the OptionMenu menu.

make_suggestions(event)

In this function, I make suggestions to the user as they type the ingredient name. I check if the entry is empty. If not, I take the entry string and go through my ingredients list. I use the startswith() method to check which elements start with the entry string. I insert those ingredients in the suggestions Listbox (listbox). Runtime for this function will be O(n*m), where n is the length of the ingredients list and m is the length of the entered string.

ingredient_onselect(event)

This is a bounded function to the Listbox(listbox), which displays suggested ingredients. As an ingredient gets selected I change allergies_string meaning to the selected ingredient.

calculate_distance()

As I read data, I call this function and calculate the distance from the current location to the restaurants' location. For this, I go through the dictionary using for loop. I update the distance value in my dictionary as the default value is 0. (I could generate distance data directly instead of the coordinates, but I really wanted to add the map feature to my app). Runtime for this function is O(n), where n is the number of restaurants.

new_activity()

This function is bounded to the 'Show matched meals' button (finish_button). As I call this function, first of all, I check if it has already been called. If no, then I open a new window(window2). If yes, then I destroy the window2. In this function, I get the user's mood and call the ukp or ukp_calories functions respectively. Although, before I call those functions, I go through my dictionary(particularly, restaurant menu items) and make a list of their prices, costs, and indexes. Then I call one of those functions and check the returned value. If the returned value is valid then I append my final restaurant and matched meals list(**answ**). After that, I call the quicksort function. In this function, I use the Message widget for every case that might not display the answers for the user (for example if there are no items under that price or the user tried to sort a list by cost when the selected mood is hangry). After all these operations, I display the final list(**answ**) on the window2 using the Listbox widget.

onselect(event)

This function is bounded to the Listbox - mylist, which is a list of final restaurants where I found the matching meals. As the user selects a restaurant from this list, I display the following info about the restaurant: Restaurant name, restaurant rating, distance to the restaurant, maximum calories/minimum price, item names and ingredient. Also, a button, which calls display_map() function. To display all this information I needed to get the

restaurant index(key) in myData but I had a problem with getting it (for example I know that key for Restaurant1 in myData equals 0, but since I sort mylist(displayed list) by different features, the list index of Restaurant1 wasn't 0 anymore). I was thinking about inserting a restaurant index as a paired element in mylist with the restaurant name, but instead, I decided to get the restaurant name as a string and save the only number at the end of the string (10:len(restaurant_name), 10 is the length of word "restaurant") as an integer. After getting a restaurant number, accessing its information was easier.

display_map()

This function can be called when a particular restaurant is selected. As it's called, I open a new window. In this function, I use the Canvas widget to create ovals. Each oval is a restaurant. I go through my final list of restaurants and matched meals, and I check the coordinates of each restaurant. If they are equal to the coordinates of the chosen restaurant I paint it yellow, otherwise I pent it red. I also display the current location by creating a blue oval on the current coordinates(which I declared at the beginning of the main.py).

Extension/Improvements and Summary:

For future work, I am considering giving a user chance to see several options from one restaurant at the same time and make rejecting a matched meal possible. For this, I would probably use another dictionary to which I would add the meals items string as a key, and before displaying the meal I would check if I have it in the used meals dictionary. I would also have to modify ukp and ukp_calories algorithms for that.

I would probably change the design of the app because as much as I love the primitive design, I think it's not really that enjoyable for users.

I want to add a few words about the run time of my program. In my code, there are many nested lists, which is not really a good thing for the run time, but since the numbers are pretty low (every n is under 50) it does not affect the runtime that much. And even if I apply my code to the real-life data, I'll probably discuss the closest restaurants and numbers will still be pretty low.

At this point, I tried to create a minimum viable product for my app, tried to find as many bugs as possible, and solved them.