**Homework 3 Student Name: Chinmay Samak**

AuE 8930: Computing and Simulation for Autonomy

Instructor: Prof. Bing Li, Clemson University, Department of Automotive Engineering

\* Refer to [Syllabus](https://docs.google.com/document/d/1ekwIf3SZGUO1toGO_iUMmZqzQb7wOpZjfq427YnCjyc/edit?usp=sharing) for homework (late) submission, grading and plagiarism policies;

\* Submission due Mon. 10/9/2023 11:59 pm via Canvas, include:

* This document (with answers), and with your program results/visualization;
* A .zip file of (modified) source code and data if any, which the TA might run.

\* All time complexity should in the big O notation.

\* Control your code versions in Github or Bitbucket git repositories.

\* Code template is in [this Github repo](https://github.com/fengziyue/CU-Computing-Autonomy/tree/master/Homework3), which can be used as a baseline for your homework.

\* If you complete this homework using C++, your final grade will be with a bonus scale 105%.

In this case, for the question templates, you’ll also need to transfer it into C++.

\* Development/coding environment for your programming:

* + IDE like [PyCharm](https://www.jetbrains.com/help/pycharm/installation-guide.html) for Python with [Anaconda](https://www.anaconda.com/products/distribution) as Python installations, or
  + CLion for C++ if use C++, or any other tools you prefer.

\* The extra questions are optional. You max score is capped as 100.

\* This homework includes two parts, and you are supposed to complete both A and B parts.

Part-A

**Question 1 (10’)**

Given an array of integers, find two numbers in it such that they can add up to a specific number.

You may assume there are exactly one solution, you can’t use the same element twice. (Only time-complexity optimized solution gets full grade)

Example:

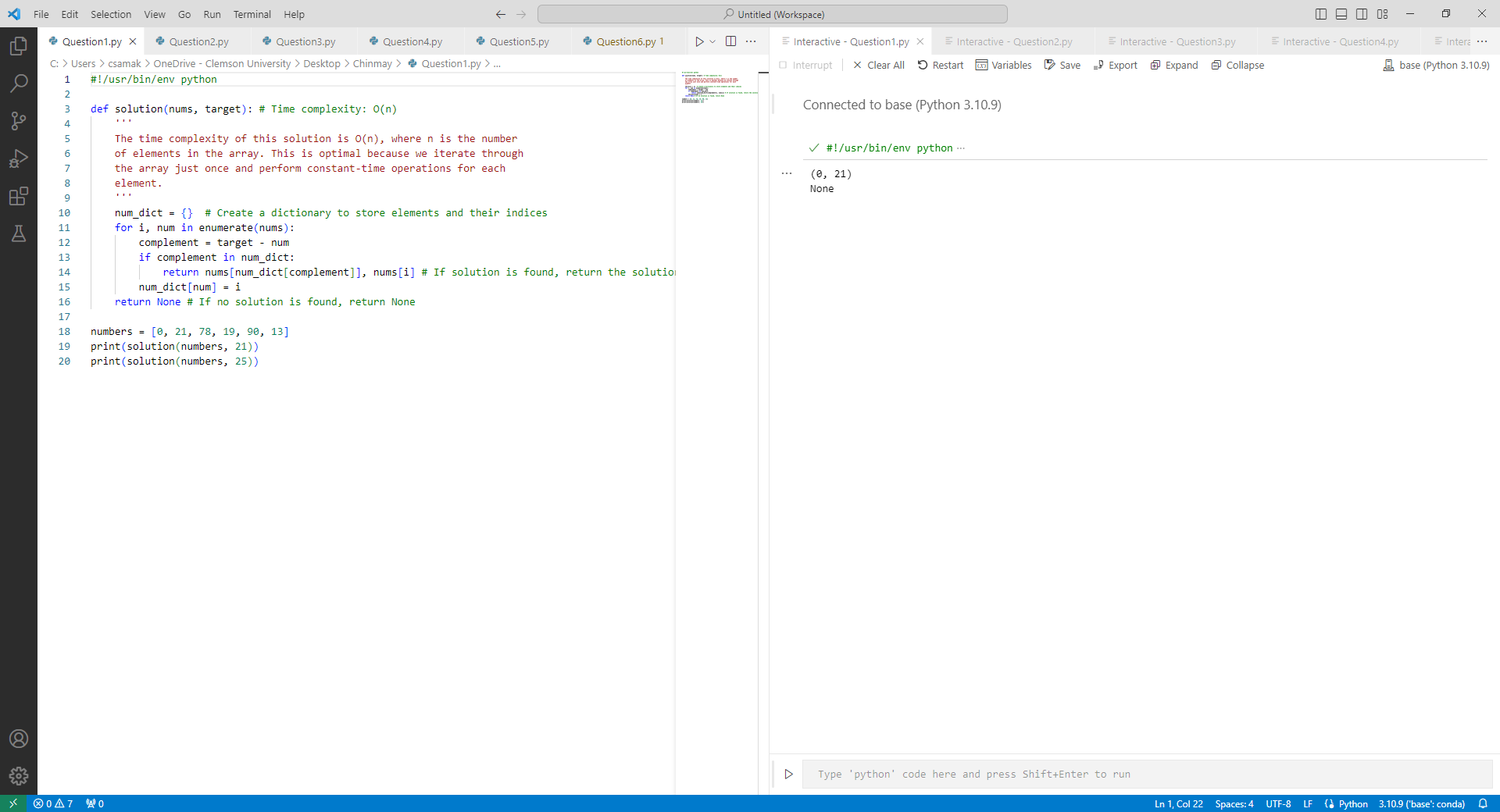
Given [2, 7, 11, 4], Target = 13.

The answer is 2 and 11.

Modify the “solution” function in the question1.py.

(Analyze your time complexity)

The time complexity of this solution is O(n), where n is the number of elements in the array. This is optimal because we iterate through the array just once and perform constant-time operations for each element.



**Question 2 (10’)**

Given a binary tree, find the max depth of it. Modify the “solution” function in the question2.py (Analyze your time complexity, and only time-complexity optimized solution gets full grade)

The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. This is optimal because it visits each node only once.

A screenshot of a computer

Description automatically generated

**Question 3 (5’)**

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example:

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

Explanation: 342 + 465 = 807.

Modify the “solution” class in question3.py, you may design your input to test it.

The time complexity of the provided solution is O(max(N, M)), where N is the length of the first linked list l1, and M is the length of the second linked list l2. This is because the algorithm processes each digit in both linked lists once, and the number of digits processed is determined by the length of the longer of the two input linked lists.

This time complexity is considered optimal for this problem because you must visit each digit in both numbers at least once to compute the sum. Therefore, this solution is time-complexity optimized and achieves the best possible time complexity for adding two numbers represented as linked lists.

A screenshot of a computer

Description automatically generated

**Question 4 (5’)**

Given a string s, find the length of the longest substring without repeating characters. You can expect the string length is less than 100, and only contains English letters.

Example 1:

Input: s = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

Modify the “solution” class in question4.py, you may design your input to test it.

The time complexity of this solution is O(n), where n is the length of the input string s. This is because we iterate through the string once, and each character is processed exactly once. Therefore, this solution is time-complexity optimized.

A screenshot of a computer

Description automatically generated

**Question 5 (5’)**

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. An input string is valid if:

* Open brackets must be closed by the same type of brackets.
* Open brackets must be closed in the correct order.
* Every close bracket has a corresponding open bracket of the same type.

Modify the “solution” function in the question5.py. (Analyze your time complexity)

The time complexity of this solution is O(n), where n is the length of the input string s. This is because we iterate through the string once, and each character is processed exactly once. Therefore, this solution is time-complexity optimized.

A screenshot of a computer

Description automatically generated

**Question 6 (5’)**

Use OpenCV to do a bilateral filter to an image, modify from question6.py, you may use your favorite image, visualize the images before and after the filtering using matplotlib.

The time complexity of the bilateral filter implementation in OpenCV is typically considered to be O(N), where N is the number of pixels in the image. However, the actual time complexity can vary depending on the hardware and specific optimization techniques used by OpenCV. This code provides a time-complexity optimized solution for applying the bilateral filter and visualizing the images.

A screenshot of a computer

Description automatically generated

**Question 7 (10’)**

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. (Note: A leaf is a node with no children.)

Example:

Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

Modify the “solution” class in question7.py, test the above example and design your test case.

The time complexity of this solution is O(N), where N is the number of nodes in the binary tree. This is because we visit each node in the tree exactly once. Therefore, this solution is time-complexity optimized.

A screenshot of a computer

Description automatically generated

**Question 8 (10’)**

Given two strings s and t, return true *if* t *is an anagram of* s*, and* false *otherwise*.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** s = "anagram", t = "nagaram"

**Output:** true

**Example 2:**

**Input:** s = "rat", t = "car"

**Output:** false

**Constraints:**

1 <= s.length, t.length <= 5 \* 104

s and t consist of lowercase English letters.

Modify the “solution” function in the question8.py. (Analyze your time complexity)

The time complexity of this solution is O(n), where n is the length of the input strings s and t. This is because we iterate through both strings once to count the character frequencies, and the dictionary comparison is also done in O(n) time. Therefore, this solution is time-complexity optimized.

A screenshot of a computer

Description automatically generated

**Extra Question 1 (2’)** (***Extra***: means it is optional for you to do)

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

Example:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6

You may design your input to test it.

The time complexity of this solution is O(n), where n is the length of the input array nums. This is because we iterate through the array once, and at each step, we perform constant-time operations. Kadane's algorithm is a well-known and time-complexity optimized solution for this problem.

A screenshot of a computer

Description automatically generated

**Extra Question 2 (2’)**

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example:

Input: lists = [[1,4,5],[1,3,4],[2,6]]

Output: [1,1,2,3,4,4,5,6]

Modify the “solution” class in extra\_question2.py, you may design your input to test it.

The time complexity of this solution is O(N\*log(k)), where N is the total number of nodes in all the linked lists, and k is the number of linked lists. This is because in each iteration, we perform a constant-time operation to extract the minimum element from the min-heap and potentially add a new element to it. This solution is time-complexity optimized for merging k sorted linked lists.

A screenshot of a computer

Description automatically generated

**Extra Question 3 (2’)**

Write a NumPy program to get the values and indices of the elements that are bigger than 10 in a given array. - Modify the extra\_question3.py

The time complexity of this operation is O(N), where N is the total number of elements in the array. It's a linear time operation because we examine each element in the array only once to create the boolean mask and find the indices. Therefore, this is a time-complexity optimized solution for this task.

A screenshot of a computer

Description automatically generated

**Extra Question 4 (2’)**

Use template matching with OpenCV to find Messi’s face in an image, try all 6 methods and plot the result. - Modify the extra\_question4/code.py

The time complexity of template matching depends on the size of the main image and the template. In this code, template matching is applied to each method separately, so the time complexity is determined by the number of methods and the size of the images. This code is time-complexity optimized for performing template matching with OpenCV.

A screenshot of a computer

Description automatically generated

**Extra Question 5 (2’)**

Write a NumPy program to add, subtract, multiply, divide two arrays element-wise.

After you *import numpy*

The first step is to use NumPy to create two arrays: *a*, *b*

*a* and *b* should be the same dimensioned.

You initialize the values for *a* and *b* when you use *numpy.array* to create them.

Then *a* and *b* can apply the *+ - \* /* , like in the ways that two integers can do.

*numpy* will do the corresponding element-wise math operations implicitly.

Please do it for both *a* and *b* for two cases: they are 1D arrays and 2D arrays, such as *a* and *b* both are 4x1, and both are 4x4.

The time complexity of these operations depends on the size of the arrays. For 1D arrays of length n, these operations are O(n). For 2D arrays of dimensions n x m, these operations are O(n \* m). This code is time-complexity optimized for performing element-wise operations on NumPy arrays.

A screenshot of a computer

Description automatically generated

**Extra Question 6 (2’)**

Use SciPy for an application of Discrete Fourier Transform (DFT), modify the extra\_question6.py, and apply DFT to the array “a” and visualize both original and result signals.

The time complexity of performing the DFT using SciPy is O(N\*log(N)), where N is the length of the input signal. This code is time-complexity optimized for applying the DFT and visualizing the original and result signals.

A screenshot of a computer

Description automatically generated

Part-B

**Demo existing and revise search algorithms (40’)**

* Reference code repo:

<https://github.com/fengziyue/CU-Computing-Autonomy/tree/master/Homework3/map-path-search>

* The TA will run and test your algorithms results using the GUI;

[a] For this question, you have the existing reference:

Occupancy gridmap class library:

*Homework3/map-path-search/gridmap.py*

Occupancy gridmap-based A\* (A-start path searching algorithm) implementation:

*Homework3/map-path-search/a\_star\_occupancy.py*

As the default, you can use the 8 connectivity for this whole question.

1) Demo existing reference and get to know the behaviour of its path search. (2’)

The demo run file is: *examples/occupancy\_map\_8n.py*

A screenshot of a computer

Description automatically generated

2) Implement occupancy gridmap-based Dijkstra for same functionality as (1) (18’)

If you prefer, you can use this as the template to revise:

*Homework3/map-path-search/a\_star\_occupancy.py*

A screenshot of a computer

Description automatically generated

[b] For this question, you have the existing reference:

Quadtree-map class library:

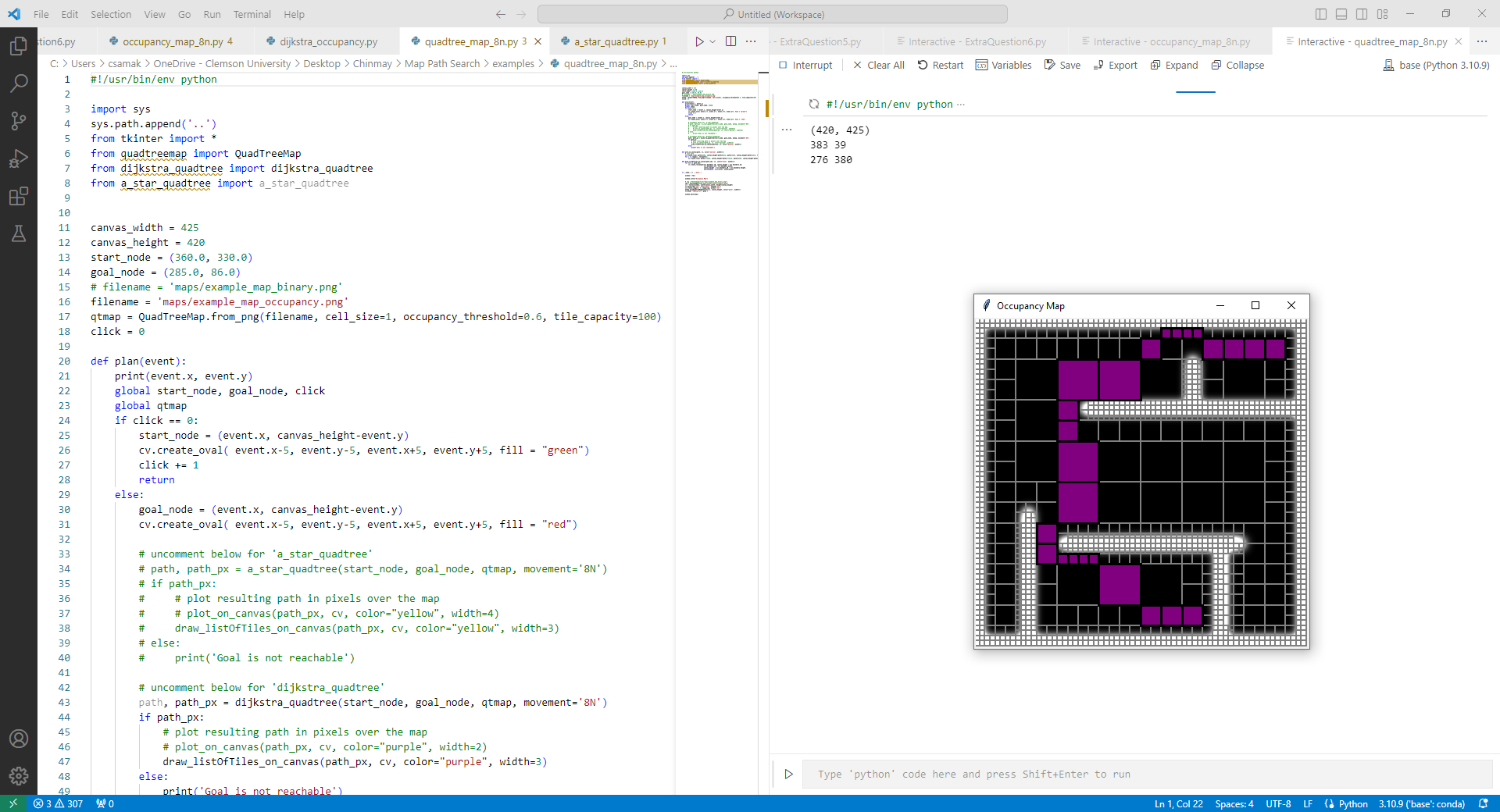
*Homework3/map-path-search/quadtreemap.py*

Quadtree map-based Dijkstra path searching algorithm implementation:

*Homework3/map-path-search/dijkstra\_quadtree.py*

3) Demo existing reference and get to know the behaviour of its path search. (2’)

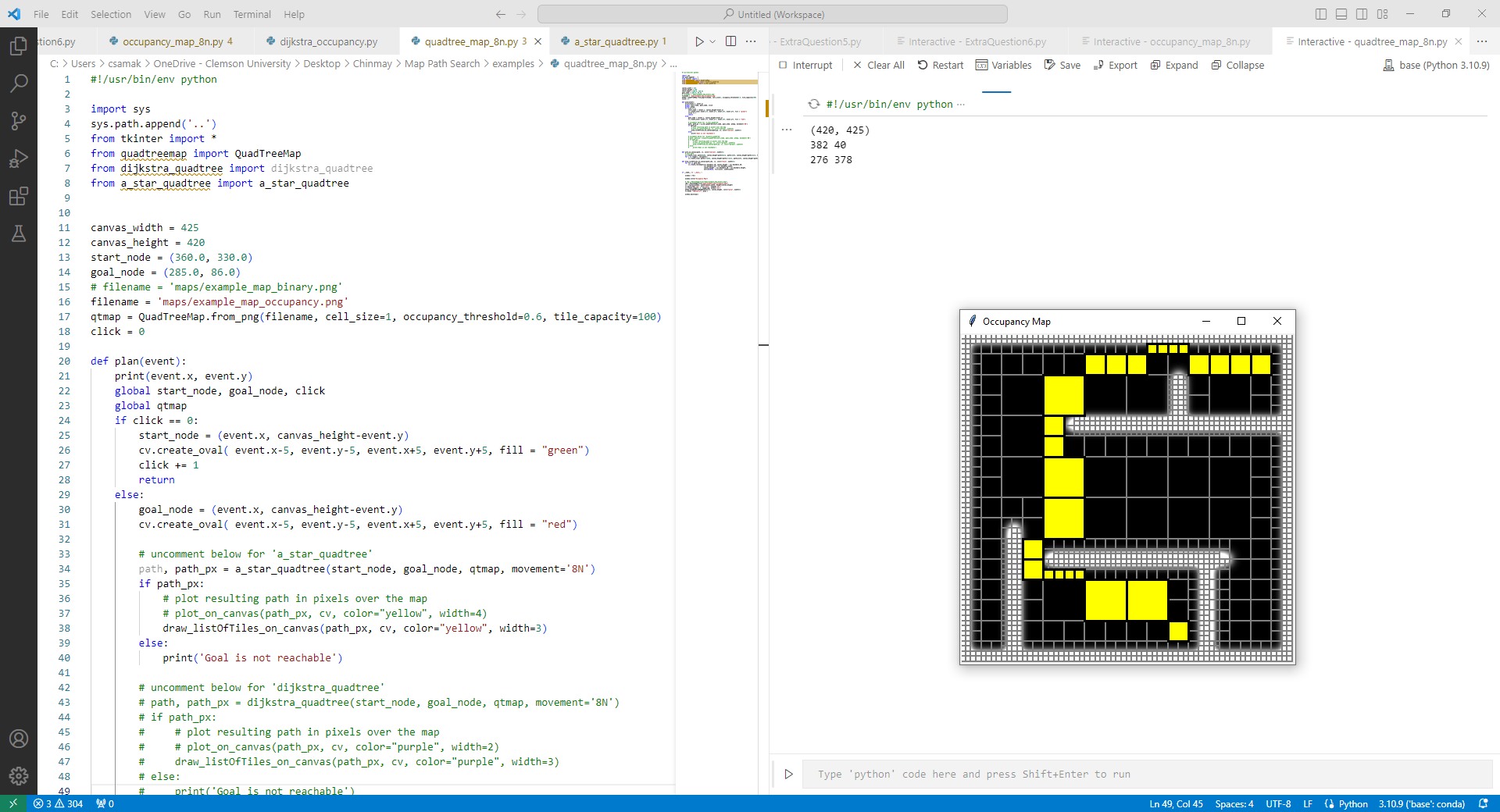
The demo run file is: *examples/quadtree\_map\_8n.py*



4) Implement Quadtree map-based A\* for same functionality as (3) (18’)

If you prefer, you can use this as the template to revise:

*Homework3/map-path-search/dijkstra\_quadtree.py*



[c] Extra credits (optional to complete):

* 5) Try a few of different granularity, and describe the potential affect (2’);

The granularity of the grid map can significantly affect the performance and behavior of both Dijkstra's and A\* algorithms when applied to pathfinding problems. Here are a few scenarios with different granularities and their potential effects on these algorithms:

**Fine Granularity:**

* In a fine-grained grid map where each cell represents a small area, both Dijkstra's and A\* algorithms can find detailed and precise paths.
* In this case, the algorithms are more computationally intensive because there are many cells to explore, and they may take longer to execute.
* Fine granularity allows for precise pathfinding around obstacles and complex terrain.

**Coarse Granularity:**

* In a coarse-grained grid map with larger cells, Dijkstra's and A\* algorithms can find paths more quickly because there are fewer cells to explore.
* However, the paths generated may be less precise due to the limited granularity, and they may not capture intricate details in the environment.
* Coarse granularity is suitable for scenarios where speed is more critical than path precision.

**Variable Granularity:**

* Using variable granularity, one can adapt the grid map to have different levels of granularity in different regions.
* For example, one might use fine granularity around obstacles or complex areas and coarse granularity in open spaces.
* This approach can combine the benefits of both fine and coarse granularities but may require more sophisticated algorithms to handle transitions between granularities.

**Dynamic Granularity:**

* Dynamic granularity adjusts the grid map's granularity dynamically based on the specific pathfinding task or the complexity of the environment.
* When navigating through open areas, the grid map can have a coarser granularity to speed up the search. In contrast, it can switch to fine granularity when approaching obstacles.
* Dynamic granularity can improve the efficiency of pathfinding while maintaining path precision.

**Granularity vs. Memory:**

* Finer granularity requires more memory to store the grid map, as each cell occupies memory.
* Coarser granularity reduces memory usage but may lead to inaccuracies in pathfinding.
* The choice of granularity should balance the memory constraints of the system with the need for precise pathfinding.

A screenshot of a computer

Description automatically generated

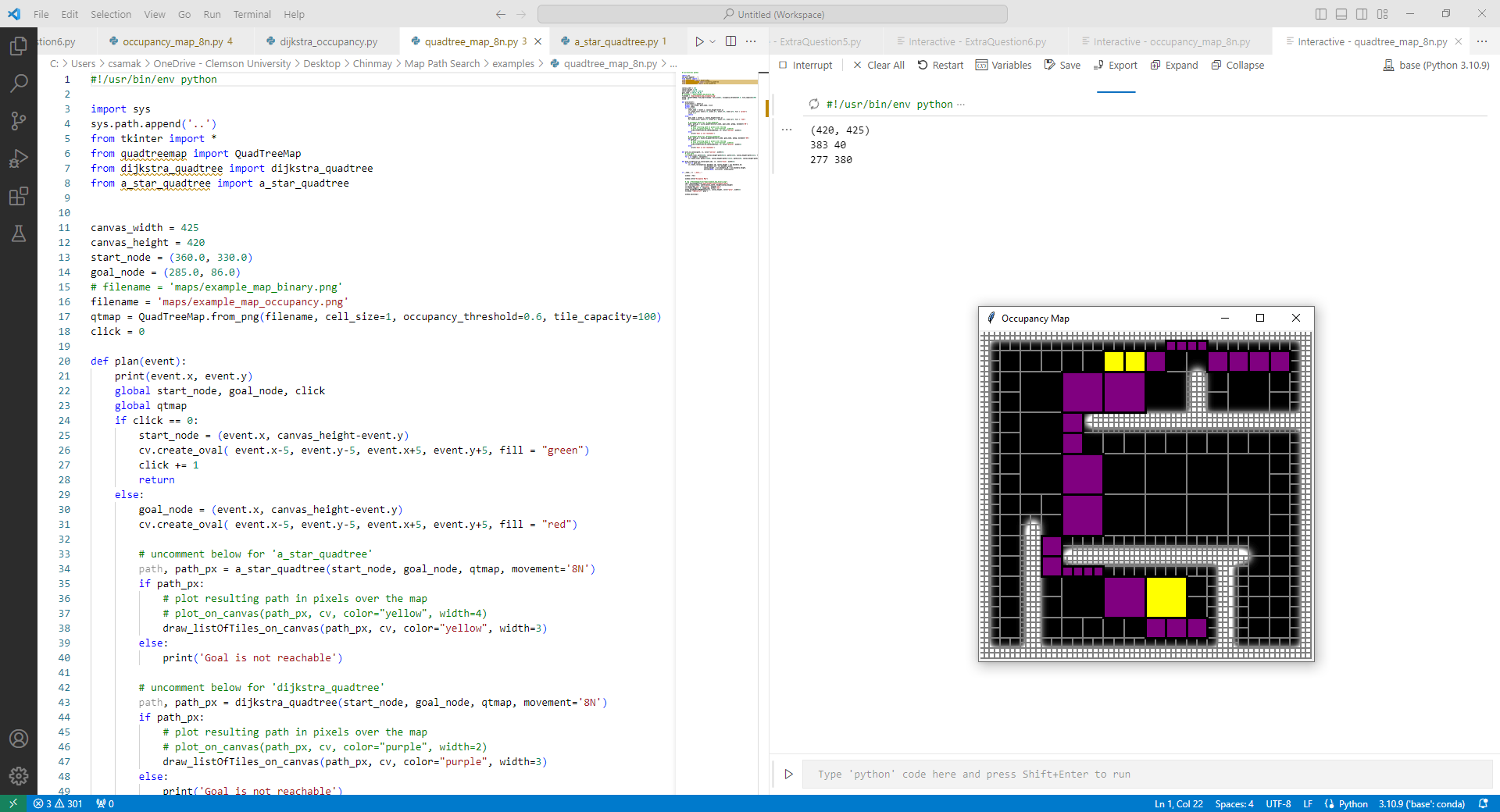
A screenshot of a computer

Description automatically generated

* 6) Show both Dijkstra and A\* algorithms into the GUI for the same mouse events (2’);

A screenshot of a computer

Description automatically generated



* 7) Analyze your time complexity of each algorithm (2’);

**Dijkstra's Algorithm Time Complexity:**

Dijkstra's algorithm explores nodes in the order of their total cost from the start node. Here are the key components contributing to the time complexity:

* **Priority Queue Operations:** In each iteration, the algorithm performs operations like inserting nodes into the priority queue and extracting the node with the minimum total cost. These operations have a time complexity of O(log N), where N is the number of nodes in the queue.
* **Loop Over Neighbors:** For each node, the algorithm examines its neighbors. In the worst case, each node can have a constant number of neighbors, so this part contributes O(E), where E is the number of edges in the graph.

Overall, the **time complexity of Dijkstra's algorithm is O((V + E) \* log(V))**, where V is the number of vertices (nodes) and E is the number of edges in the graph.

**A\* Search Algorithm Time Complexity:**

A\* search algorithm is similar to Dijkstra's algorithm but with the addition of a heuristic function. Here's how the time complexity is affected:

* **Priority Queue Operations:** Similar to Dijkstra's algorithm, priority queue operations contribute O(log N) time complexity per operation.
* **Loop Over Neighbors:** Like Dijkstra's, the algorithm examines neighbors, contributing O(E) in the worst case.
* **Heuristic Function:** The heuristic function is evaluated for each node, adding an extra cost. In the worst case, the heuristic function evaluation adds a constant time to each node.

Overall, the **time complexity of A\* search algorithm is O((V + E) \* log(V))**, where V is the number of vertices (nodes) and E is the number of edges in the graph. The presence of a heuristic function can reduce the number of nodes explored in practice, but in the worst case, it doesn't change the overall time complexity.

Both algorithms have the same worst-case time complexity because A\* search is essentially a modified version of Dijkstra's algorithm with a heuristic that can help in practice but doesn't change the asymptotic complexity. The choice between them depends on whether a heuristic can provide useful guidance in a specific problem domain.

Below are some visualization hints for your references:

