

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 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](#), 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](#) for Python with [Anaconda](#) 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.

```

File Edit Selection View Go Run Terminal Help
Question1.py X Question2.py Question3.py Question4.py Question5.py Question6.py I ...
C > Users > csmak > OneDrive - Clemson University > Desktop > Chinmay > Question1.py ...
1 #!/usr/bin/env python
2
3 def solution(nums, target): # Time complexity: O(n)
4     """
5         The time complexity of this solution is O(n), where n is the number
6         of elements in the array. This is optimal because we iterate through
7         the array just once and perform constant-time operations for each
8         element.
9     """
10    num_dict = {} # Create a dictionary to store elements and their indices
11    for i, num in enumerate(nums):
12        complement = target - num
13        if complement in num_dict:
14            return nums[num_dict[complement]], nums[i] # If solution is found, return the solution
15        num_dict[num] = i
16    return None # If no solution is found, return None
17
18 numbers = [0, 21, 78, 19, 98, 13]
19 print(solution(numbers, 21))
20 print(solution(numbers, 25))

```

Connected to base (Python 3.10.9)

```

Interactive - Question1.py X Interactive - Question2.py X Interactive - Question3.py X Interactive - Question4.py X ...
Interrupt | Clear All Restart Variables Save Export Expand Collapse
base (Python 3.10.9)

```

```

✓ #!/usr/bin/env python ...
...
(0, 21)
None

```

Type 'python' code here and press Shift+Enter to run

In 1, Col 22 Spaces:4 UTF-8 LF Python 3.10.9 (base)conda

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.

```

File Edit Selection View Go Run Terminal Help
Question1.py X Question2.py X Question3.py Question4.py Question5.py Question6.py I ...
C > Users > csmak > OneDrive - Clemson University > Desktop > Chinmay > Question2.py ...
1 #!/usr/bin/env python
2
3 class TreeNode(object):
4     """Definition of a binary tree node."""
5     def __init__(self, x):
6         self.val = x
7         self.left = None
8         self.right = None
9
10    def solution(root): # Time complexity: O(n)
11        """
12            The time complexity of this solution is O(n), where n is the number
13            of nodes in the binary tree. This is optimal because it visits each
14            node only once.
15        """
16        if root is None:
17            return 0
18        left_depth = solution(root.left)
19        right_depth = solution(root.right)
20        return max(left_depth, right_depth) + 1
21
22 a15 = TreeNode(15)
23 a7 = TreeNode(7)
24 a20 = TreeNode(20)
25 a9 = TreeNode(9)
26 a3 = TreeNode(3)
27 a20.left = a15
28 a20.right = a7
29 a3.left = a9
30 a3.right = a20
31 print(solution(a3))

```

Connected to base (Python 3.10.9)

```

Interactive - Question1.py X Interactive - Question2.py X Interactive - Question3.py X Interactive - Question4.py X ...
Interrupt | Clear All Restart Variables Save Export Expand Collapse
base (Python 3.10.9)

```

```

✓ #!/usr/bin/env python ...
...
3

```

Type 'python' code here and press Shift+Enter to run

Cell 1 of 2

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: $(2 \rightarrow 4 \rightarrow 3) + (5 \rightarrow 6 \rightarrow 4)$

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.

The screenshot shows a Jupyter Notebook interface with several tabs open. The active tab is 'Untitled (Workspace)'. The code in the cell is as follows:

```
# Definition for singly-linked list.
class ListMode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def addTwoNumbers(self, l1: ListMode, l2: ListMode) -> ListMode:
        carry = 0
        dummy_head = ListMode()
        current = dummy_head
        while l1 or l2:
            x = l1.val if l1 else 0
            y = l2.val if l2 else 0
            sum = x + y + carry
            carry = sum // 10
            current.next = ListMode(sum % 10)
            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next
            current = current.next
        if carry > 0:
            current.next = ListMode(carry)
        return dummy_head.next # Linked list representing the sum

    def extractNumbers(self, result: ListMode) -> str:
        result_str = ""
        while result:
            result_str += str(result.val) + " -> "
            result = result.next
        return result_str.rstrip(" -> ")

    def extractInteger(self, head):
        num_str = ""
        current = head
```

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: $s = \text{"abcabcbb"}$

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.

The screenshot shows a Jupyter Notebook interface with several tabs at the top: Question1.py, Question2.py, Question3.py, Question4.py (which is currently active), Question5.py, and Question6.py. The main area displays the following Python code for Question 4:

```
1 #!/usr/bin/env python
2
3 class Solution: # Time complexity: O(n)
4     def lengthOfLongestSubstring(self, s: str) -> int:
5         """
6             The time complexity of this solution is O(n), where n is the length of the
7             input string s. This is because we iterate through the string once, and each
8             character is processed exactly once. Therefore, this solution is time-complexity
9             optimized.
10            ...
11
12         char_set = set()
13         max_length = 0
14         left = 0
15         for right in range(len(s)):
16             while s[right] in char_set:
17                 char_set.remove(s[left])
18                 left += 1
19             char_set.add(s[right])
20             max_length = max(max_length, right - left + 1)
21
22         # Example usage:
23         solution = Solution()
24         s = "abcabcbb"
25         result = solution.lengthOfLongestSubstring(s)
26         print(result)
```

The code defines a `Solution` class with a method `lengthOfLongestSubstring` that takes a string s and returns its length. It uses a sliding window approach with a set to keep track of characters seen so far. The code also includes an example usage at the bottom.

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 Jupyter Notebook interface. The left pane shows a code cell with the following Python code:

```
1 #!/usr/bin/env python
2
3 class Solution: # Time complexity: O(n)
4     def isValid(self, s: str) -> bool:
5         ...
6
7         The time complexity of this solution is O(n), where n is the length of the
8         input string s. This is because we iterate through the string once, and each
9         character is processed exactly once. Therefore, this solution is time-complexity
10        optimized.
11
12        stack = []
13        bracket_pairs = {')': '(', ')': '(', '}': '{', ']': '['}
14        for char in s:
15            if char in bracket_pairs.values():
16                stack.append(char)
17            elif char in bracket_pairs.keys():
18                if not stack or stack.pop() != bracket_pairs[char]:
19                    return False
20                else:
21                    return False
22        return len(stack) == 0
23
24 # Example usage:
25 solution = Solution()
26 s = "((())"
27 result = solution.isValid(s)
28 print(result)
```

The right pane shows the output of the code execution:

```
✓ #!/usr/bin/env python ...
...
True
```

Below the code cell is a text input field with the placeholder "Type 'python' code here and press Shift+Enter to run".

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.

The screenshot shows a Jupyter Notebook interface with several tabs at the top: Question1.py, Question2.py, Question3.py, Question4.py, Question5.py, Questions.py, and Untitled (Workspace). The main area displays a Python script for bilateral filtering:

```

1 #!/usr/bin/env python
2
3 import cv2
4 import matplotlib.pyplot as plt
5
6 ...
7 The time complexity of the bilateral filter implementation in OpenCV is typically
8 considered to be O(N), where N is the number of pixels in the image. However, the
9 actual time complexity can vary depending on the hardware and specific optimization
10 techniques used by OpenCV. This code provides a time-complexity optimized solution
11 for applying the bilateral filter and visualizing the images.
12 ...
13
14 # Read the image.
15 img = cv2.imread('OriginalImage.jpg')
16
17 # Apply bilateral filter.
18 bilateral = cv2.bilateralFilter(img, d=15, sigmaColor=75, sigmaSpace=75)
19
20 # Save the output.
21 cv2.imwrite('FilteredImage.jpg', bilateral)
22
23 # Visualize the original and filtered images.
24 plt.subplot(121), plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB)), plt.title('Original Image')
25 plt.subplot(122), plt.imshow(cv2.cvtColor(bilateral, cv2.COLOR_BGR2RGB)), plt.title('Bilateral Fi
26 plt.show()

```

Below the code, two images are displayed side-by-side: 'Original Image' and 'Bilateral Filtered Image'. Both images show a landscape scene with a bridge and trees at sunset.

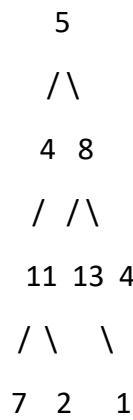
At the bottom of the notebook, there is a text input field with the placeholder 'Type "python" code here and press Shift+Enter to run' and a status bar indicating 'Cell 1 of 1'.

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,



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.

The screenshot shows a Jupyter Notebook interface with multiple tabs open. The main code cell contains Python code for a binary tree problem, specifically a solution to check if a path sum exists in a binary tree. The code uses a recursive approach to traverse the tree and calculate path sums. It includes comments explaining the time complexity and example usage. The code is as follows:

```

1 #!/usr/bin/env python
2
3 # Definition for a binary tree node.
4 class Treemode:
5     def __init__(self, val=0, left=None, right=None):
6         self.val = val
7         self.left = left
8         self.right = right
9
10 class Solution: # Time complexity: O(N)
11     def hasPathSum(self, root: Treemode, sum: int) -> bool:
12         ...
13
14         The time complexity of this solution is O(N), where N is the number of nodes
15         in the binary tree. This is because we visit each node in the tree exactly
16         once. Therefore, this solution is time-complexity optimized.
17
18     if not root:
19         return False
20
21     # Check if it's a leaf node and if its value matches the remaining sum
22     if not root.left and not root.right:
23         return root.val == sum
24
25     # Recursive check for left and right subtrees
26     left_has_path_sum = self.hasPathSum(root.left, sum - root.val)
27     right_has_path_sum = self.hasPathSum(root.right, sum - root.val)
28
29     return left_has_path_sum or right_has_path_sum
30
31     # Example usage:
32
33     # Create a binary tree and then call hasPathSum.
34     # For example, to check if root-to-leaf path with sum 22 exists:
35     root = Treemode(5)
36     root.left = Treemode(4)
37     root.right = Treemode(8)
38     root.left.left = Treemode(11)
39     root.left.left = Treemode(13)
40     root.left.left = Treemode(1)
41     root.left.left.left = Treemode(7)
42     root.left.left.right = Treemode(2)
43     root.right.right.right = Treemode(1)
44
45     solution = Solution()
46     result = solution.hasPathSum(root, 22)
47
48     print(result)

```

The notebook also shows an interactive cell below the code, which runs the code and prints 'True'.

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.

The screenshot shows a Jupyter Notebook interface with multiple tabs open. The main code cell contains Python code for checking if two strings are anagrams. The code uses a dictionary to count character frequencies in both strings and then compares them. It includes a check for string lengths and handles empty strings. The output cell shows the result of running the code with two test cases: ("anagram", "nagaram") which prints True, and ("rat", "car") which prints False.

```
1 #!/usr/bin/env python
2
3 class Solution: # Time complexity: O(n)
4     def isAnagram(self, s: str, t: str) -> bool:
5         ...
6             The time complexity of this solution is O(n), where n is the length
7             of the input strings s and t. This is because we iterate through both
8             strings once to count the character frequencies, and the dictionary
9             comparison is also done in O(n) time. Therefore, this solution is
10            time-complexity optimized.
11
12     # Check if lengths of string s and t match
13     if len(s) != len(t):
14         return False
15     char_count_s = {}
16     char_count_t = {}
17
18     # Count character frequencies in string s
19     for char in s:
20         char_count_s[char] = char_count_s.get(char, 0) + 1
21
22     # Count character frequencies in string t
23     for char in t:
24         char_count_t[char] = char_count_t.get(char, 0) + 1
25
26     # Example usage:
27     solution = Solution()
28     print(solution.isAnagram("anagram", "nagaram")) # Should print True
29     print(solution.isAnagram("rat", "car")) # Should print False
```

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.

The screenshot shows a Jupyter Notebook interface with multiple tabs open. The main code cell contains the following Python code:

```
Question4.py  Questions.py  Question4.py  Question4.py  Question4.py  ExtraQuestion1.py  ...  Untitled (Workspace)
C > Users > csmak > OneDrive - Clemson University > Desktop > Chinmay > ExtraQuestion1.py ...
1 #!/usr/bin/env python
2
3 class Solution: # Time complexity: O(n)
4     def maxSubArray(self, nums):
5         ...
6             The time complexity of this solution is O(n), where n is the length of
7             the input array nums. This is because we iterate through the array once,
8             and at each step, we perform constant-time operations. Kadane's algorithm
9             is a well-known and time-complexity optimized solution for this problem.
10
11         max_sum = nums[0]
12         current_sum = nums[0]
13         for num in nums[1:]:
14             current_sum = max(num, current_sum + num)
15             max_sum = max(max_sum, current_sum)
16
17         return max_sum
18
19     # Example usage:
20     solution = Solution()
21     nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
22     result = solution.maxSubArray(nums)
23     print(result) # This should print 6 for the provided example.
```

The notebook also includes an interactive Python console on the right side with the output "6".

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 $O(N \cdot \log(k))$, 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.

```

ons.py  Question1.py  Question2.py  Questions3.py  ExtraQuestion1.py  ExtraQuestion2.py  Untitled (Workspace)
1 #!/usr/bin/env python
2
3 from queue import PriorityQueue
4
5 # Definition for singly-linked list.
6 class ListNode:
7     def __init__(self, val=0, next=None):
8         self.val = val
9         self.next = next
10
11 class Solution: # Time complexity: O(N*log(k))
12     def mergeKLists(self, lists):
13
14         The time complexity of this solution is O(N*log(k)), where N is the
15         total number of nodes in all the linked lists, and k is the number
16         of linked lists. This is because in each iteration, we perform a
17         constant-time operation to extract the minimum element from the
18         min-heap and potentially add a new element to it. This solution is
19         time-complexity optimized for merging k sorted linked lists.
20
21     if not lists:
22         return None
23
24     # Create a priority queue (min-heap)
25     min_heap = PriorityQueue()
26
27     # Push the first node from each list into the min-heap
28     for i, node in enumerate(lists):
29         if node:
30             min_heap.put((node.val, i, node))
31
32     dummy_head = ListNode()
33     current = dummy_head
34
35     while not min_heap.empty():
36         val, list_idx, node = min_heap.get()
37         current.next = node
38         current = current.next
39         if node.next:
40             min_heap.put((node.next.val, list_idx, node.next))
41
42     return dummy_head.next
43
44
45     # Create a linked list from a list of values.
46     def createLinkedList(values):
47         if not values:
48             return None
49         head = ListNode(values[0])
50         current = head
51
52         for val in values[1:]:
53             current.next = ListNode(val)
54             current = current.next
55
56     return head

```

Type 'python' code here and press Shift+Enter to run

In 1, Col 22 Spaces:4 UTF-8 LF Python 3.10.9 (base) conda

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.

The screenshot shows the Spyder IDE interface with several windows open. On the left, there are multiple tabs for files like 'Question1.py', 'Question2.py', etc. The main code editor window contains the following Python script:

```
#!/usr/bin/env python
import numpy as np
...
# 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.
...
x = np.array([[0, 10, 20], [20, 30, 40]])
print("Original array:")
print(x)
...
# Find values greater than 10
values_gt_10 = x[x > 10]
print("Values greater than 10:")
print(values_gt_10)
print("Indices of values greater than 10:")
...
# Find indices of values greater than 10 using np.where
indices_gt_10 = np.where(x > 10)[0]
print("Indices of values greater than 10:")
print(indices_gt_10)
```

The right side of the interface shows the 'Interactive' window with the following output:

```
Original array:
[[ 0 10 20]
 [20 30 40]]
Values greater than 10:
[20 30 40]
Indices of values greater than 10:
[0 1 1]
```

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.

The screenshot shows a Jupyter Notebook interface with several code cells and their corresponding visual outputs.

```

1 #!/usr/bin/env python
2
3 import cv2
4 from matplotlib import pyplot as plt
5
6 ...
7 The time complexity of template matching depends on the size of the main image
8 and the template. In this code, template matching is applied to each method
9 separately, so the time complexity is determined by the number of methods and
10 the size of the images. This code is time-complexity optimized for performing
11 template matching with OpenCV.
12
13
14 # Load the main image and the template
15 img = cv2.imread('Messi Full.jpg', 0)
16 template = cv2.imread('Messi Face.jpg', 0)
17 w, h = template.shape[::-1]
18
19 # All the 6 methods for comparison in a list
20 methods = ['cv2.TM_CCOEFF', 'cv2.TM_CCOEFF_NORMED', 'cv2.TM_CCORR',
21           'cv2.TM_CCORR_NORMED', 'cv2.TM_SQDIFF', 'cv2.TM_SQDIFF_NORMED']
22
23 for meth in methods:
24     img_copy = img.copy()
25     method = eval(meth)
26
27     # Apply template Matching
28     res = cv2.matchTemplate(img_copy, template, method)
29     min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)
30
31     # For methods TM_SQDIFF and TM_SQDIFF_NORMED, take minimum
32     if method in [cv2.TM_SQDIFF, cv2.TM_SQDIFF_NORMED]:
33         top_left = min_loc
34     else:
35         top_left = max_loc
36
37     # Draw a rectangle around the matched region
38     bottom_right = (top_left[0] + w, top_left[1] + h)
39     cv2.rectangle(img_copy, top_left, bottom_right, 255, 2)
40
41     # Plot the result
42     plt.subplot(121), plt.imshow(res, cmap='gray')
43     plt.title('Matching Result')
44     plt.xticks([]), plt.yticks([])
45     plt.subplot(122), plt.imshow(img_copy, cmap='gray')
46     plt.title('Detected Point')
47     plt.xticks([]), plt.yticks([])
48     plt.suptitle(meth)
49
50 plt.show()

```

The notebook displays two sets of results for the `cv2.TM_CCOEFF` and `cv2.TM_CCORR` methods. Each set includes a "Matching Result" image showing the grayscale input image with a heatmap overlay indicating the match score, and a "Detected Point" image showing the original image with a red rectangle highlighting the detected region where the template was found.

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*,

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.

```

File Edit Selection View Go Run Terminal Help
Untitled (Workspace)
C:\Users\csmak>OneDrive - Clemson University>Desktop>Chinmay>ExtraQuestion6.py > ...
1 #!/usr/bin/env python
2
3 import numpy as np
4
5 ...
6 The time complexity of these operations depends on the size of the arrays. For 1D arrays
7 of length n, these operations are O(n). For 2D arrays of dimensions n x m, these operations
8 are O(n * m). This code is time-complexity optimized for performing element-wise operations
9 on NumPy arrays.
10 ...
11
12 # Create 1D arrays
13 a_1d = np.array([1, 2, 3, 4])
14 b_1d = np.array([5, 6, 7, 8])
15
16 # Element-wise operations for 1D arrays
17 addition_1d = a_1d + b_1d
18 subtraction_1d = a_1d - b_1d
19 multiplication_1d = a_1d * b_1d
20 division_1d = a_1d / b_1d
21
22 # Create 2D arrays
23 a_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
24 b_2d = np.array([[17, 18, 19, 20], [21, 22, 23, 24], [25, 26, 27, 28], [29, 30, 31, 32]])
25
26 # Element-wise operations for 2D arrays
27 addition_2d = a_2d + b_2d
28 subtraction_2d = a_2d - b_2d
29 multiplication_2d = a_2d * b_2d
30 division_2d = a_2d / b_2d
31
32 # Print the results
33 print("1D Array Operations:")
34 print("Addition: ", addition_1d)
35 print("Subtraction: ", subtraction_1d)
36 print("Multiplication: ", multiplication_1d)
37 print("Division: ", division_1d)
38
39 print("\n2D Array Operations:")
40 print("Addition: ", addition_2d)
41 print("Subtraction: ", subtraction_2d)
42 print("Multiplication: ", multiplication_2d)
43 print("Division: ", division_2d)

```

Connected to base (Python 3.10.9)

```

✓ #!/usr/bin/env python ...
...
10 Array Operations:
Addition: [ 6  8 10 12]
Subtraction: [-4 -4 -4 -4]
Multiplication: [ 5 12 21 32]
Division: [0.2          0.33333333 0.42857143 0.5         ]

20 Array Operations:
Addition:
[[18 20 22 24]
[26 28 30 32]
[34 36 38 40]
[42 44 46 48]]
Subtraction:
[[-10 -16 -16 -16]
[-16 -16 -16 -16]
[-16 -16 -16 -16]
[-16 -16 -16 -16]]
Multiplication:
[[ 17 36 57 80]
[108 132 161 192]
[225 260 297 336]
[377 420 465 512]]
Division:
[[0.0988235 0.11111111 0.15780474 0.2
{0.238809524 0.27272727 0.30434783 0.33333333}
{0.36 0.38461538 0.48748741 0.42857143}
{0.44827586 0.46666667 0.48387097 0.5
]]]

```

Type 'python' code here and press Shift+Enter to run

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.

```

File Edit Selection View Go Run Terminal Help
Untitled (Workspace)
C:\Users\csmak>OneDrive - Clemson University>Desktop>Chinmay>ExtraQuestion6.py > ...
1 #!/usr/bin/env python
2
3 from matplotlib import pyplot as plt
4 import numpy as np
5 from scipy.fft import fft
6
7 ...
8 The time complexity of performing the DFT using SciPy is O(N*log(N)), where N is
9 the length of the input signal. This code is time-complexity optimized for applying
10 the DFT and visualizing the original and result signals.
11 ...
12
13 # Define original signal
14 fre = 5 # Frequency in terms of Hertz
15 fre_samp = 50 # Sample rate
16 t = np.linspace(0, 2, *fre_samp, endpoint=False)
17 a = np.sin(fre * 2 * np.pi * t)
18
19 # Plot settings
20 figure, axis=plt.subplots(2)
21
22 # Plot the original signal
23 axis[0].plot(t, a)
24 axis[0].set_xlabel('Time (s)')
25 axis[0].set_ylabel('Amplitude')
26 axis[0].set_title('Original Signal')
27
28 # Perform Discrete Fourier Transform (DFT) on original signal
29 A = fft(a)
30 N = len(a)
31 frequencies = np.fft.fftfreq(N, 1 / fre_samp)
32
33 # Plot the DFT Result
34 axis[1].stem(frequencies, np.abs(A) / N)
35 axis[1].set_xlabel('Frequency (Hz)')
36 axis[1].set_ylabel('Amplitude')
37 axis[1].set_title('DFT Result')
38
39 # Plot settings
40 plt.tight_layout()
41 plt.show()

```

Original Signal

DFT Result

Type 'python' code here and press Shift+Enter to run

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`

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

The screenshot shows a Jupyter Notebook interface with several code cells and a visualization window.

```

# File: radquestions4.py
# This file contains a Python script for performing path search using Dijkstra's algorithm on an occupancy grid map. It includes imports for sys, tkinter, numpy, gridmap, and matplotlib.pyplot, along with a_star_occupancy and dijkstra_occupancy modules. The script initializes a canvas, sets start and goal nodes, and handles mouse events to mark nodes. It then performs path planning using either A* or Dijkstra's algorithm and visualizes the result on a black background with a purple path outline.

```

The visualization window titled "Occupancy Map" shows a rectangular arena with obstacles. A purple line represents the path found by the algorithm, starting from a green dot and ending at a red dot.

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

Quadtree-map class library:

Homework3/map-path-search/quadtreenode.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/quadtreenode_8n.py*

The screenshot shows a Jupyter Notebook interface with several code cells and a visualization window.

```

# File: radquestions4.py
# This file contains a Python script for performing path search using Dijkstra's algorithm on an occupancy grid map using a quadtree data structure. It includes imports for sys, tkinter, numpy, gridmap, and matplotlib.pyplot, along with a_star_quadtreenode and dijkstra_quadtreenode modules. The script initializes a canvas, sets start and goal nodes, and handles mouse events to mark nodes. It then performs path planning using either A* or Dijkstra's algorithm on the quadtree representation and visualizes the result on a black background with a purple path outline.

```

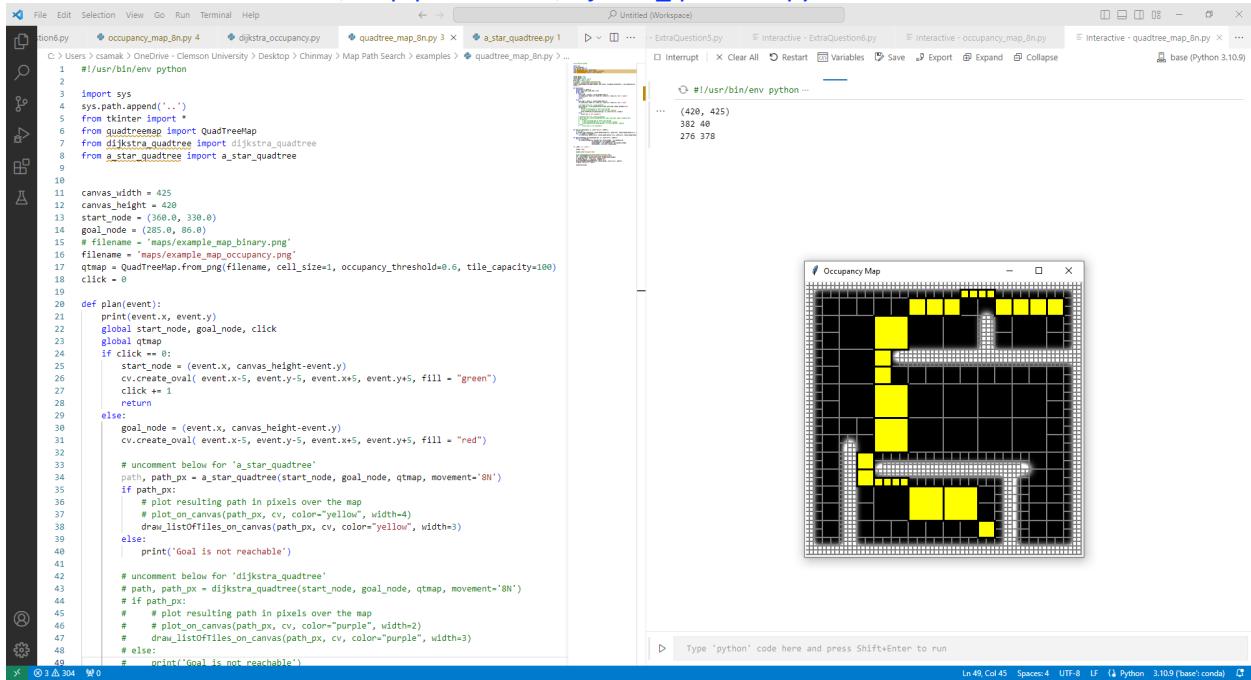
The visualization window titled "Occupancy Map" shows a rectangular arena with obstacles. A purple line represents the path found by the algorithm, starting from a green dot and ending at a red dot. The map is overlaid with a grid of small squares, indicating the quadtree structure used for the search.

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



The screenshot shows a Jupyter Notebook interface with several open files. On the left, the code for `dijkstra_quadtree.py` is visible, containing Python code for implementing A* search on a quadtree map. On the right, there is a visualization titled "Occupancy Map" showing a grid-based map with yellow and black cells, representing obstacles and free space respectively. The terminal output at the top right shows the execution of the script and some numerical results.

```
#! /usr/bin/env python
# Import sys
# Import Tkinter
# From quadtreemap import QuadTreeMap
# From dijkstra_quadtree import dijkstra_quadtree
# From a_star_quadtree import a_star_quadtree

# canvas_width = 425
# canvas_height = 420
start_node = (360, 0, 330, 0)
goal_node = (285, 0, 86, 0)
# filename = 'maps/example_map_binary.png'
filename = 'maps/example_map_occupancy.png'
qmap = QuadTreeMap.from_png(filename, cell_size=1, occupancy_threshold=0.6, tile_capacity=100)
click = 0

def planevent():
    print(event.x, event.y)
    global start_node, goal_node, click
    global qmap
    if click == 0:
        start_node = (event.x, canvas_height - event.y)
        cv.create_oval(event.x - 5, event.y - 5, event.x + 5, event.y + 5, fill = "green")
        click += 1
    else:
        goal_node = (event.x, canvas_height - event.y)
        cv.create_oval(event.x - 5, event.y - 5, event.x + 5, event.y + 5, fill = "red")

    # uncomment below for 'a_star_quadtree'
    path, path_px = a_star_quadtree(start_node, goal_node, qmap, movement='SH')
    if path:
        # plot resulting path in pixels over the map
        # plot_on_canvas(path_px, cv, color="yellow", width=4)
        draw_listoffiles_on_canvas(path_px, cv, color="yellow", width=3)
    else:
        print('Goal is not reachable')

    # uncomment below for 'dijkstra_quadtree'
    # path, path_px = dijkstra_quadtree(start_node, goal_node, qmap, movement='SH')
    # if path:
    #     # plot resulting path in pixels over the map
    #     # plot_on_canvas(path_px, cv, color="purple", width=2)
    #     # draw_listoffiles_on_canvas(path_px, cv, color="purple", width=3)
    # else:
    #     print('Goal is not reachable')

    # Type 'python' code here and press Shift+Enter to run
    Ln 49 Col 45 Spaces: 4 UTF-8 LF Python 3.10.9 (base) conda
```

[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.

The screenshot shows a Jupyter Notebook interface with several code cells and a visualization window.

Code Cells:

```
#!/usr/bin/env python
1
2
3 import sys
4 sys.path.append('..')
5 from tkinter import *
6 import time
7 from gridmap import OccupancyGridMap
8 import matplotlib.pyplot as plt
9 from a_star.occupancy import a_star_occupancy
10 from dijkstra.occupancy import dijkstra_occupancy
11 from util import plot_path
12 import copy
13
14 canvas_width = 420
15 canvas_height = 420
16 start_node = (360.0, 330.0)
17 goal_node = (285.0, 86.0)
18 # gmap = OccupancyGridMap.from_png('maps/example_map_occupancy.png', 1)
19 gmap = OccupancyGridMap.from_png('maps/example_map_binary.png', 1)
20 click = 0
21
22 def plan(event):
23     print(event.x, event.y)
24     global start_node, goal_node, click
25     global gmap
26     if click == 0:
27         start_node = (event.x, canvas_height-event.y)
28         cv.create_oval(event.x-5, event.y-5, event.x+5, event.y+5, fill = "green" )
29         click += 1
30     return
31
32 else:
33     goal_node = (event.x, canvas_height-event.y)
34     cv.create_oval(event.x-5, event.y-5, event.x+5, event.y+5, fill = "red" )
35
36 # uncomment below for a_star.occupancy
37 path, path_px = a_star_occupancy(start_node, goal_node, copy.deepcopy(gmap), 'BH')
38 if path_px:
39     # plot resulting path in pixels over the map
40     plot_on_canvas(path_px, cv, color="yellow", width=4)
41 else:
42     print('Goal is not reachable')
43
44 # uncomment below for dijkstra.occupancy
45 path, path_px = dijkstra_occupancy(start_node, goal_node, copy.deepcopy(gmap), 'BH')
46 if path_px:
47     # plot resulting path in pixels over the map
48     plot_on_canvas(path_px, cv, color="purple", width=2)
49 else:
50     print('Goal is not reachable')
```

Output Window:

```
292 97
328 304
```

Occupancy Map Visualization:

The screenshot shows a Jupyter Notebook interface with several open files. On the left, the code for `occupancy_map_8n.py` is visible, containing Python code for pathfinding using Dijkstra's, A*, and Quadtree methods. The right side features a visualization titled "Occupancy Map" showing a grid-based map with various colored cells (black, white, yellow, purple) representing different environments or obstacles. Below the visualization is a terminal window showing the output of a command, and at the bottom, there is a code editor and a run button.

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

The screenshot shows a Jupyter Notebook workspace with several open files. On the left, the code for `quadtree_map_0n.py` is visible, containing Python code for implementing Dijkstra's and A* search algorithms using a quadtree data structure. On the right, a visualization titled "Occupancy Map" shows a grid-based map with various colored cells (black, purple, yellow) representing obstacles and free space. A path is plotted in red, starting from a purple node and ending at a yellow node. The terminal output at the top right shows the execution of the script and some statistics: (428, 425), 383 40, 277 386.

- 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:

