



UNIVERSITI MALAYA

Algorithm Analysis & Design

The Mystery of Marshall Mansion Murder

Name	Matric Number
Mujahed Yahia Murad Mohammed	S2042189
Zalat Bara Sherif Ibrahim	S2115641
Samer Abukhader	S2100683
Makrem Abdulkrim Ajmi Ziani	S2116735
Aiman Muhamad Hasan Talib	17203116

Part 1: Who poisoned Marshal?	4
Discussion	4
DFS	4
Flood fill	4
BFS	4
Runtime complexity	5
Pseudocode	5
Code	6
Part 2: Cracking the chest lock code	7
Discussion	7
Random Guess	7
Educated guess	7
Brute force	7
Pseudocode	8
Runtime complexity	
Complexity: $O(k^n)$	8
Code	8
Part 3: Same but not identical	9
Discussion	9
Word-to-word comparison	9
Sentence comparison	9
2 pointer algorithm	9
Pseudocode	10
Runtime complexity	10
Code	11
Part 4: Find that book	12
Discussion	12
Linear Search	12
Jump Search	12
Binary Search	12
Pseudocode	13
Runtime complexity	13
Code	13
Part 5: Secret Message	14
Discussion	14
Frequency Analysis	14
Bruteforce	14
Regular Decoding	15
Pseudocode	15
Runtime complexity	15
Code	15
Part 6: Find the next clue	16
Discussion	16

Branch and Bound	16
Bruteforce	16
Dynamic Programming	17
Pseudocode	17
Runtime complexity	17
Code	18
Part 7: Almost there!	19
Discussion	19
Frequency Analysis	19
Similar letters	19
Dictionary lookup	19
Pseudocode	20
Runtime complexity	20
Code	21
Part 8: Murder Suspect	23
Discussion	23
Multicriteria preference analysis	23
Objective criteria	23
Weight Approach	24
Pseudocode	25
Runtime complexity	26
Code	26
Part 9: Story Ending	27

Part 1: Who poisoned Marshal?

Discussion

The question is asking us to investigate and determine who poisoned Mr. Marshall. We have been given the layout of the mansion and the surrounding areas. The objective is to search all the rooms in the main building for any clues that could help identify the person responsible for the murder. This question can be approached from many different directions. If we consider the rooms in the mansion to be nodes in a graph and connect them accordingly, we will be able to use some graph traversal algorithm to traverse all the rooms.

DFS

DFS (Depth-First Search) is a graph traversal algorithm that starts at a chosen node and explores as far as possible along each branch before backtracking to explore other branches. It prioritises exploring deeper nodes in the graph before visiting sibling nodes. Even though it is possible to implement this algorithm to search the rooms, it is not as effective as BFS due to its backtracking nature.

Flood Fill

The flood fill algorithm is particularly useful for tasks such as graph colouring, region labelling, or determining connected components within a graph. By traversing the graph in a manner similar to BFS and DFS. While it may be modified to fit our purpose of searching all the rooms, it is more inclined towards other applications, and so creating a different version of flood fill that serves this question is not needed.

BFS

BFS is an algorithmic technique used for graph traversal. It explores all the neighbours of a node at the current depth level before moving on to the nodes at the next level. It is a systematic approach that efficiently explores the graph, visiting each node once, without unnecessary backtracking. In the context of investigating the murder, BFS can be a useful algorithm to search the rooms, as it will spread and search the rooms efficiently and without missing any of them.

Runtime complexity

Complexity: $O(V + E)$

The runtime of BFS is $O(V + E)$, where **V** represents the number of vertices and **E** represents the number of edges in the graph. That comes from the main loop which goes over all the vertices and edges only once.

Pseudocode

1. Initialise empty set 'visited' and empty queue 'queue'
2. Add start_node to queue
3. While queue is not empty
 - a. Remove first node from queue
 - b. If removed node has not been visited
 - i. Output node
 - ii. Add node to visited set
 - iii. For every neighbour of the node
 1. If neighbour has not been visited
 - a. Add the neighbour to the queue

Code

```
def bfsIterative(self, start_node):
    # Initialize a visited set
    visited = set()
    # Initialize the queue
    queue = Queue()
    queue.enqueue(start_node)
    # While the queue is not empty
    while not queue.is_empty():
        # Dequeue the front node
        node = queue.dequeue()
        # If the node has not been visited
        if node not in visited:
            # Process node
            print("Visiting:", node)
            self.output.append(node)
            # Add node to visited
            visited.add(node)
            # Add neighbors to the queue
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    queue.enqueue(neighbor)
```

```
class Queue:
    def __init__(self):
        self.queue = []

    def is_empty(self):
        return len(self.queue) == 0

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue.")
        return self.queue.pop(0)

    def size(self):
        return len(self.queue)
```

Part 2: Cracking the chest lock code

Discussion

The given task is to crack the code of the lock, since we aren't given any hints and we have no where to look for them, the approach that is going to be used is going to involve some kind of guessing. While guessing is not an optimal approach but that is the only viable approach.

Random Guess

Usually when unlocking a lock that has digit combinations a person would randomly guess the code, this might be a viable solution if there are no hints to lead anywhere, this might be faster than other methods or slower, it is random afterall.

Educated guess

Randomly guessing is one way, but guessing based on statistics is definitely a safer way. This way is most likely going to be more efficient than random guessing, trying codes that are more likely to be the correct ones, like 111, 222, 123, 456, 999, and such, is a more efficient approach. While this approach will eventually lead us to brute forcing the rest of the codes, it is a better approach than random guessing.

Brute force

The lock has only 3 digits, making the possible combinations 1000 in total. Trying all of the different combinations is fine for this small number, but for larger numbers it will quickly become computationally inefficient, but since there are only 3 digits in this problem, the brute forcing of this lock is going to be quick. Other advantages include not having to keep track of the random guesses that have been made, making it quite simple to implement.

Pseudocode

1. Initialise a list of digits to be used
2. Initialise a stack to store the combinations
3. While the stack is not empty
 - a. Set combination to the last element of the stack
 - i. If the combination length is equal to the number of digits in the lock
 1. Try to unlock the lock
 2. If lock unlocked
 - a. Return the result
 - ii. Else If the combination length is less than the number of digits in the lock
 1. Loop over all digits
 - a. Add the combination plus the digit to the stack

Runtime complexity

Complexity: $O(k^n)$

The function uses an iterative approach to generate all possible combinations of the 3 digits, so the complexity of the generation of the combinations is **$O(k^n)$** , where **k** is the number of digits and **n** is the length of the combination. Since we have 10 digits possible to choose from, and 3 digits for the combination the program will take 10^3 steps to run, it is similar to having 3 nested for loops that go for 10 times each, it will also have a complexity of **$O(k * k * k)$** or **$O(k^n)$**

Code

```
Output
def bruteforce_iterative(self):
    # Initialize the digits list and stack
    digits = list(range(self.start, self.end + 1))
    stack = [[]]
    # While the stack is not empty
    while stack:
        # Pop the top combination
        combination = stack.pop()
        # If the combination reached the length needed
        if len(combination) == self.digits:
            # Unlock the lock
            result = self.lock.unlock("".join(map(str, combination)))
            if result:
                self.found = True
                return self.found
        else:
            # Add the digits to the combination
            for digit in digits:
                stack.append(combination + [digit])
```

Part 3: Same but not identical

Discussion

In this question we are given almost 2 identical letters and we are asked to tell the differences apart. Looking at it initially, the letters seem to only have differences in some words, but after some work, it is apparent that one of the letters is longer than the other. This requires a work around.

Word-to-word comparison

Word to word comparison is most likely the first idea that comes to mind, this would work if there was no difference in length, but since there is a difference in length, meaning one letter is longer than the other, word to word comparison will result in a very incorrect answer, the comparison will be fine, until the point where the extra word shows up, at that point the whole comparison is shifted and all of the comparisons will produce an incorrect result.

Sentence comparison

In this method, the letters are split up into sentences, this makes it so that the difference in the length can be accounted for, if in any case the sentences are not equal all that is needed is to add padding to the shorter sentence, that makes both sentences equal in length. The only problem with this approach is that if the extra word in the sentences is in the middle, this will still throw off the comparison and we will have a shift. But if the extra word is always in the end this method will work.

2 pointer algorithm

Two pointers may not be the most intuitive but if you know the idea it will be easy to work with. It is a good option as it doesn't need any edits or workarounds to get it functioning. It works by putting first splitting the letters up into sentences and then comparing the sentences using two pointers, the first pointer is placed at the first word of the first sentence and the second pointer is placed at the first word of the second sentence, the words pointed at by the pointer are compared to see if they are the same or not, and both pointers advance together. In the end, if one of the pointers hasn't covered the whole sentence, that means there is an extra word in that sentence, so we add that word. This method only works when the extra word is placed at the end. With the assumption that the extra word is always at the end, either this method or the sentence comparison method can work, but this approach was chosen because it is easier to implement and understand.

Pseudocode

1. Read the letters and split them into sentences
2. Initialise pointer1 to 0
3. Initialise pointer2 to 0
4. For every sentence1 and sentence2 in letter1 and letter2
 - a. While pointer1 is less than length of sentence1 and pointer2 is less than length of sentence2
 - i. If the word of pointer1 is unequal to word of pointer2
 1. Mark those words as mismatch
 - ii. Increment pointer1 and pointer2 by 1
 - b. If pointer1 did not reach the end of sentence1
 - i. Mark the rest of the words in sentence1 as new words
 - c. If pointer2 did not reach the end of sentence2
 - i. Mark the rest of the words in sentence2 as new words

Runtime complexity

Complexity: $O(n*m*k)$

The main loop of the code has **$O(n)$** where **n** is the number of sentences in the letters. Then the pointers keep looping until they reach the end of the sentence, which has a complexity of **$O(m)$** where **m** is the number of words in the sentence. In each inner loop there is a comparison that happens between the words, that takes **$O(k)$** where **k** is the number of letters in the word. Finally when the main while loop finishes there is a chance that one of the pointers didn't reach the end of the sentence, which means that pointer will continue looping over the rest of the sentence, but that is very insignificant to consider, because it will have a complexity of **$O(\text{\# words in sentence} - \text{\# words finished})$** , which will most likely be very small.

This makes the overall complexity of the code **$O(n * m * k)$** , where **n** is the number of sentences, and **m** is the number of words in the sentences, and **k** is the number of letters in every word (for the comparison).

Code

```
2-pointer Approach

for sentence1, sentence2 in zip(letter1, letter2):
    # Initialize the pointers
    pointer1 = 0
    pointer2 = 0

    # While pointers are not out of bounds
    while pointer1 < len(sentence1) and pointer2 < len(sentence2):
        word1 = sentence1[pointer1]
        word2 = sentence2[pointer2]

        # Compare the words
        if not sm.brute_force(word1, word2):
            book1.append(word1)
            book2.append(word2)
        # Increment the pointers
        pointer1 += 1
        pointer2 += 1

    # If pointer has not reached end of sentence
    if pointer1 < len(sentence1):
        # Add the rest of the sentence to the book title
        while pointer1 < len(sentence1):
            book1.append(sentence1[pointer1])
            pointer1 += 1
    # If pointer has not reached end of sentence
    if pointer2 < len(sentence2):
        # Add the rest of the sentence to the book title
        while pointer2 < len(sentence2):
            book2.append(sentence2[pointer2])
            pointer2 += 1
```

Part 4: Find that book

Discussion

The question now asks how do we find the book within hundreds of books around the library knowing that the library keeps the book sorted in alphabetical order. This just implies the use of a search algorithm to find the book, but the books are sorted in alphabetical order, so we might be able to leverage that.

Linear Search

A straightforward process known as a linear search progressively examining each item in a list until a match is discovered. Although it is simple to build, it is not the best option for a big library. Searching through hundreds of books might take a while, especially if the one you're looking for is at the end of the list.

Jump Search

Jump search is a good search approach for sorted arrays, it involves splitting the array into blocks and traversing those blocks linearly, it might seem like it is just normal linear search, but checks are done to skip some blocks, if the target element is greater than the element at the end of the current block, the block can just be skipped, but if it is smaller, then the block can be searched. This method might work well with this problem but it is quite complex and does not result in constant time complexity, meaning it could be slower than expected.

Binary Search

Binary search is also a search method that is used for sorted arrays, it works by halving down the search space, resulting in a very efficient time complexity. It works by comparing the middle element of the array with the target, if the target is smaller then it is definitely present in the left half, so the right half does not need to be searched.

Pseudocode

1. Initialise low to 0
2. Initialise high to last index in book list
3. While low is less than or equal to high
 - a. Calculate middle index
 - b. If middle index has the book title
 - i. Output book title and index
 - c. If middle index comes before the book title in alphabetical order
 - i. Increment the low by middle index + 1
 - d. If middle index comes after the book title in alphabetical order
 - i. Decrement the high by middle index + 1

Runtime complexity

Complexity: $O(\log(n))$

The search area is cut in half in each phase of binary search, thereby lowering the number of factors to take into account. The search space is reduced by half each time until the target element is located or it is not found. The quantity of iterations needed to locate the target element is hence logarithmic with respect to the size of the input. In more technical terms, binary search has a time complexity of **$O(\log n)$** , where **n** is the size of the input.

Code

```
Binary Search

def binarysearch(self, book_list, book_title):
    # Initialize the low and high pointers
    low = 0
    high = len(book_list) - 1
    # While the low pointer is less than or equal to the high pointer
    while low <= high:
        # Calculate the mid pointer
        mid = (low + high) // 2
        mid_title = book_list[mid]
        # If book found return index
        if mid_title == book_title:
            return mid

        # If book not found, adjust pointers
        elif mid_title < book_title:
            low = mid + 1
        else:
            high = mid - 1
    # If book not found return -1
    return -1
```

Part 5: Secret Message

Discussion

The question proposes an encoded text:

“Ymfy ujwxts nx htrns! ktw rj! Nk dtz knsi ymnx styj, qttp fwtzsi rd uwtujwyd. Mnsy: N anxnyji ymj fwjf bnym rd ywtqqjd kwtr ymj lfwjjs xmji. - 5”

To decode the text we need to determine the cipher type first. With a quick look it looks like a substitution cipher, and some popular substitution ciphers include Caesar’s Cipher, the Alberti Cipher, and the Vigenère Cipher. The number at the end of the text doesn’t seem to be a part of the message and after further trial and error it seems to be the shift of the Caesar Cipher.

Caesar’s Cipher works by shifting the alphabet by a certain number, for example:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E

When encoding text an A will change to an F, a B will change to a G, and so on. But when decoding the table is flipped, so an F will be an A, a G will be a B. Following this rule the word “Ymfy” decoded will be “That”.

Frequency Analysis

In cryptography, unknown ciphers are usually decoded using frequency analysis, a method where letter frequency in the ciphered text is calculated and compared to the letter frequency of the source language. The most frequent letter in the English language is E, so if the most frequent letter in a ciphered text is F, it will most likely correspond to E in its unciphered form. This technique is used when nothing is known about the cipher method, it also does not guarantee a correct answer.

Bruteforce

Since the cipher is known to be Caesar’s Cipher, bruteforcing the different values of the shift is a feasible solution, since there are only 25 possible shifts it won’t be very computationally expensive, the only downside to this problem is that every produced possible decipher will have to be manually checked, while some solution can be implemented like checking for the existence of English dictionary words in the deciphered text to determine if its correctly deciphered, it is still not optimal and can result in wrong answers.

Normal Decoding

Instead of using frequency analysis and risk getting a wrong answer, or using brute force and manually checking 25 possible deciphers, we can just use the given shift, since the shift is known to be 5, looping over every letter and unshifting it will give the correct deciphered text.

Pseudocode

1. Create deciphering dictionary
2. Initialize deciphered text variable
3. For every letter in ciphered text
 - a. Get corresponding letter from dictionary
 - b. Append to deciphered text variable
4. Return deciphered text variable

Runtime complexity

Complexity: $O(n)$

The code loops over every letter of the ciphered text, this gives **$O(n)$** complexity, where **n** is the number of letters in the cipher. For the rest of the code it is just the calculation of the corresponding letter, which is **$O(1)$** . Same goes for appending the deciphered letter to the string, **$O(1)$** complexity. This gives a total of **$O(n)$** complexity for the whole program.

The code uses a utility function that creates a dictionary which stores the deciphered values of the letters, but that loops over all letters of the alphabet, so it doesn't scale with the input.

Code

```
Create Dictionary

def create_dictionary(self):
    # String of english letters alphabet
    alphabet = string.ascii_lowercase

    # Dictionary to store the encryption
    decryption_dict = {}
    # Loop through the alphabet
    for i in range(len(alphabet)):
        # Calculate the decrypted letter
        decrypted_letter = alphabet[(i - self.shift) % 26]
        # Add the decrypted letter to the dictionary
        decryption_dict[alphabet[i]] = decrypted_letter
        decryption_dict[alphabet[i].upper()] = decrypted_letter.upper()
    return decryption_dict
```

```
Decypher

def decypher(self):
    decrypted = ""
    # For every character in the text
    for char in self.text:
        # If the character is a letter
        if char.isalpha():
            # Decrypt the character
            decrypted_char = self.cipher[char]
            # Add the decrypted character to the decrypted text
            decrypted += decrypted_char
        else:
            # Add the character to the decrypted text
            decrypted += char
    return decrypted
```

Part 6: Find the next clue

Discussion

The question gives some clues about what the victim could have been doing before he died. Following the previous hint, the investigator visits the shed, there he finds a trolley and a bunch of items, further inspection led to the investigator figuring out that the trolley can only carry 30 kg of weight at most, with that information we have to figure out what items were carried on the trolley, this is the table of items.

Item	Weight
A sack of corn for the chicken at the barn	12 kg
A hoe for the green house	5 kg
An oil tank filled with fuel for the boat at the lake	10 kg
Four pieces of tyres for the car in the garage	16 kg

A quick calculation leads us to realise that there is no way all the items could have been carried on the trolley, that means there is a specific selection of items. This example is akin to the knapsack problem, a problem where a trader needs to carry items to sell, but can only carry a certain weight and he needs to maximise the profit, so naturally the trader should carry the lowest weighted items with the highest profit, therefore a rule can be put, minimise the weight and maximise the profit. In this question there isn't much to maximise and minimise, but we will put the assumption that the victim wanted to maximise the number of items, and minimise the weight.

Branch and Bound

Branch and Bound is an optimization technique that can be used in solving the knapsack problem, while it may not be the first option that comes to mind, it is still a viable solution. It works by creating a tree, where each node is a decision, and to further optimise the search tree, pruning or bounding is implemented to cut off useless nodes of the tree as early as possible. In the case of the knapsack problem, the decisions will be whether to include or not include an item. Since this approach is most likely going to be worse than brute force and is definitely worse than DP, we will not further explore it.

Brute force

Since there are only 4 items we can brute force the solution by getting all possible combinations of items and their respective sum of weights and choose the lowest weight with the highest item count. With 4 items, and since we can choose any number of items

from 1 to 4, we will get a total of 15 combinations, because $4C1 = 4$, $4C2 = 6$, $4C3 = 4$, and $4C4 = 1$ and the sum of all those results is 15.

Dynamic Programming

The brute force approach is not bad, but there is a faster way. Dynamic programming can help us solve this question in a more efficient manner. DP is generally described as dividing a big problem into smaller sub-problems, in this case the division is by finding out the combination for the lower weights, so if I know what is the best possible combinations of items for a bag of weight 11, I can figure out the best possible combination for a bag of weight 12 and so on and so forth until I get to 30.

Pseudocode

1. Declare a 2D array with $n + 1$ rows and $w + 1$ cols (n = # of items), (w = max weight)
2. Initialise all cells to 0
3. For every row
 - a. For every column
 - i. If the weight of the item on the current row is less than the weight of the column
 1. Initialise the current cell with the max of two other cells
($\text{cell}[i][j] = \max(\text{cell}[i-1][j], \text{cell}[i-1][j - \text{curr_weight}] + 1)$)
 - ii. Else
 1. Initialise the current cell to the value of the cell above it
($\text{cell}[i][j] = \text{cell}[i-1][j]$)
4. Declare selected items list
5. For every row from the last row
 - a. If the cell is unequal to the cell above it
 - i. Select the current item
 - ii. Shift the columns by the weight of the item
6. Return the selected items

Runtime complexity

Complexity: $O(n * w)$

The complexity is $O(n * w)$ where n is the number of items and w is the weight the trolley can carry, this comes from the main loop where all cells are looped over to get filled, the outer loop is $O(n)$, that is n items or n rows, and the inner loop is $O(w)$, that is w weights, or w columns. So that gives the main loop a complexity of $O(n * w)$, the processes done in the loops are done in constant time as they are just some arithmetic processes. The final loop that determines the items used only loops over the n rows, making its complexity $O(n)$, but this doesn't affect the overall time complexity.

Code

```
Knapsack

def knapsack(self, items, max_weight):
    # Initialize the table
    n = len(items)
    table = [[0] * (max_weight + 1) for _ in range(n + 1)]
    # Fill the table
    for i in range(1, n + 1):
        weight_i = items[i - 1]
        for w in range(1, max_weight + 1):
            if weight_i <= w:
                table[i][w] = max(table[i - 1][w], table[i - 1][w - weight_i] + 1)
            else:
                table[i][w] = table[i - 1][w]
    # Find the selected items
    selected_items = []
    w = max_weight
    for i in range(n, 0, -1):
        if table[i][w] != table[i - 1][w]:
            selected_items.append(i)
            w -= items[i - 1]
    # Return the selected items
    return selected_items[::-1]
```

Part 7: Almost there!

Discussion

This question provides 4 words with jumbled letters, apparently, the words contain a hidden message.

haTt
enPros
asH
eMvito

The letters don't seem to be jumbled in an orderly fashion, so it was most likely random, and since there is no pattern, the unjumbled words will most likely have to be brute forced. But a little detail makes everything easier, there are capital letters in the words, those are most likely the initial letters, meaning we know the words the starting letter of each word, and that shortens the letters that we have to permute to find the original word.

Frequency Analysis

Similar to a previous problem, frequency analysis can be used here to unscramble the words. By getting the frequency of the letters in the English alphabet and comparing it to that of the words, we can unscramble the letters. While this approach may work if implemented correctly, to implement correctly, however, it needs a lot of work to get it working properly and efficiently, and after all the work it may not produce the correct results.

Similar letters

A method that can be used is just looking up the words that have the same number of letters and the same letters, this can be a good approach. It is slightly simpler than the dictionary lookup method, but the problem is that it requires a whole dictionary and the dictionary needs to be narrowed down multiple times, each time differently according to each word, its length, and its letters.

Dictionary lookup

For this method, we just need parts of the dictionaries which makes this a more approachable, simpler, and easier method. We can get the permutations of the word and match every permutation with every word in the English dictionary, but that takes too long, so we can place the capital letter at the beginning of each word, and now we can match every permutation to every dictionary word that starts with that letter, and to make it even faster, we can get every dictionary word that starts with that letter and has the same length as the word we are matching. While not a very efficient approach, but one that works and will most definitely get the correct answer.

Pseudocode

1. Write all dictionary words to a list
2. Rearrange the words to place the capital letter first
3. Permute the words
4. Delete all words in the dictionary that are not the same length as the word
5. For every word in the list of words
 - a. For every word in the dictionary
 - i. If permuted word is equal to dictionary word
 1. Print word

Runtime complexity

To get the total complexity let's go over each part of the code.

Reading all paths and all words in the path is $O(n * m)$ where n is the number of dictionaries, and m is the number of words in each dictionary.

Reading all the words that we need to unscramble is $O(n)$ where n is the number of words, that is 4 in this case.

Rearranging all the words to have the capital letters be first consists of multiple operations, first looping over all the words and that is 4 words as mentioned before, and then finding the capital letters position, that is $O(m)$ where m is the number of letters in each word. Making the total complexity $O(n * m)$, m being the number of letters in each word, and n being the number of words.

Permuting all words takes $O(n * m!)$ where n is the number of the words, and m is the number of letters, this is the most complex operation of the whole code, because it takes factorial time.

Cutting down the dictionary to only have words of the same length takes $O(n * m)$ where n is the number of words, and m is the number of words in the dictionary.

Finally, the main code, it runs over all words, n words in total, then goes over all words in the dictionary, m words in total, and finally the strings are checked if they are equal or not which takes $O(k)$ time where k is the length of the word, since both words are equal in length already. Making the total complexity $O(n*m*k)$

The complexity of all the code is just the addition of the complexity of all the parts that were analysed above.

Code

```
def main():
    # Initialzie the file paths
    file_path = "Part 7/words.txt"
    word_paths = ["Tword.csv", "Pword.csv", "Hword.csv", "Mword.csv"]
    dictionary = []

    # Read the words in the paths and write to dictionary
    for path in word_paths:
        with open("Part 7/Dictionary/" + path, "r") as file:
            reader = csv.reader(file)

            rows = []

            for row in reader:
                cleaned_row = [element.strip() for element in row]
                if cleaned_row[0].isalpha():
                    rows.append(cleaned_row)

            dictionary.append(rows)

    # Read the words to unscramble
    with open(file_path, 'r') as file:
        words = file.read().split()

    # Rearrange the words to make the first letter the uppercase letter
    rearranged_words = [word[word.index(next(filter(str.isupper, word)))] +
                        word.replace(word[word.index(next(filter(str.isupper, word)))]
                        for word in words]

    # Put the words in a dictionary
    words = {}
    for word in rearranged_words:
        words[word] = []

    pm = Permute()

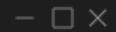
    # Permute the words and add the permutations to the dictionary
    for i in rearranged_words:
        permutations = (pm.permute(i))
        filtered_permutations = [perm for perm in permutations if perm[0] == i[0]]
        words[i] = filtered_permutations

    # Filter the dictionary
    for wk, index in zip(words.items(), range(len(words))):
        word, key = wk
        dictionary[index] = [w for w in dictionary[index] if len(w[0]) == len(word)]

    sm = StringMatching()

    # Find the permutations that match the words in the dictionary
    for wk, index in zip(words.items(), range(len(words))):
        word, key = wk
        found = False
        for k in key:
            k = k.lower()
            for w in dictionary[index]:
                if sm.brute_force(w[0], k):
                    print(w[0])
                    found = True
                    break
        if found:
            break
```

Permute



```
def permute(self, word):  
    # Base case  
    if len(word) == 1:  
        return [word]  
    # Initialize list of permutations  
    permutations = []  
    # Iterate through each character in the word  
    for i in range(len(word)):  
        # Extract the current character  
        current_char = word[i]  
        # Generate all permutations of the remaining characters  
        remaining_chars = word[:i] + word[i + 1:]  
        remaining_permutations = self.permute(remaining_chars)  
        # Add the current character to the beginning of each permutation  
        for perm in remaining_permutations:  
            permutations.append(current_char + perm)  
  
    return permutations
```


Part 8: Murder Suspect

Discussion

The question gave a list of each family members characteristics, relationship with Mr Marshall, and net worth which can be used to find one with the most significant motive

Name	Relationship	Character	Net worth (\$)
Jones Marshall	Son	Always rude to people, especially his father.	1M
Jenna Marshall	Daughter	The quiet one in the family.	700K
Peter Marshall	Brother	Animal lover.	50K
Penelope Marshall	Sister	Playful despite her old age.	500K
Will Marshall	Uncle	Retired army officer	10K

Multicriteria preference analysis

The Multicriteria Preference Analysis algorithm allows you to objectively evaluate and rank suspects based on objective criteria. By assigning weights to the criteria, you can reflect their relative importance in the decision-making process. This algorithm provides an approach to consider multiple dimensions of evidence and allows for a comprehensive assessment of potential suspects.

Machine Learning

To get the right answer to this question all the factors need to be considered, and to get an algorithmic approach that considers all factors is a hard task, especially for things like character descriptions. Machine Learning can prove helpful in solving this problem, if we get a trained ML model that understands human motives and can do analysis on their different attributes, then we can provide a solution for this question, unfortunately, building and training an ML model is very time consuming, and finding the data needed to train the model is very hard, even then if the model is trained, the accuracy might be very bad and the

results will be wrong. So while it may be a possible approach, it isn't very feasible.

Weight Approach

The weighted approach algorithm assigns weights to different information, such as relationship, personality, and net worth, and calculates a weighted score for each potential suspect. The suspect with the highest weighted score is considered the most likely culprit which allows for a more flexible and subjective evaluation of the suspects by incorporating subjective factors such as personality and relationship into the decision-making process. It recognizes that not all criteria or information may carry equal importance in determining the likelihood of someone being a murder suspect. This approach is much easier and simpler to implement compared to the other approaches, and if the weights of the values are tweaked right we can get a reasonably correct answer.

Pseudocode

1. Declare a list of family_members
2. Initialise the weights
3. Initialise max_weight as negative infinity
4. Initialise suspect as None
5. for each member in family_members
 - a. Calculate the weighted score for the current member
 - b. $\text{weight} = \text{member}['\text{net_worth}'] * \text{net_worth_weight} + \text{member}['\text{relationship_weight}'] * \text{relationship_weight} + \text{member}['\text{characteristic_weight}'] * \text{characteristic_weight}$
 - c. if weight is greater than max_weight
 - i. Update max_weight with the current weight
 - ii. Update suspect with the current member
6. Return suspect

Runtime complexity

Complexity: $O(n)$

The complexity is $O(n)$ where n is the number of family members. The algorithm iterates through each family member, resulting in a main loop with a complexity of $O(n)$.

After calculating the weighted score, the algorithm updates the maximum weight and the suspect if necessary. These operations are also constant time operations.

Finally, the algorithm returns the suspect, which is a constant time operation.

Therefore, considering the main loop complexity of $O(n)$ and the constant time complexity of the remaining operations, the overall runtime complexity is $O(n)$.

Code

```
Weighted Approach

def identify_murder_suspect(self, family_members):
    # Initialize the score and suspect
    max_score = float('-inf')
    suspect = None
    for member in family_members:
        # Calculate the score
        score = member['net_worth'] * self.net_worth_weight +
            member['relationship_weight'] * self.relationship_weight +
            member['characteristic_weight'] * self.characteristic_weight

        # Update the highest score and suspect
        if score > max_score:
            max_score = score
            suspect = member
    return suspect
```

Part 9: Story Ending

With all the clues we found, and the hints that have been already placed for us, we can narrow down the suspects. We first looked through the mansion for hints, then we found a book, and deciphered the note in it, that would be our first hint. Mr Marshall says that the person that is going to kill him is coming for him, hinting that this has been a long planned murder. Following that we looked in the shed to see what Mr Marshall could have carried in his trolley, that doesn't really point us in any directions, but it does give us one more note that needs to be deciphered, after figuring out the original message we know that there is a motive behind the killing.

With all these clues, we rule out some of the suspects, the uncle is an old distant relative, meaning that he probably has no reason to kill Mr Marshall as he won't make use of the money or anything else. The sister and brother have rather nice personalities, so they probably also have no motive to kill Mr Marshall. This leaves us with the daughter and son, the daughter is said to be the quiet one in the family, but the son is said to be rude, especially with the father, no assumptions can be made about the daughter's quietness, but we can definitely say that the son is not rude for no reason, making him the most likely suspect.

After these deductions, we confronted Jones Marshall, the son, and he admitted that he poisoned the father, it was like he knew that his intentions were clear, however, he didn't confess why he killed his father, but we think it is most likely because of previous problems that have occurred in the family.

After this long dinner, everyone needs a rest, we will be resting in our homes, while Jones Marshall rests in his prison cell. The police should take care of this from here.

Appendix

Semester 2 2022/2023	WIA2005 : ALGORITHMS ANALYSIS AND DESIGN GROUP CONTRACT
----------------------	--

A team of at most 6 students to

- Declare and identify individual strength
- Identify individual role in the team
- Agreed on meeting time, venue, communication means and approaches to arrives at any decisions
- Develop team / group social contact

Deliverable / To submit

- Project Report

Each member role and contract

Group Leader : Mujahed Yahia Murad Mohammed






Contract Item: As a Team we agree to	
• Participation	Participate actively in group discussions and meetings, and contribute to the final product to be submitted to the best of our ability.
• Communication	Communicate ideas and thoughts clearly and respect other ideas and thoughts without any objections.
• Meetings	Respect meetings time and adhere to them, and contribute effectively to the meeting topic.
• Conduct	Treat the team with respect and professionalism and to not show any forms of misconduct and disrespect towards any other member regardless of thoughts and beliefs.
• Deadlines	Respect deadlines and submit work in a timely manner, excuses that may interfere with deadlines are to be disclosed to the whole team before time
• Conflict	Resolve conflicts as a team internally and without any interference from third parties.

Clause

In any violation of the above, we agree

In the case of the violation of the terms above, the team agrees to work on the issues openly and without and to solve those issues and problems without any secrecy.

Please ensure that the items in the clause are effective and feasible.

No	Matric No	Name	Team Role	Signature
1	S2042189	Mujahed Yahia Murad Mohammed	Team Leader	
2	S2100683	Samer Abukhader	Team Member	
3	S2116735	Makrem Abdulkrim Ajmi Ziani	Team Member	
4	S2115641	Zalat Bara Sherif Ibrahim	Team Member	
5	17203116	Aiman Muhamad Hasan Talib	Team Member	

--	--

(Assessor:

Date Received:

)

Appendix B

FILA FORM – University of Malaya					
	FACTS	IDEAS	LEARNING ISSUES	ACTION	DATELINE
P1	What do we know about the task? Explore the mansion	What do we need to find out? Methods to explore the mansion		Who is going to do it? Makram	
	Phases Research Implementation Discussion Reporting	Practical Ideas Exploring the mansion relates to using graph algorithms to search graphs and trees, those same algorithms can be implemented here to solve this problem	Activities Group discussion & Research Group review and confirmation	Deadline 21/05	
P2	What do we know about the task? Crack a 3 digit lock	What do we need to find out? How to crack a 3 digit lock		Who is going to do it? Baraa	
	Phases Research Implementation Discussion Reporting	Practical Ideas Cracking a lock whether it be a physical or digital lock has its techniques that are well-known and can be used here to solve this problem.	Activities Group discussion & Research Group review and confirmation	Deadline 21/05	
P3	What do we know about the task? Finding the differences between 2 letters	What do we need to find out? How to find the differences between the letters		Who is going to do it? Makram	
	Phases Research Implementation Discussion Reporting	Practical Ideas Here we have a regular case of trying to match two different things and find the differences between them. We can use different comparison methods and so on to solve this problem	Activities Group discussion & Research Group review and confirmation	Deadline 21/05	

P4	What do we know about the task? Finding a book in a library	What do we need to find out? What are ways to find a book in an alphabetically sorted library	Who is going to do it? Samer
	Phases Research Implementation Discussion Reporting	Practical Ideas Usually this problem can happen in the real-world, if the library is alphabetically sorted, a human will start from somewhere and then move to the letter they need to be at using their knowledge of the alphabet	Activities Group discussion & Research Group review and confirmation Deadline 21/05
P5	What do we know about the task? Need to decipher a ciphered message	What do we need to find out? Cipher type and methods of deciphering it	Who is going to do it? Mujahed
	Phases Research Implementation Discussion Reporting	Practical Ideas The problem needs us to decipher a message, this problem can be a real-world problem, the methods to solving it would be to analyse the cipher and understand it, then learn how to decipher it and start deciphering it	Activities Group discussion & Research Group review and confirmation Deadline 21/05
P6	What do we know about the task? Find the items carried on the trolley	What do we need to find out? A way to figure out the items carried on the trolley	Who is going to do it? Mujahed
	Phases Research Implementation Discussion Reporting	Practical Ideas This is keen to the knapsack problem, which can be used in real life to maximise and minimise some number	Activities Group discussion & Research Group review and confirmation Deadline 21/05
P7	What do we know about the task?	What do we need to find out?	Who is going to do it?

	Unscramble a number of words		Figure out how to unscramble random word scrambles		Baraa	
	Phases Research Implementation Discussion Reporting	Practical Ideas Unscrambling words is a regular task that can face us in the real word, solving it may take time but it is a simple and easy task		Activities Group discussion & Research Group review and confirmation		Deadline 21/05
P8	What do we know about the task? Find the murder suspect		What do we need to find out? The motive of the murder suspect		Who is going to do it? Aiman	
	Phases Research Implementation Discussion Reporting	Practical Ideas Unravelling murder mysteries is not something that we do normally, but it certainly happens. To do so one needs to find clues and help solve the mystery		Activities Group discussion & Research Group review and confirmation		Deadline 21/05