

UNIVERSITI MALAYA

WIA2005 Algorithm Design and Analysis

Group Project: The Mystery of Marshall Mansion Murder

Name	Matric Number
HAZEEQ SYAKIRIN BIN AZAHAR	U2000687
MUHAMAD IZZUL IZZANI BIN ABU BAKAR	U2102733
MOHAMAD QHALISH BIN MOHD HAROMA	U2102865
MUHAMMAD ARIF EZUAN BIN MOHD FAUZI	U2102743
MUHAMMAD ADAM AIMAN BIN HELMI	U2001853
MUHAMMAD MUQRI QAWIEM BIN HANIZAM	U2000726

TABLE OF CONTENT

Introduction	4
Part 1: Who poisoned Marshall?	5
A. Discussion	5
B. Pseudocode	8
C. Running Time Complexity	8
D. Code	9
Part 2: Cracking the chest lock code	11
A. Description	11
B. Pseudocode	13
C. Running Time Complexity	13
D. Code	14
Part 3: Same but not identical	15
A. Discussion	15
B. Pseudocode	18
C. Running Time Complexity	18
D. Code	19
Part 4: Find that book.	21
A. Discussion	21
B. Pseudocode	23
C. Running Time Complexity	24
D. Code	24
Part 5: Secret message	28
A. Discussion	28
B. Pseudocode	31
C. Running Time Complexity	32
D. Code	33
Part 6: Find the next clue.	34
A. Discussion	34
B. Pseudocode	37
C. Running time complexity	37
D. Code	38
Part 7: Almost there!	40
A. Description	40
B. Pseudocode	42
C. Running Time Complexity	42
D. Code	44
Part 8: Murder Suspect	45
A. Discussion	45
B. Pseudocode	47
C. Running Time Complexity	47
D. Code	47
Part 9: Story Ending	51
Appendices	53
Appendix A - Group Contract	53
Appendix B - FILA form	56

References	62
Source Code	63

62
63

Introduction

"The Mystery of Marshall Mansion Murder" revolves around a captivating incident that occurred during a Christmas dinner at the esteemed Marshall Mansion. The protagonist, an acquaintance named Rivers, extends an invitation to the protagonist to join a private gathering at the mansion due to their recent move to the city and lack of Thanksgiving plans.

The mansion belongs to Mr. Phillip Marshall, a wealthy businessman renowned for his manufacturing enterprises across the United States. Rivers, who was adopted by Mr. Marshall at a young age, plays an active role in assisting him with the management of his business affairs. The dinner is attended by a close-knit group comprising Mr. Marshall's children, Jones and Jenna, his siblings Peter and Penelope, his uncle Will, and a few other intimate friends of the family, amounting to approximately 15 individuals.

The evening commences with a delightful feast, offering an abundance of delectable dishes. However, the ambiance is abruptly shattered when Mr. Marshall experiences a sudden medical emergency. Gasping for breath, he collapses from his chair, plunging the entire gathering into chaos and alarm. The protagonist rushes to Mr. Marshall's aid, attempting to provide immediate assistance until professional medical help arrives. Tragically, despite their efforts and the imminent arrival of an ambulance, Mr. Marshall ceases to breathe, leaving behind a scene of shock and disbelief.

The untimely demise of Mr. Marshall under mysterious circumstances forms the basis of the captivating story, thrusting the protagonist into an enthralling investigation. Amidst the atmosphere of suspicion and hidden motives, the protagonist becomes immersed in a quest to unravel the truth behind the enigmatic murder at Marshall Mansion. The story delves into a thrilling journey as the protagonist untangles the secrets, motives, and potential suspects connected to the haunting event, leading to an enthralling climax that will keep readers on the edge of their seats.

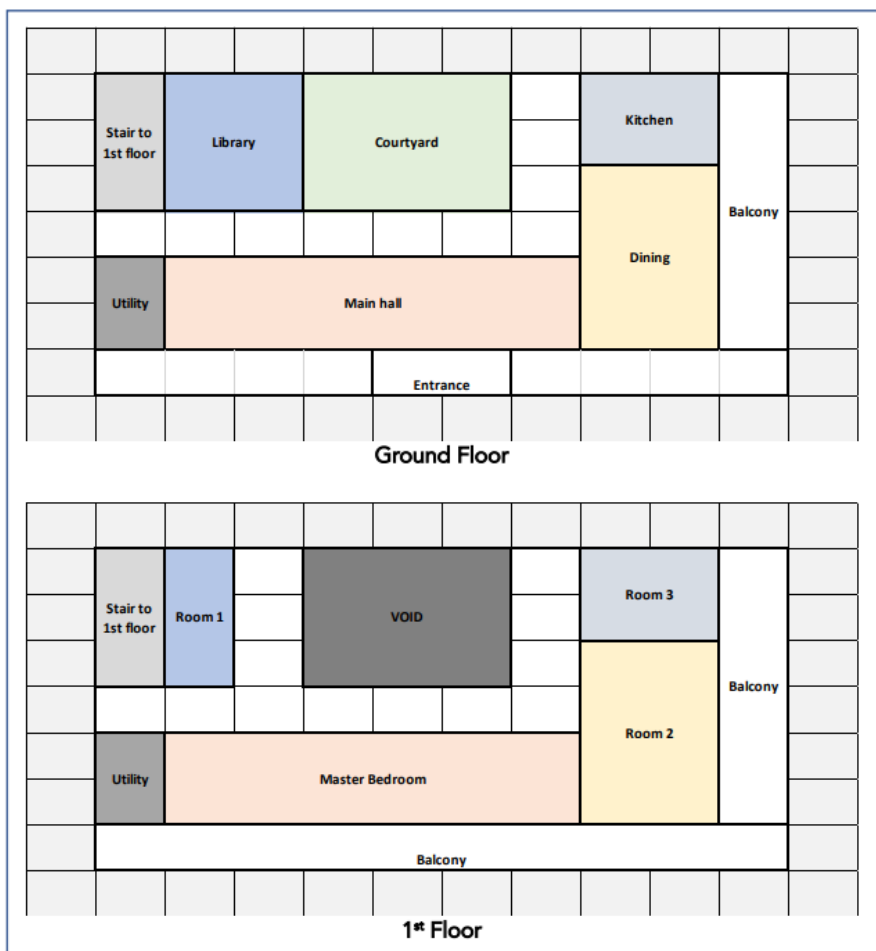
Part 1: Who poisoned Marshall?

A. Discussion

Description

A few moments before Mr Marshall passed away, he whispered, “I know who did this”. As a detective who works for the Police department in this city, I realized this was not an accident; instead, someone had murdered Mr Marshall.

After the mansion was closed for investigation and everyone had left, I started looking for clues. The mansion was a big building with many rooms over a huge land and a lake, and I managed to get a layout of the building (Figure 1) and the surrounding areas (Figure 2). The main building must have some clues, so I will search all the rooms.



Problem:

How to search all the rooms in the building without missing any?

For this part of the problem, there is a clue to find who murdered Mr Marshall in one of the rooms in the main building. In order to find the clue, all the rooms in the main building need to be searched. There are three possible algorithm that can be implemented in order to search all rooms in the mansion which is Linear Search, Breadth-First Search and Depth-First Search.

Assumption :

1. The search will start at the entrance of the ground floor.
2. The balconies, courtyard and stair will be considered as rooms.
3. There are three balconies and two utilities will be labeled as balcony 1, balcony 2, balcony 3, utility 1 and utility 2.
4. All the rooms will be searched although clues have been found.

Based on the assumption, there are three algorithms suitable to solve the problem which are Linear Search, Breadth-First Search and Depth-First Search.

- Linear search

Linear search is a simple searching algorithm used to find a specific element in a list or array. It starts at the beginning of the list and checks each element sequentially until it either finds the target element or reaches the end of the list. It is called "linear" because it performs a linear comparison of elements in a sequence. In this case, we search through the list of all the rooms in the mansion from the starting until the end of the list.

- BFS (Breadth-First Search)

BFS is a graph traversal algorithm that explores all the vertices of a graph in breadth-first order, meaning it visits all the vertices at the same level before moving on to the next level. It starts at a given source vertex and explores all of its neighbors first before moving to the next set of neighbors. Starting from a given room, BFS would visit all the neighboring rooms first, then move on to the neighbors of those rooms, and so on. BFS is useful for finding the shortest path between two vertices because it guarantees that the shortest path will be discovered first when exploring the graph level by level.

- DFS (Depth-First Search)

DFS is another graph traversal algorithm that explores all the vertices of a graph, but in depth-first order. It starts at a given source vertex and explores as far as possible along each branch before backtracking. In other words, it explores one branch completely before moving on to the next branch. DFS uses a stack or recursion to keep track of vertices and their exploration order. In this case, the graph is similar to the BFS but instead of exploring all the vertices on the same level before moving to the next level, it will explore one branch completely till the end before backtracking and explore to the other branch.

Algorithm	Linear Search	BFS	DFS
Complexity	Simple	Moderate	Moderate
Data structure	List	Unweighted graph	Unweighted graph
Advantage	<ul style="list-style-type: none"> - Easy to implement as it require minimal and simple code - Low time complexity which is $O(n)$ as it only search through list 	<ul style="list-style-type: none"> - Guarantees finding the shortest path between two vertices in an unweighted graph - More systematic exploration as it prioritize level by level exploration 	<ul style="list-style-type: none"> - Use memory more efficiently than BFS, as it explores deeply before backtracking. - Beneficial for problems that require backtracking or pruning of search spaces
Limitation	Require visiting each room sequentially. Since the room in the mansion is not in linear structure and has multiple routes to search all the rooms in the mansion, this algorithm is not suitable for efficiently exploring all the rooms.	Since there are multiple rooms connecting to each other, it can potentially result in redundant visits to some rooms as some rooms will be explored multiple times.	Will explore to farthest room in the 1st floor before backtracking and return to the ground floor again without finishing exploring the 1st floor. This would be time consuming and inefficient

Among the three algorithms, BFS is the most suitable algorithm for solving the problem of exploring all the rooms in the mansion. While linear search may have shorter code and time complexity, it is limited to handling simple lists. In the case of the mansion, where the rooms are not in a sequential order, linear search would require sorting the list of rooms first, making the exploration less efficient.

On the other hand, DFS is not the optimal choice as it explores the mansion by going as deep as possible before backtracking. This backtracking process can significantly increase the time taken to explore all the rooms compared to BFS, which systematically explores all nearby rooms before moving on to farther rooms.

BFS, on the other hand, is well-suited for exploring all the rooms in an efficient manner. It explores the mansion in a breadth-first manner, visiting all the neighboring rooms of a given room before moving on to the next level of rooms. This ensures that all reachable rooms are visited and can be particularly useful when there is a need to find the shortest path or determine connectivity between rooms.

In summary, BFS is selected as the most suitable algorithm for exploring all the rooms in the mansion due to its ability to efficiently visit all reachable rooms, handle non-sequential lists, and ensure comprehensive coverage of the entire mansion.

B. Pseudocode

1. Define the adjacency list representing the mansion's rooms.
2. Define the BFS function that takes the starting room as input.
3. Create an empty set to keep track of visited rooms.
4. Create an empty queue.
5. Enqueue the starting room with "No clue" status.
6. Run the BFS algorithm.
7. While the queue is not empty:
 - Dequeue a room and its clue status from the queue.
8. Check if the room has not been visited before.
9. If the room has not been visited:
 - Mark the room as visited.
10. Get the neighbors of the current room from the adjacency list.
11. Check the clue status based on the current room's clue status.
12. Print the visited room and its clue status.
13. Enqueue the neighbors of the current room with the updated clue status.
14. Perform BFS starting from the "entrance" on the ground floor.

C. Running Time Complexity

The running time complexity of the provided code is dependent on the number of rooms and the connections between them in the mansion. Let's denote the number of rooms as "n".

In the worst-case scenario, where all rooms are interconnected, the BFS algorithm will visit each room exactly once. Since each room is added to the visited set once and all its neighboring rooms are added to the queue, the time complexity for visiting each room and its neighbors is $O(1)$.

Since the problem required searching all the rooms in the mansion, the best-case and average-case will be similar to the worst-case scenario. The code will not stop although clues have been found in one of the rooms until all the rooms in the mansion have been explored.

Therefore, the overall time complexity of the BFS algorithm in this code is $O(n)$, assuming that accessing the adjacency list and performing operations on the queue take constant time.

D. Code

BFS (Breadth-First Search)

```
from collections import deque

# Define the adjacency list representing the mansion's rooms
adjacency_list = {
    "entrance": {"neighbors": ["main hall"], "clue": False},
    "main hall": {"neighbors": ["utility", "dining"], "clue": False},
    "dining": {"neighbors": ["balcony", "main hall", "kitchen"], "clue": False},
    "balcony": {"neighbors": ["dining", "kitchen"], "clue": False},
    "kitchen": {"neighbors": ["balcony", "dining", "courtyard"], "clue": False},
    "courtyard": {"neighbors": ["kitchen", "library"], "clue": False},
    "library": {"neighbors": ["courtyard", "stair"], "clue": True},
    "utility": {"neighbors": ["main hall"], "clue": False},
    "stair": {"neighbors": ["library", "room 1", "utility 2"], "clue": False},
    "room 1": {"neighbors": ["stair"], "clue": False},
    "utility 2": {"neighbors": ["stair", "master bedroom", "balcony 3"], "clue": False},
    "master bedroom": {"neighbors": ["utility", "room 2", "balcony 3"], "clue": False},
    "room 2": {"neighbors": ["master bedroom", "room 3", "balcony 2", "balcony 3"], "clue": False},
    "room 3": {"neighbors": ["room 2", "balcony 2"], "clue": False},
    "balcony 2": {"neighbors": ["room 3", "room 2", "balcony 3"], "clue": False},
    "balcony 3": {"neighbors": ["utility", "master bedroom", "room 2", "balcony 2"], "clue": False},
}

def bfs(start_room):
    visited = set()
    queue = deque([(start_room, "No clue")])

    while queue:
        room, clue_status = queue.popleft()
        if room not in visited:
            visited.add(room)
            neighbors = adjacency_list[room]["neighbors"]
            if adjacency_list[room]["clue"]:
                clue_status = "Found clue!!!"
            else:
                clue_status = "Found nothing...."
            print("Visited:", room, "-->", clue_status)
            queue.extend((neighbor, clue_status) for neighbor in neighbors)
```

```
# Perform BFS starting from the "entrance" on the ground floor
print("Begin searching:")
bfs("entrance")
```

Output:

```
Begin searching:
Visited: entrance --> Found nothing....
Visited: main hall --> Found nothing....
Visited: utility --> Found nothing....
Visited: dining --> Found nothing....
Visited: balcony --> Found nothing....
Visited: kitchen --> Found nothing....
Visited: courtyard --> Found nothing....
Visited: library --> Found clue!!!
Visited: stair --> Found nothing....
Visited: room 1 --> Found nothing....
Visited: utility 2 --> Found nothing....
Visited: master bedroom --> Found nothing....
Visited: balcony 3 --> Found nothing....
Visited: room 2 --> Found nothing....
Visited: balcony 2 --> Found nothing....
Visited: room 3 --> Found nothing....
```

Part 2: Cracking the chest lock code

A. Description

I was looking in the library; there were stacks of books on the table and hundreds of books on the shelves around the room. Apparently, Mr Marshall does a lot of reading and spends much time in this room. I found an old safe on one of the shelves, and after close inspection, it requires a 3-digit number combination to open it (Figure 2). I look around for the code, but I guess I must figure out how to crack this safe.

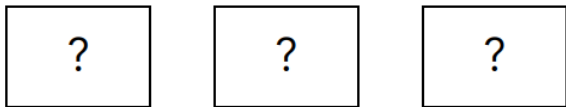


Figure 3: 3-digit number combination for the old safe

Problem:

What is the possible number combination for the lock?

For this part, we needed to find a way to open the old safe. To do that, we need to give the correct number combination. Since we don't have any clue on what the actual code is, our option is to try multiple combinations until the safe opens.

Assumptions:

1. Since the actual 3-digit number combination is not given, we'll assume that it's a randomized number between 0 to 999.

In order to open the safe, we'll have to try a lot of possible combinations. There are 3 ways to do this. First is by exhausting all combinations using brute force algorithms. The second way is to use a binary search algorithm to narrow down the possible combinations. The final way is through a randomized guessing algorithm where the program will, as the name suggests, guess using random combinations and it'll keep on guessing until the safe is open.

Algorithm	Brute Force Algorithm	Binary Search Algorithm	Randomized Guessing Algorithm
Introduction to Algorithm	A straightforward method of solving a problem that relies on sheer computing power and trying every possibility rather than advanced techniques to	A method used to find the position of a specific value, in this case the correct code in a sorted array. It works with the principle of divide and conquer.	A method that is self explanatory. The program will randomly pick a number and try it. If it fails, it'll try a different number

	improve efficiency. In this case, we'll try every possible lock combination starting from 000 to 999.		until the safe is open.
Advantages	<ul style="list-style-type: none"> - A guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem. - A generic method and not limited to any specific domain of problems. - Ideal for solving small and simpler problems. 	<ul style="list-style-type: none"> - Faster especially for large arrays. - More efficient than other searching algorithms. - Well suited for searching large datasets that are stored in external memory. 	<ul style="list-style-type: none"> - Utilizes random guessing strategy. This allows it to potentially crack the code in fewer attempts on average. - Guarantees finding the correct combination. - Ideal for solving small and simpler problems.
Limitations	<ul style="list-style-type: none"> - Inefficient. The time complexity goes above $O(n!)$ - Relies more on compromising the power of a computer system for solving a problem than on a good algorithm design. - Slow and not constructive or creative. 	<ul style="list-style-type: none"> - Data must be ordered. - In this part, the knowledge of what the code is needed in order for this algorithm to work. 	<ul style="list-style-type: none"> - Inefficient. The larger the search space, the longer it takes to finish. - It might guess the same number more than once. - Uses a lot of memory to store the numbers that have been used to guess.
Modifications	Instead of letting the algorithm go through all the possible combinations, it will stop when it finds the correct one. This can save some time.	Initial guess is randomized. This will allow it to potentially crack the code in fewer attempts on average	Create a list of guessed numbers to prevent the program from guessing the same number multiple times.

Chosen algorithm is the Randomized Guessing Algorithm. It's the 2nd fastest algorithm and it's more realistic and practical to use in this situation. It does not require the prior knowledge

of what the combination is like Binary Search Algorithm and it uses the element of randomness which allows it to crack the code in fewer attempts than brute force algorithm.

B. Pseudocode

1. Pick a random number between a given range.
2. Try the number.
3. If the lock opens, the current combination is the correct one.
4. If it fails, repeat step 1-3 until you find the correct combination.

C. Running Time Complexity

The running time complexity of the algorithm is $O(1000)$ which can be simplified as $O(1)$. This is because the upper bound is fixed. The time it takes to find the correct code does not depend on the input size and remains constant regardless of the value of the variable "code". Compared to the other two algorithms, the Randomized Guessing Algorithm shares the same time complexity as Brute Force Algorithm. But the randomness in the Randomized Guessing Algorithm makes it better. The time complexity of the Binary Search Algorithm is $O(\log n)$. It's lower but as mentioned before, the algorithm is not suitable to be used in this situation.

Best case happens when the first attempt correctly guesses the desired solution which can be denoted as $O(1)$.

The average case is proportional to the number of possibilities. In this case, the search space is 0 to 999 (1000). Each guess has an equal probability of being the correct code, so on average, the code is expected to be cracked after trying half of the possibilities. Therefore, average case is $O(500)$ which simplifies to $O(1)$.

Worst case happens after guessing all possible solutions. In this case, $O(1000)$ which is $O(n)$.

D. Code

```
import random
print("Randomized Guessing Algorithm")
code = random.randint(0,999)
guess = random.randint(0,999)
guessed = []
while(guess!=code):

    guessed.append(guess)
    while guess in guessed:
        guess=random.randint(0,999)

print("Case cracked. Code is "+str(code).zfill(3))
```

Output

```
Randomized Guessing Algorithm
Case cracked. Code is 601
```

Part 3: Same but not identical

A. Discussion

For part 3, we need to identify differences between two letters that seem to be identical but are actually not. Those differences are needed in order to find the next clue for part 4. The letters are as below:

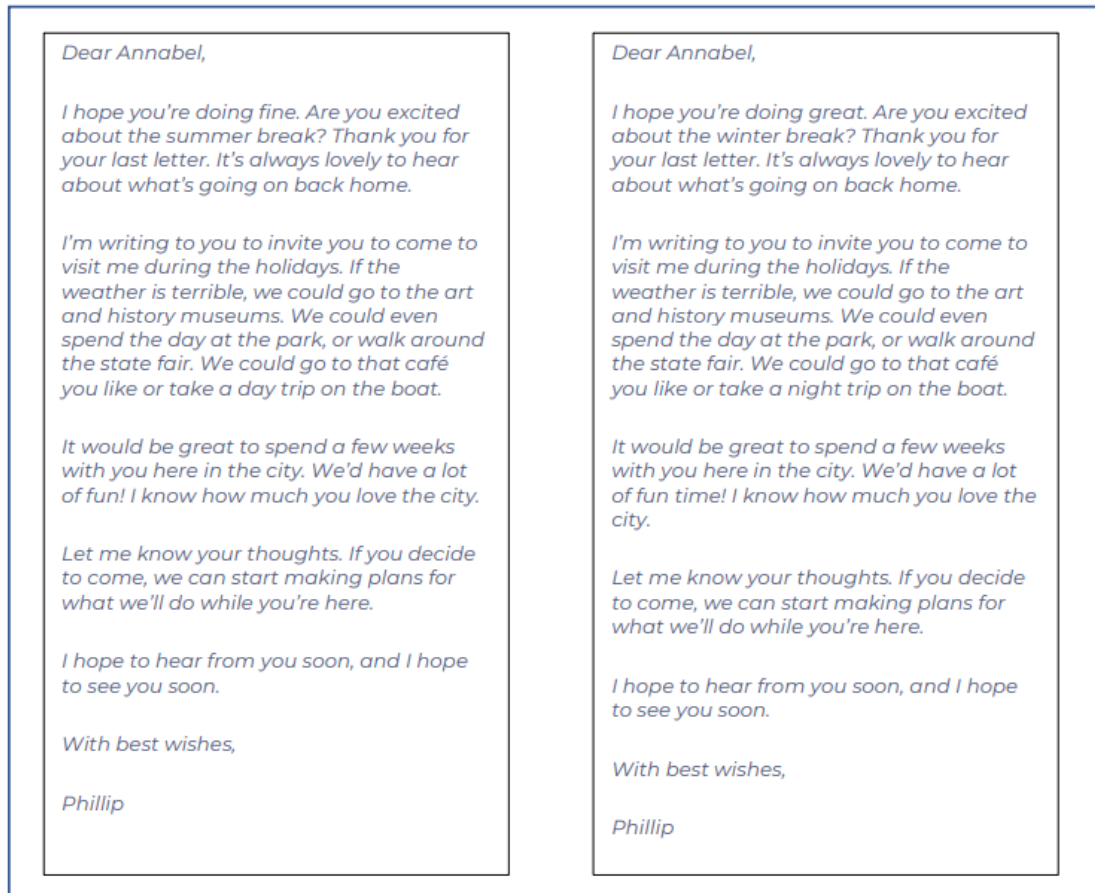


Figure 4: The two letters from the safe.

The problem stated is quite straight forward as we need to point out every single differences detected between these two letters. Before we proceed in finding the suitable algorithms that can be used to solve the problem, we need to properly specify the problem at hand. Thus, our assumptions for the problem are as below:

Assumptions

1. Any differences in spelling will be considered as different words.
2. The order of the different words are not important. Only the words are compulsory.

Based on the assumptions above, we concluded that it could be solved using the Longest Common Subsequences Algorithm as the algorithm works best for finding differences between two sequences. Other than that, there are also other algorithms which could be used to solve the problem, which are Merge Sort and quick Sort. In

order to choose the best algorithm, we will do a comparison check in order to find the best one.

Algorithm	Merge Sort	Quick Sort	Longest Common Subsequences
Introduction to Algorithm	Merge sort is a recursive sorting algorithm that operates by dividing the unsorted list into smaller sublists, sorting them individually, and then merging them back together to obtain a sorted result.	Quick sort follows a divide-and-conquer approach. It starts by selecting a pivot element from the array and partitioning the other elements into two subarrays, according to whether they are less than or greater than the pivot. The pivot is then in its final sorted position.	The LCS (Longest Common Subsequence) algorithm is a method that finds the longest shared sequence between two or more sequences. It works by comparing elements and creating a table to track the common elements.
Advantages	<ul style="list-style-type: none"> - Provide stable and predictable performance. - Guarantees a time complexity of $O(n \log n)$ in all cases, making it efficient for large datasets. - Allows for accurate identification of differences while maintaining the overall structure of the letters. 	<ul style="list-style-type: none"> - Has an average-case time complexity of $O(n \log n)$, making it efficient for most practical scenarios. - In-place partitioning and recursive nature reduce the need for additional memory. 	<ul style="list-style-type: none"> - It efficiently finds the longest common subsequence, which helps identify the common elements between the two letters. - Considering the elements that are not part of the common subsequence, can identify the differing elements. - Allows for flexible comparisons, as it does not require the elements to be in the same order. - LCS can handle cases where the differences are not contiguous.
Limitations	<ul style="list-style-type: none"> - It only compares corresponding elements during the merging process. - Does not provide 	<ul style="list-style-type: none"> - It does not consider higher-level linguistic structures or semantic meaning. - It treats each element 	<ul style="list-style-type: none"> - The time and space complexity of the LCS algorithm can be relatively high, especially for larger inputs.

	<p>information about the positions or context of the differences within the original letters.</p> <ul style="list-style-type: none"> - It can only determine which elements are different between the two letters and provide a sorted list of those differences. 	<p>independently without considering the surrounding context.</p> <ul style="list-style-type: none"> - If there is misalignment or differences in structure between the two texts, the algorithm may not provide accurate results. - When dealing with large texts, the algorithm's performance may degrade. 	<ul style="list-style-type: none"> - It may not scale well for extensive analyses or comparisons. - It does not offer insights into the specific additions, deletions, or modifications needed to convert one letter into another. - It does not highlight the exact character differences or their locations. - It does not directly provide information about the specific differences or modifications required to transform one letter into another.
--	--	--	--

Based on the assumptions above, we concluded that it could be solved using the Longest Common Subsequences Algorithm. But, in order to apply the algorithms into the problem, some modifications need to be done. Those are the modifications that need to be done.

1. The modified LCS algorithm need to be modified so it can record the differences between the two letters. Instead of solely focusing on common elements, it tracks the elements that are not part of the common subsequence.
2. A backtracking step will be introduced to trace the table and identify the differing elements. This involves examining the values in the table and determining the direction to move (diagonal, up, or left) based on the maximum value. When encountering non-matching elements, they are considered differences.
3. The algorithm identifies differences and they will be marked or stored in a separate list or data structure. This allows for easy access and further processing of the identified differences.

4. During the backtracking step, the modified LCS algorithm compares elements from the original letters to determine differences. It considers the positions of elements and their respective order to accurately identify differing elements.

B. Pseudocode

1. Create a function ``lcsCompare`` that takes ``arr1``, ``arr2``, ``differences1``, and ``differences2`` as input.
2. Implement ``initializeTable`` to create a table and initialize its first row and column.
3. Develop ``fillTable`` to populate the table based on the elements of ``arr1`` and ``arr2``.
4. Create ``backtrack`` to trace the table and identify the differences between ``arr1`` and ``arr2``.
5. Implement ``markDifference`` to add non-matching elements to the respective difference lists.
6. Inside ``lcsCompare``, call ``initializeTable``, ``fillTable``, and ``backtrack`` in sequence.
7. The differences will be stored in ``differences1`` and ``differences2`` for further use.

C. Running Time Complexity

The best case occurs when the two input arrays (the letters) have no differences and are identical to each other. In this case, the LCS algorithm will fill the entire table with match values, so there will be no differences to be backtracked and printed. The time complexity for the best case is $O(m*n)$, where m and n represent the lengths of the input arrays.

On the other hand, for the average case scenario, it occurs when the two input arrays (the letters) have some common elements and some differences. The time complexity for the average case is also $O(m*n)$ because, in general, the algorithm needs to fill the entire table to identify the longest common subsequence. The backtracking and printing steps will take additional time proportional to the length of the LCS.

Lastly, in the worst case scenario, it will occur when the two input arrays (the letters) have no common elements, and every element is different. In this case, the LCS algorithm will have to compare every element of array 1 with every element of array 2. Thus, resulting in filling the entire table with mismatch values. The time complexity for the worst case is $O(m*n)$, where m and n represent the lengths of the input arrays.

D. Code

```
def lcsCompare(arr1, arr2, differences1, differences2):
    m = len(arr1)
    n = len(arr2)

    # Initialize the table with zeros
    table = [[0] * (n+1) for _ in range(m+1)]

    # Fill the table
    for i in range(1, m+1):
        for j in range(1, n+1):
            if arr1[i-1] == arr2[j-1]:
                table[i][j] = table[i-1][j-1] + 1
            else:
                table[i][j] = max(table[i-1][j], table[i][j-1])

    # Backtrack to find the differences
    i = m
    j = n
    while i > 0 and j > 0:
        if arr1[i-1] == arr2[j-1]:
            i -= 1
            j -= 1
        elif table[i-1][j] >= table[i][j-1]:
            markDifference(arr1[i-1], None, differences1, differences2)
            i -= 1
        else:
            markDifference(None, arr2[j-1], differences1, differences2)
            j -= 1

    while i > 0:
        markDifference(arr1[i-1], None, differences1, differences2)
        i -= 1
    while j > 0:
        markDifference(None, arr2[j-1], differences1, differences2)
        j -= 1

def markDifference(element1, element2, differences1, differences2):
    if element1 is not None:
        differences1.append(element1)
    if element2 is not None:
        differences2.append(element2)

def store_essays():
```

```

essays = []
for i in range(2):
    essay = []
    print(f"Enter your {i + 1} essay. Press Enter on an empty line to finish.")
    while True:
        line = input()
        if line == "":
            break
        words = line.split()    # Split the line into words
        essay.extend(words)     # Add the words to the essay list
    essays.append(essay)
return essays

# Example usage
essays_array = store_essays()
arr1 = essays_array[0]
arr2 = essays_array[1]

differences1 = []
differences2 = []

lcsCompare(arr1, arr2, differences1, differences2)

print("Differences in Letter 1:")
print(' , '.join(differences1))
print("\nDifferences in Letter 2:")
print(' , '.join(differences2))

```

Output

```

Differences in Letter 1:
fun! , day , summer , fine.

Differences in Letter 2:
time! , fun , night , winter , great.

```

Part 4: Find that book.

A. Discussion

The different words found between the two letters sounded like a book title. I look around the library, there are hundreds of books here, but thankfully they are sorted alphabetically.

Problem:

How to find the book quickly?

In part 4, the task is to find the possible book title in the library in the quickest time from the different words found between the two letters in Part 3. There are hundreds of books in the library and they are sorted alphabetically. As there is no specification required to solve this problem, we came up with our own assumptions in order to help us solve this problem. Our assumption are as follow:

- The book title is derived from different words found in the second letter.
- The unique words 'Great', 'Winter', 'Night', and 'Fun time' are the potential candidates for the book title.

For the algorithm that can be used to solve this problem, there are few notable algorithms that we suggest could be used to solve this problem. Those algorithms are binary search, brute force algorithm and Rabin-Karp algorithm. To choose the best algorithm for this problem, we compared each one of these algorithms in order to find the perfect one.

	Binary Search Algorithm	Brute Force Algorithm	Rabin-Karp Algorithm
Introduction to Algorithm	Binary search is a divide-and-conquer algorithm that can be used to search for a value in a sorted list. It works by repeatedly dividing the list in half and searching the smaller half until the value is found.	The brute force algorithm is a simple algorithm that can be used to search for a value in a list. It works by simply comparing the value to each item in the list until it is found.	The Rabin-Karp algorithm is a hash-based algorithm that can be used to search for a value in a list. It works by creating a hash value for the value that we are searching for. The hash value is a unique identifier for the value, and it can be used to quickly compare the value to the items in the list.
Advantages	1. A very efficient algorithm. It can find the value in a sorted list in logarithmic time.	1. A very simple algorithm. It is easy to understand and implement.	1. It can find the value in a list in linear time. This means that the time it takes to find the

	<p>This means that the time it takes to find the value increases logarithmically as the size of the list increases.</p> <p>2. A very accurate algorithm. It will always find the value in the list, if it is present.</p>	<p>2. Can be used to search for any value in any type of list.</p> <p>3. Can be efficient for small lists.</p>	<p>value increases linearly as the size of the list increases.</p> <p>2. A very accurate algorithm. It will always find the value in the list, if it is present.</p> <p>3. A very robust algorithm. It is not affected by changes in the case of the letters, or by small changes in the value.</p>
Limitations	<p>1. Requires the list to be sorted before it can be used. This can be a time-consuming process, especially for large lists.</p> <p>2. Binary search is not very efficient for small lists.</p>	<p>1. Inefficient for large lists. This is because the algorithm needs to compare the value to each item in the list. For a list with n items, the brute force algorithm will need to make n comparisons.</p> <p>2. Not suitable for all types of lists. It can only be used for lists where the value is known to be present. For lists where the value is not known to be present, the brute force algorithm will not be able to find the value.</p> <p>3. Can be computationally expensive. The algorithm needs to compare the value to each item in the list. For large lists, this can take a significant amount of time.</p>	<p>1. Relies on a good hash function to generate the hash values for the text and pattern strings. If the hash function is not good, then the algorithm may not be able to find the pattern in the text.</p> <p>2. Can be inefficient for long patterns. This is because the algorithm needs to compute the hash value for the entire pattern each time it compares the pattern to the text.</p> <p>3. Can produce false positives. This occurs when the hash value of the pattern matches the hash value of a substring of the text that is not the pattern. To reduce the number of false positives, a good hash function should be used.</p>

So, we decided to use the binary search algorithm. The books in the library are sorted alphabetically. This allows binary search to quickly narrow down the search space by repeatedly dividing it in half. This significantly reduces the number of comparisons needed compared to brute force, where each title would need to be checked individually. On the other hand, the Rabin-Karp algorithm is primarily used for pattern matching in strings rather than searching in a sorted list. It involves hashing and comparing patterns, which may not be as efficient or appropriate for searching book titles in a sorted library.

But in order for us to use binary search algorithms, we need to apply some modifications to the binary search algorithm so that it can be used to solve this problem. The modification will be as follows:

- In an original binary search algorithm, the goal is to find a single target element in a sorted list. The algorithm compares the target element with the middle element of the list and adjusts the search range accordingly. It continues to divide the search range in half until the target element is found or the search range is exhausted.
- Instead of searching for a single target element, we are searching for multiple target words within the book titles. The modification involves using the `all()` function and a list comprehension to check if all the target words are present in the current book title. This condition is evaluated within the binary search loop.

B. Pseudocode

1. Initialize the low and high pointers to 0 and the length of the list, respectively.
2. While the low pointer is less than or equal to the high pointer:
 - a. Calculate the middle pointer by taking the average of the low and high pointers.
 - b. Compare the value to the item at the middle pointer.
 - i. If the value is equal to the item at the middle pointer, then the value has been found and the algorithm returns the middle pointer.
 - ii. If the value is less than the item at the middle pointer, then the value must be in the lower half of the list. In this case, the low pointer is set to the middle pointer + 1.
 - iii. If the value is greater than the item at the middle pointer, then the value must be in the upper half of the list. In this case, the high pointer is set to the middle pointer - 1.
3. Return -1 if the value is not found.

C. Running Time Complexity

For binary search algorithms, the running time complexity is generally $O(n \log n)$ where n represents the number of books in the library.

For the best case, the target words are found at the beginning of the list. In this case, the algorithm only needs to compare the target words to the first book in the list. This takes constant time, $O(1)$.

For the average case, the target words are randomly distributed throughout the list. In this case, the algorithm will need to compare the target words to about half of the books in the list. This takes a running time complexity of $O(\log n)$.

For the worst case, the target words are not found in the list. In this case, the algorithm must compare the target words to every book in the list. This takes a running time complexity of $O(n)$.

D. Code

```
def binary_search(target_words, book_list):  
  
    # sort the books alphabetically  
    book_library.sort()  
  
    start = 0  
    end = len(book_list) - 1  
  
    while start <= end:  
        mid = (start + end) // 2  
        mid_title = book_list[mid].lower()  
  
        if all(word.lower() in mid_title for word in target_words):  
            return mid  
        elif mid_title < target_words[0].lower():  
            start = mid + 1  
        else:  
            end = mid - 1  
  
    return -1  
  
def find_book(target_words, book_library):  
    lower_target_words = [word.lower() for word in target_words]  
  
    for index, title in enumerate(book_library):
```



```

        lower_title = title.lower()
        if all(word in lower_title for word in lower_target_words):
            print(f"Book containing all target words [ {'',
''.join(target_words)} ] found : '{title}', Index : {index}")
            return

    print(f"Book containing all target words '{',
''.join(target_words)}' not found in the library.")

def search_books(target_words, book_library):
    find_book(target_words, book_library)

# library book list
book_library = [
    "The Alchemist", "To Kill a Mockingbird", "The Great Gatsby",
    "1984", "Harry Potter and the Sorcerer's Stone",
    "The Lord of the Rings", "The Hunger Games", "The Catcher in
the Rye", "The Da Vinci Code", "Gone Girl",
    "The Girl With the Dragon Tattoo", "The Book Thief", "The
Curious Incident of the Dog in the Night-Time", "The Martian", "Where
the Crawdads Sing",
    "The Hitchhiker's Guide to the Galaxy", "The Kite Runner",
    "The Help", "The Fault in Our Stars", "Me Before You",
    "The Book of Negroes", "The Girl on the Train", "The Woman in
the Window", "The Shack", "Americanah",
    "Educated", "Where the Wind Leads", "The Nightingale", "The
Alice Network", "The Guernsey Literary and Potato Peel Pie Society",
    "The Seven Husbands of Evelyn Hugo", "The Midnight Library",
    "The Giver of Stars", "Pachinko", "The Four Winds",
    "The Last Days of Ptolemy Grey", "The Book of Longings", "The
Lincoln Highway", "Malibu Rising", "The Thursday Murder Club",
    "The House in the Cerulean Sea", "The Ministry for the
Future", "Project Hail Mary",
    "The Great Winter Night Time", "Don Quixote", "War and
Peace", "Ulysses", "Pride and Prejudice", "In Search of Lost Time",
    "Great Expectations", "Adventures of Huckleberry Finn",
    "Crime and Punishment", "Moby-Dick", "Hamlet", "The Odyssey",

```

"The Iliad", "The Adventures of Tom Sawyer", "Little Women",
"Jane Eyre", "The Scarlet Letter", "The Adventures of Don Quixote", "The Metamorphosis", "The Grapes of Wrath", "Things Fall Apart", "The Color Purple",
"The Handmaid's Tale", "The God of Small Things", "The Time Traveler's Wife", "Divergent", "The Secret History", "Animal Farm", "The Diary of a Young Girl",
"The Little Prince", "Romeo and Juliet", "The Chronicles of Narnia", "Fahrenheit", "The Giver",
"Charlotte's Web", "Of Mice and Men", "Wuthering Heights", "Night", "Gone With the Wind", "The Picture of Dorian Gray", "Brave New World", "Les Miserable",
"Memoirs of a Geisha", "The Secret Garden", "A Christmas Carol", "The Adventures of Tom Sawyer",
"Ender's Game", "One Hundred Years of Solitude", "A Tale of Two Cities", "The Outsiders", "Anne of Green Gables", "Winnie The Pooh",
"A Thousand Splendid Suns", "Life of Pi", "Tuesday With Morrie", "The Count of Monte Cristo", "Catch-22", "Anna Karenina", "Flowers for Algernon",
"Slaughterhouse-Five", "The Old Man and the Sea", "Frankenstein", "MacBeth", "Lolita", "Siddhartha", "Little House Series",
"A Tree Grows in Brooklyn", "A Clockwork Orange", "Uncle Tom's Cabin", "The Stand", "Atlas Shrugged", "All Quiet on the Western Front",
"The Poisonwood Bible", "The Brothers Karamazov", "The Good Earth", "I Know Why the Caged Bird Sings", "A Wrinkle in Time",
"Dracula", "Matilda", "Sense and Sensibility", "The Perks of Being a Wallflower", "Complete Tales and Poems",
"Fountainhead", "Where the Red Fern Grows", "The Princess Bride", "East of Eden", "The Lovely Bones", "Charlie and the Chocolate Factory", "Watership Down",
"The Five People You Meet in Heaven", "A Prayer for Owen Meany", "Rebecca", "Angel's Ashes", "Perfume", "The Bell Jar", "The Call of the Wild", "Dune", "Bridge of Terabithia", "Water for Elephants", "The Divine Comedy",
"A Midsummer Night's Dream", "The Three Musketeers", "The Name of the Rose", "Persuasion", "The Red Tent",

"The Road", "The Girl with the Dragon Tattoo", "The Pillars of the Earth", "Oliver Twist", "The Canterbury Tales", "And Then Were None",

"The Secret Life of Bees", "His Dark Materials Trilogy", "On the Road", "Heart of Darkness", "Love in the Time of Cholera", "The Master and Margarita", "The Shadow of the Wind", "Interview With the Vampire", "Invisible Man",

"In Cold Blood", "Aesops Fables", "Gulliver's Travels", "The Origin of Species", "Walden", "Roots", "The Glass Castle", "The Boy in the Striped Pajamas", "Sophie's World", "The Screwtape Letters", "Robinson Crusoe", "The Strange Case of Dr. Jekyll and Mr. Hyde",

"Candide", "The Prince", "The Complete Sherlock Holmes", "Fight Club", "The Art of War", "The Mists of Avalon", "The Time Machine", "Watchmen", "The Godfather", "The Trial", "The Sun Also Rises", "Tuck Everlasting", "Stranger in a Strange Land", "Emma", "Atonement",

"The Complete Brothers Grimm Fairy Tales", "Beloved", "James and the Giant Peach", "Leaves of Grass", "Bury My Heart at Wounded Knee", "The Things They Carried", "Their Eyes Were Watching God", "The Phantom Toolbooth", "Number the Stars", "Middlesex", "The World According to Garp",

"A Separate Peace", "Great Winter Night Time", "Winter Night : A Great Time for Fun"

]

```
target_words = input("Enter the target words (separated by SPACE):  
").split()  
search_books(target_words, book_library)
```

Output :

```
Book containing all target words [ great, winter, night, time, fun ] found : 'Winter Night : A Great Time for Fun', Index : 203
```

Part 5: Secret message

A. Discussion

In part 5, we are required to decode a specific secret message in order to find the next clue to Part 6. The secret message are as below:

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 lfwijjs
xmji. - 5

There are many ways to actually tackle the problem as it is quite general. Before we proceed in finding the algorithms that can be used to solve the problem, we need to properly dissect and specify the problem at hand. Thus, our assumptions for the problem are as below:

Assumptions

1. The secret message left by victim is quite short and cannot be decoded using advance algorithms that require long text such as Frequency Analysis Algorithm and Index of Coincidence (IOC).
2. As the victim is a businessman, there is a high possibility that he does not have information and knowledge regarding professional encryption strategy. Hence, the secret message that he left might possibly be created using only his creativity.
3. There might be clue left by victim on how to decode the secret message.

Based on the assumptions above, we concluded that the secret message left behind by victim was implemented using simple encryption method. Thus, it could be solved using the Brute Force Algorithm as the algorithm works best for simple and direct problems. For the algorithm, there are a few notable variations which could be used to solve the problem, which are Caesar Cipher, Vigenere Cipher and Transposition Cipher. To choose the best variation, we need to do a comparison check in order to find the best one.

	Caesar Cipher	Vigenere Cipher	Transposition Cipher
Introduction to Algorithm	The Caesar Cipher is a simple substitution cipher where each letter in the message is shifted a certain number of positions down or up the alphabet. To decode a message encoded with a Caesar cipher, we can apply the opposite shift to each letter.	The Vigenere Cipher works by using a keyword or phrase to determine multiple letter shifts during decryption. Each letter of the message is shifted based on the corresponding letter of the keyword. This creates a more complex encryption pattern compared to the simple letter shifting of the Caesar cipher. To decode the message, the same keyword is used to reverse the shifting process and retrieve the original message.	Transposition cipher works by rearranging the order of characters in the message based on a predetermined key. By applying the key's arrangement pattern in reverse, the recipient can restore the original order of the characters and reveal the message. Transposition ciphers do not change the actual characters but manipulate their positions, making it important to know the correct key to successfully decode the message.
Key Parameter	Shift value	Repeating keyword	Transposition rule
Security Level	Relatively weak	Stronger than Caesar cipher if keyword is long	Security depends on complexity and secrecy of the transposition rule
Key Length	Single integer value	Variable length based on keyword length	Variable length based on the transposition rule
Best-use Case	The Caesar cipher is best suited for situations where basic or minimal security is required and the focus is more on simplicity and ease of implementation rather than strong encryption.	The Vigenere cipher is best suited for scenarios where a relatively strong level of encryption is required, but not necessarily the highest level of security.	Transposition ciphers are best used in scenarios where the focus is on obscuring the message from casual observers or individuals without knowledge of the specific transposition method or key.
Limitation	1. Limited Key Space - The Caesar cipher has a fixed key space of 25 possibilities since there are only 25 potential shifts in the English alphabet. This means	1. Keyword Dependency - The Vigenere cipher requires knowledge of the keyword used during encryption. If the length of the keyword is	1. Keyword Dependency - The transposition cipher heavily relies on the knowledge of the correct key or arrangement pattern

	<p>that if the actual key used for encryption is not known, we would need to try all possible shifts to find the correct decryption.</p> <p>2. Unidirectional Shift - The Caesar cipher uses the concept of shifting each letter up or below a total number of times for each letter as a group. In the case of different direction or value of each letter, it will take a very long time to find the correct interpretation.</p>	<p>unknown, it is hard to determine the exact key length. If the keyword is long or randomly generated, decrypting the message without the correct key becomes extremely difficult.</p>	<p>used during encryption. Without the correct key, it becomes extremely difficult to decrypt the message accurately.</p>
--	--	---	---

Out of all three variations of Brute Force algorithms, we think that Caesar cipher is the best method to solve the secret message. This is because as stated in our assumptions, we assume that the secret message is encoded purely through the victim's creativity as a layman. Thus, the lower security level of Caesar cipher works best in this case as it provides the best solution for simple decoding problems. Furthermore, we assumed that the clue left behind by the victim in the secret message is the negative integer (-5) as it is the only pattern that stands out by itself. Out of all three variations, only the key for Caesar cipher needs to be in single integer value.

B. Pseudocode

1. Initialize an empty string called plaintext to store the decrypted message.
2. Iterate through each character, denoted as char, in the given ciphertext.
3. Check if char is an alphabetic character.
4. If char is alphabetic:
 - i. Determine the ASCII offset based on whether char is lowercase or uppercase.
 - ii. Decrypt the character by subtracting the ASCII offset and the given shift value, then take the modulo 26 and add the ASCII offset back.
 - iii. Append the decrypted character, denoted as decrypted_char, to the plaintext.
5. If char is not alphabetic, simply append it to the plaintext.
6. Return the plaintext as the decrypted message.
7. Set the ciphertext as the encrypted message you want to decrypt.
8. For each shift value from 1 to 25, do the following:
 - i. Decrypt the ciphertext using the current shift value in a forward direction.
 - ii. Store the decrypted message in a variable called decrypted_message_forward.
 - iii. Decrypt the ciphertext using the negative of the current shift value in a backward direction.
 - iv. Store the decrypted message in a variable called decrypted_message_backward.
 - v. Print the decrypted message for the current shift value in the forward direction.
 - vi. Print the decrypted message for the current shift value in the backward direction.
 - vii. Print a blank line to improve readability.

C. Running Time Complexity

For Caesar cipher, the algorithm time complexity is generally $O(n)$, where 'n' represents the length of the message being decoded. The same as other algorithm, it also has its best, worst and average case. Each of the case are as below:

Best Case: The best case for Caesar cipher algorithm is when the shift value is 0. Technically, this means that no decryptions are needed to solve the problem, as it is already solved. Hence, the time complexity is $O(1)$ or constant time, as the algorithm does not need to perform any calculations or iterations.

Worst Case: Considering that Caesar cipher algorithm only has a total of 26 shifts, it can achieve the worst case when the shift value is at its max (25). This is because the algorithm needs to iterate through the problem 25 times to perform shifts on the characters in order to decode the problem. Hence, the time complexity is $O(n)$ since it depends on the total number of iterations of the problem.

Average Case: The average case for Caesar cipher algorithm is when the total number of shifts is between the best and worst case (not 0 and 25). Hence, the time complexity is still the same as the worst case, which is $O(n)$ as it also depends on the total number of iterations for the problem.

Our Case: For the Caesar cipher algorithm, there are two possible ways to decode the secret message. The first way is to shift backward 5 times while the second way is to shift forward 21 times. As Caesar cipher has a fixed key space of 26, both the forward and backward shifts mirror each other, resulting in two different ways to solve the problem. Hence in our case, the running time complexity is considered as Average Case, as the total number of shifts for both solutions are between 0 and 25. Thus, the time complexity is $O(n)$.

D. Code

Source Code

```
def caesar_decrypt(ciphertext, shift):
    plaintext = ""
    for char in ciphertext:
        if char.isalpha():
            ascii_offset = ord('a') if char.islower() else ord('A')
            decrypted_char = chr((ord(char) - ascii_offset - shift) % 26 + ascii_offset)
            plaintext += decrypted_char
        else:
            plaintext += char
    return plaintext

ciphertext = "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 lfwij's xmji. - 5"

# Try all possible shift values from 1 to 25
for shift in range(1, 26):
    decrypted_message_backward = caesar_decrypt(ciphertext, shift)
    decrypted_message_forward = caesar_decrypt(ciphertext, -shift)

    print(f"Shift {shift} (Forward): {decrypted_message_forward}")
    print(f"Shift {shift} (Backward): {decrypted_message_backward}")
    print()
```

Example Output (+5 and -5)

```
Shift 5 (Forward): Drkd zobcyx sc mywsxq pyb wo! Sp iye psxn drsc xydo, vyyu kbyexn wi zbyzobdi. Rxd: S fscsdon dro kbok gsd'r wi dbyvvoi pbyw dro qkbnx cron. - 5
Shift 5 (Backward): That person is coming for me! If you find this note, look around my property. Hint: I visited the area with my trolley from the garden shed. - 5
```

Example Output (+21 and -21)

Shift 21 (Forward): That person is coming for me! If you find this note, look around my property. Hint: I visited the area with my trolley from the garden shed. - 5
Shift 21 (Backward): Drkd zobcyx sc mywsxq pyb wo! Sp iye psxn drsc xydo, vyyu kbyexn wi zbyzobdi. Rstd: S fscsdon dro kbok gsdr wi dbyvvoi pbyw dro qkbnox cron. - 5

Part 6: Find the next clue.

A. Discussion

I went to the shed. True enough, there is a trolley and several other items. The trolley can carry at most 30 kg of items. And the items in the shed are as follows (Table 1):

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

Table 1: Items in the garden shed and their weight.

I thought Mr Marshall must have visited the area based on his trolley capacity to carry these items. The next clue must be there.

Problem:

Find out which item was carried on the trolley

For this part, we are presented with a scenario where Mr Marshall visits a shed and discovers a trolley along with several items. The trolley has a maximum weight capacity of 30 kg, with the items in the shed along with their respective weights.

The assumptions that can be made are:

1. **Items are independent of each other:** The problem assumes that the items in the shed are independent of each other. This means that carrying one item does not affect the feasibility or weight capacity for carrying other items. It allows us to consider different combinations of items while determining the items carried on the trolley.
2. **The trolley has a maximum weight capacity:** The problem states that the trolley can carry at most 30 kg of items. This assumption provides a constraint for the algorithm and helps determine the subset of items that can be carried within the weight limit.

For this problem, we have proposed three algorithms which is Dynamic Programming, Greedy Algorithm and Brute Force:

1. Brute Force Algorithm:

Justification for the problem: The brute force algorithm can be applied to check all possible combinations of items to find the one that fits within the trolley's capacity. It would calculate the total weight of each combination and compare it with the trolley's capacity until a combination is found that fits. This algorithm guarantees finding the correct solution, ensuring that no potential combination is missed. However, it may be computationally expensive for larger sets of items.

2. Greedy Algorithm:

Justification for the problem: The greedy algorithm can be used to select items based on their weight, starting with the heaviest and gradually adding others until the trolley's capacity is reached. This algorithm provides a simple and efficient approach to finding a solution. In the context of this problem, it would involve selecting items one by one based on their weight until the trolley's capacity is filled. However, the greedy algorithm may not always yield the optimal solution.

3. Dynamic Programming Algorithm:

Justification for the problem: The dynamic programming algorithm, specifically the knapsack problem variation, can be applied to solve the problem of determining which item was carried on the trolley. It breaks down the problem into smaller subproblems, where each subproblem represents whether a specific item should be included in the trolley or not. The solutions to these subproblems are then used to determine the optimal combination of items that fit within the trolley's capacity. This algorithm efficiently handles larger sets of items and provides an optimal solution by considering all possible combinations.

Algorithm	Dynamic Programming	Greedy algorithm	Brute Force
Introduction to Algorithm	Dynamic programming is an algorithmic technique that solves optimization problems by breaking them into smaller overlapping subproblems. It efficiently stores and reuses computed results to avoid redundant computations, leading to improved time and space complexity.	A greedy algorithm makes locally optimal choices at each step, hoping to find a globally optimal solution. It selects the best available option at each stage without considering long-term consequences. Greedy algorithms are simple to implement and often provide fast solutions, but they may not	Brute force is a straightforward approach that exhaustively explores all possible solutions to a problem. It iterates through all choices, evaluating each one to find a satisfying solution. Brute force algorithms can be inefficient for large problems but are

		always guarantee an optimal solution.	useful for small-scale instances or as a baseline approach for correctness verification.
Advantages	<p>Efficiently solves complex optimization problems by breaking them down into smaller overlapping subproblems.</p> <p>Avoids redundant computations by storing and reusing computed results, improving time and space complexity.</p>	<p>Simple to implement and understand.</p> <p>Often provides fast solutions, making it suitable for problems with large input sizes.</p>	<p>Guarantees finding an optimal solution by systematically exploring all possible solutions.</p> <p>Suitable for small-scale problems or cases where no specific problem structure or optimization techniques are known.</p>
Limitations	<p>In the context of this problem, if the shed contains a large number of items, the dynamic programming algorithm's memory usage may become a limitation, especially if the available computational resources or memory capacity are limited</p>	<p>In this problem, the greedy algorithm may select the heaviest item first and continue adding items based on weight until the trolley's capacity is reached.</p> <p>However, this approach may not necessarily result in the combination of items that best matches what was carried on the trolley. It might overlook other combinations that fit within the trolley's capacity but have a different arrangement of items.</p>	<p>The brute force algorithm may become computationally expensive when dealing with larger sets of items.</p> <p>In the context of this problem, if the number of items in the shed is significantly large, checking all possible combinations of items to find the one that fits within the trolley's capacity may take a considerable amount of time and computational resources.</p>

In this case we have decided to use a **Dynamic Programming** to determine which item was carried on the trolley. It is well-suited for optimization problems like this, where we want to find the best solution within given constraints.

With dynamic programming, we can explore various combinations of items while considering their weights and the trolley's weight capacity. This allows us to find the best combination of items that maximizes the trolley's weight capacity without surpassing it.

B. Pseudocode

1. Initialize the number of items as the length of the "items" list.
2. Create a 2D list, "dp," with dimensions (num_items + 1) x (trolley_capacity + 1) and fill it with zeros.
3. Iterate over each item from 1 to num_items:
 - a. Get the weight of the current item.
 - b. Iterate over each capacity from 1 to trolley_capacity:
 - If the weight of the current item is less than or equal to the current capacity:
 - Set $dp[i][capacity]$ as the maximum value between $dp[i-1][capacity]$ and $dp[i-1][capacity - weight] + weight$.
 - Otherwise, set $dp[i][capacity]$ as $dp[i-1][capacity]$.
4. Initialize an empty list, "carried_items," and set capacity as trolley_capacity.
5. Retrieve the total weight carried on the trolley from $dp[num_items][trolley_capacity]$.
6. Iterate over the items in reverse order (from num_items to 1):
 - If $dp[i][capacity]$ is not equal to $dp[i-1][capacity]$:
 - Add the current item and its weight to the "carried_items" list.
 - Decrease the capacity by the weight of the current item.
7. Reverse the order of "carried_items" to correct the item order.
8. Return the "carried_items" list and the total weight.

C. Running time complexity

Worst Case:

The worst-case time complexity of the code is $O(n * W)$, where n represents the number of items and W represents the trolley capacity. In the worst case, the code needs to consider all possible combinations of items and capacities, resulting in a time complexity that grows linearly with the product of the number of items and the trolley capacity.

Best Case:

The best case occurs when the trolley capacity is very small or when the items list is empty. In this scenario, the code quickly terminates after initializing the necessary variables and returns an empty list of carried items. The time complexity of the best case is $O(1)$ since it involves a constant amount of operations, regardless of the input size.

Average Case:

The average case time complexity is also $O(n * W)$, assuming a random distribution of item weights and trolley capacity. On average, a portion of the total number of combinations, proportional to the input size, needs to be considered. The time complexity grows linearly

with the product of the number of items and the trolley capacity, providing a reasonable estimate for the average performance of the code.

D. Code

```
def findCarriedItem(items, trolley_capacity):
    num_items = len(items)
    dp = [[0] * (trolley_capacity + 1) for _ in range(num_items + 1)]

    for i in range(1, num_items + 1):
        weight = items[i - 1][1]
        for capacity in range(1, trolley_capacity + 1):
            if weight <= capacity:
                dp[i][capacity] = max(dp[i - 1][capacity], dp[i - 1][capacity
- weight] + weight)
            else:
                dp[i][capacity] = dp[i - 1][capacity]

    carried_items = []
    capacity = trolley_capacity
    total_weight = dp[num_items][trolley_capacity]

    for i in range(num_items, 0, -1):
        if dp[i][capacity] != dp[i - 1][capacity]:
            item, weight = items[i - 1]
            carried_items.append(item)
            capacity -= weight

    carried_items.reverse() # Correcting the order of the carried items

    return carried_items, total_weight

# Example usage
items = [
    ("A sack of corn for the chicken at the barn.", 12),
    ("A hoe for the greenhouse.", 5),
    ("An oil tank filled with fuel for the boat at the lake.", 10),
    ("Two pieces of tires for the car in the garage.", 8),
    ("Two pieces of tires for the car in the garage.", 8)
]
trolley_capacity = 30

carried_items, total_weight = findCarriedItem(items, trolley_capacity)
```

```
print("The item(s) carried on the trolley are:")
for item in carried_items:
    print("- " + item)

print("Total weight carried on the trolley:", total_weight)
```

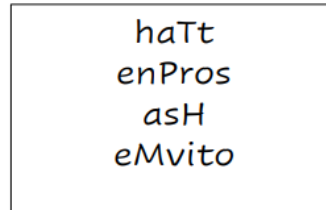
Output:

```
The item(s) carried on the trolley are:
- A sack of corn for the chicken at the barn.
- An oil tank filled with fuel for the boat at the lake.
- Two pieces of tires for the car in the garage.
Total weight carried on the trolley: 30
```


Part 7: Almost there!

A. Description

Based on the items carried, I visited all the areas. I found another secret message (Figure 5) in a bottle in one of the areas!



```
haTt
enPros
asH
eMvito
```

Figure 6: The secret message.

Although it didn't make sense initially, I realised each line was a word with jumbled letters.

Problem:

What is the secret message?

This part requires us to unjumble the secret message.

Assumptions:

1. The capital letters of each secret word is the first letter of it.
2. A sentence will be made once all the words are unjumbled.

To solve this part, we needed a dictionary file that contains words that will be used to compare with the secret message. We will then unjumbled each word and in the end combine all of it to form a sentence. The first solution is to use brute force algorithm by trying every possible permutations of the secret word and compare it to the dictionary. The 2nd solution is a modified pattern matching algorithm. It compares the length of the jumbled up words with the one inside the dictionary and slowly removes the common letters. Last solution is an anagram algorithm which compares the arranged secret word with the arranged words inside the dictionary file.

Algorithm	Brute Force Algorithm	Modified Pattern Matching Algorithm	Anagram Algorithm
Introduction to Algorithm	A straightforward method of solving a problem that relies on sheer computing power and trying every possibility	Compare the length of the jumbled up words with the one inside the dictionary. If the length is the same, remove the letter from the	An algorithm that compares two strings by arranging both strings alphabetically. In this case, we will

	<p>rather than advanced techniques to improve efficiency.</p> <p>Uses permutations on the jumbled up words and compares all possible permutations to a dictionary.</p>	<p>jumbled up word that matches with the one in the dictionary. By the end, if there are no more letters in the jumbled up words, then there's a match.</p>	<p>arrange the jumbled up words and the words in the dictionary and compare both of them.</p>
Advantages	<ul style="list-style-type: none"> - A guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem. - Code is relatively simple and easy to understand. - Stop as soon as it finds a match, which is efficient. 	<ul style="list-style-type: none"> - Just like Brute Force, a guaranteed way to find the correct solution. - Can handle variations in the input word such as lengths or different arrangements of letters. - Narrow down the search space by comparing lengths and first letter. 	<ul style="list-style-type: none"> - Faster compared to previous algorithms. Reduced time complexity - Utilize sorting. Easier comparison and identification of anagrams. Reduces search space and optimizes the matching process. - Can handle longer words.
Limitations	<p>Case sensitive. The output can be misled if the word is an anagram to the jumbled up words and they have the same length. Example, "has" and "ash".</p>		
	<ul style="list-style-type: none"> - The time complexity is high. Goes above $O(n!)$, where n is the maximum length of the word. - Iterates through all permutations 	<ul style="list-style-type: none"> - Not widely recognized but closely resembles a pattern matching algorithm. Difficult to explain to others. 	<ul style="list-style-type: none"> - The algorithm relies on the size of the dictionary. As the number of words increases, the time complexity also increases. - Requires memory to store the sorted words inside the dictionary.
Modifications	Add another requirement when	Add another requirement when	Add another requirement when

	comparing the permutations with the words inside the dictionary. It will only consider a match if both are similar and the front letter of the permutation is a capital letter.	comparing the words with the words inside the dictionary. It will only consider a match if there are no more letters in the jumbled up words and they share the same front letters.	comparing the words with the words inside the dictionary. It will only consider a match if both sorted words are similar and they share the same front letters.
--	---	---	---

Chosen algorithm : Anagram Algorithm. Compared to brute force, it has a lower time complexity. When compared with the modified pattern matching, it has a higher time complexity but the sorting operation in the anagram algorithm is generally more efficient than the string manipulations used in modified pattern matching.

B. Pseudocode

1. Load a dictionary of valid words.
2. Accept the jumbled word as input.
3. Arrange the word alphabetically.
4. Arrange the words inside the dictionary alphabetically.
5. Compare the arranged word and the arranged words in the dictionary.
6. If there's a match, add it to the list of solutions.
7. Repeat step 3-6 for all the other words
8. Display the sentence.

C. Running Time Complexity

The running time complexity is $O(w * m * (k \log k))$ where w is the number of words in the list, m is the number of words in the dictionary and k is the length of the longest word.

Compared to brute force algorithm variation 1 ($O(k!)$, where k is the longest word), this algorithm has a shorter running time complexity. It also has lower memory usage and a more efficient search process. When compared with the 2nd brute force algorithm ($O(n * m * k)$, where n is the length of the word, m the number of words in the dictionary and k is the length of the longest word), anagram algorithm also has a shorter running time complexity.

The best case scenario occurs when the input word is an anagram of a word in the dictionary and it is encountered early in the search process. In this case, the algorithm would identify the anagram quickly after comparing it with only a few words in the dictionary. The time complexity of the best case is $O(1)$, as it can find a match in constant time.

The worst case scenario happens when the input word is not an anagram of any word in the dictionary. In this case, the algorithm would need to compare the input word with every word in the dictionary. The worst case time complexity is $O(w * m * k * \log k)$, where w is the

number of words in the list, m is the number of words in the dictionary and k is the length of the longest word.

The average case scenario assumes a random distribution of input words, some of which are anagrams and others that are not. In this case, the algorithm would need to compare the input word with a portion of the dictionary before finding a match or determining that no match exists. The average case time complexity is also approximately $O(w * m * k * \log k)$ as well.

D. Code

```
def load_dictionary(file_path):  
    with open(file_path, 'r') as file:  
        word_list = [word.strip().lower() for word in file]  
    return word_list  
  
def anagram(word,dictionary):  
  
    for i in range(len(word)):  
        if word[i].isupper():  
            front = word[i]  
  
    sorted_word = sorted(word.lower())  
  
    for dict_word in dictionary:  
        if len(dict_word) == len(word):  
            sorted_dict_word = sorted(dict_word)  
  
            if (sorted_dict_word == sorted_word and  
dict_word[0]==front.lower()):  
                solved_words=dict_word  
    return (solved_words)  
print("Anagram Algorithm\n")  
dictionary_file = "C:/Users/ACER/Desktop/Python/dictionary.txt"  
words = ['haTt','enPros', 'asH', 'eMvito']  
answer = []  
  
dictionary = load_dictionary(dictionary_file)  
for i in range(len(words)):  
    answer.append(anagram(words[i],dictionary))  
  
print(answer)
```

Output

```
Anagram Algorithm  
['that', 'person', 'has', 'motive']
```

Part 8: Murder Suspect

A. Discussion

So, from the last message, I know the murderer must have a strong motive. Is it money? Or something else?

I list each family member's characteristics, relationship with Mr Marshall, and net worth below (Table 2).

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

Table 2: Mr Marshall's family members, characteristics, and wealth.

The murderer must be one of them. But who?

Problem:

Who has the most significant motive to be the suspect in this murder?

The problem requires us to identify the family member with the most significant motive to be the suspect in a murder case. We are provided with a list of family members' characteristics, their relationships with the victim (Mr. Marshall), and their respective net worth. The goal is to determine which family member has the strongest motive for the murder.

Solution suggestion :

- We suggest using a scoring approach to determine the suspects in the murder case.
- The scoring quantitatively assesses various factors related to each family member, including their relationship, characteristics, and net worth.
- The algorithm assigns weights to specific attributes and characteristics to capture their potential significance in relation to the motive for murder.

Assumptions :

In order to understand how we solve this problem, we make the following assumptions.

- We assume that certain attributes, such as being a son or daughter or having specific characteristics, can contribute to a higher motive for murder. The weights assigned to

these attributes are subjective and based on assumptions made during the scoring process.

- We assume that the net worth of each family member can potentially influence their motive for murder. We assign a higher motive score to family members with lower net worth, indicating a potential financial motive.

	Greedy Algorithm	Dynamic Programming	Brute Force Algorithm
Introduction to Algorithm	A greedy algorithm is an algorithm that makes the locally optimal choice at each step in the hope of finding a globally optimal solution. Greedy algorithm always chooses the solution that seems best at the moment, without considering the consequences of its choices later on.	Dynamic programming is an algorithm that solves a problem by first solving smaller versions of the problem, and then using the solutions to the smaller problems to solve the larger problem.	A brute-force algorithm is an algorithm for solving problems by trying all possible solutions. In other words, the brute-force algorithm tries every possible combination of values until it finds a combination that solves the problem.
Advantages	<ol style="list-style-type: none"> 1. Simple to understand and implement. 2. A good choice for problems where time and space complexity are not important. 	<ol style="list-style-type: none"> 1. Can be used to solve problems that are too complex for the greedy algorithm. 	<ol style="list-style-type: none"> 1. A guarantee to find the optimal solution, if one exists.
Limitations	<ol style="list-style-type: none"> 1. Not always guaranteed to find the optimal solution. 2. Only considers the local optimality of each step, and does not consider the global optimality of the solution. 	<ol style="list-style-type: none"> 1. Can be difficult to implement and inefficient for problems with large numbers of subproblems. 2. Requires the algorithm to store the solutions to all subproblems, which can be very memory-intensive. 	<ol style="list-style-type: none"> 1. Tries all possible solutions to the problem, which can be very time-consuming for problems with large numbers of possible solutions.

So, we decided to use the greedy algorithm to find the most likely suspect. It is the simplest and most efficient algorithm. We considered using the dynamic programming or brute-force algorithms, but they are more time-consuming and we felt that the time it would take to implement and run them would outweigh the benefits of finding the optimal solution. We believe that the greedy algorithm is a good compromise between speed and accuracy.

In order for us to use the greedy algorithm optimally, we modified a bit of the original greedy algorithm. In the original greedy algorithm, the focus is on making locally optimal choices at each step to achieve the overall best solution. The algorithm usually operates on arrays or lists, without explicitly using dictionaries or key-value pairs.

In our modified code, we introduced a dictionary to store each family member's information. The dictionary contains keys such as "name", "relationship", "character", and "net_worth" to represent different attributes of the family members. This allows us to associate multiple properties with each family member and access them using the respective keys.

B. Pseudocode

1. Initialize a set of suspects to be empty.
2. For each family member:
 1. Calculate the motive score for the family member.
 2. If the motive score is greater than the current maximum motive score:
 1. Set the current maximum motive score to the motive score.
 2. Add the family member to the suspects set.
3. Sort the set of suspects by their motive scores, in descending order.
4. Return the top k suspects from the set of suspects, where k is the number of suspects that we want to return.

C. Running Time Complexity

For the best case running time complexity, the family members are very small such as one. The *for* loop iterating over family members will execute only once. Sorting the family members will take $O(1)$ time complexity since there are very few elements to sort. Therefore, the best case time complexity is $O(1)$.

For the average case, the number of family members is moderate. The *for* loop iterating over family members will execute 'n' times. Sorting the family members will take $O(n \log n)$ time complexity. Therefore, the average case time complexity is $O(n \log n)$.

For the worst case, the number of family members is large. The *for* loop iterating over family members will execute 'n' times, where 'n' is the total number of family members. Sorting the family members will take $O(n \log n)$ time complexity. Therefore, the worst case time complexity is $O(n \log n)$.

D. Code

```
def greedy_suspect(family_members, max_suspects):  
    # Calculate the motive score for each family member.  
    for member in family_members:
```



```

        motive_score = calculate_motive_score(member)
        member["motive_score"] = motive_score

    # Sort the family members by their motive score, in descending
    order.
    family_members.sort(key=lambda x: x["motive_score"],
reverse=True)

    # Extract the names of the top max_suspects family members.
    suspects = [member["name"] for member in
family_members[:max_suspects]]

    # Return the list of suspect names.
    return suspects

def calculate_motive_score(member):
    motive_score = 0

    # Assign motive score based on relationship
    if "Son" in member["relationship"] or "Daughter" in
member["relationship"]:
        motive_score += 3
    elif "Brother" in member["relationship"] or "Sister" in
member["relationship"]:
        motive_score += 2
    elif "Uncle" in member["relationship"]:
        motive_score += 1

    # Assign motive score based on characteristics
    if "rude" in member["character"]:
        motive_score += 3
    elif "quiet" in member["character"]:
        motive_score += 2
    elif "lover" in member["character"] or "playful" in
member["character"] or "retired" in member["character"]:
        motive_score += 1

    # Assign motive score based on net worth
    if member["net_worth"] < 100000:

```

```

        motive_score += 1
    elif member["net_worth"] > 99999:
        motive_score -= 1

    return motive_score

def main():
    # Get the number of family members from the user.
    num_members = int(input("Enter the number of family members: "))

    # Create a list to store the family members' information.
    family_members = []

    # Prompt the user for each family member's information.
    for i in range(num_members):
        name = input("Enter the name of family member {}:
".format(i+1))
        relationship = input("Enter the relationship of {}:
".format(name))
        character = input("Enter the character of {}: ".format(name))
        net_worth = float(input("Enter the net worth of {} ($):
".format(name)))

        # Create a dictionary to store the family member's
        information.
        member = {
            "name": name,
            "relationship": relationship,
            "character": character,
            "net_worth": net_worth
        }
        # Add the member to the list.
        family_members.append(member)

    # Find the most likely suspect using the greedy algorithm.
    suspects = greedy_suspect(family_members, 3)

    # Print the result.
    print("The most likely suspects are:")

```

```
    for suspect in suspects:
        print(suspect)

if __name__ == "__main__":
    main()
```

Output :

```
The most likely suspects are:
Jones Marshall
```

Part 9: Story Ending

In the small town of Marshallville, a cloud of darkness loomed over the Marshall Mansion. The mysterious death of Mr. Phillip Marshall, a wealthy businessman, had sent shockwaves through the community. As a detective working for the local police department, it was my duty to unravel the truth behind this heinous crime.

The investigation began at the entrance of the ground floor of the mansion. Determined to leave no stone unturned, I embarked on a meticulous search through each room, hoping to find any clue that would lead me to the identity of the murderer. Employing a methodical approach, I utilized the Breadth-First Search algorithm, meticulously examining all sixteen rooms within the mansion.

My tireless search eventually led me to the library, where I stumbled upon a suspicious old safe. The lock presented a challenge, as it required a three-digit combination. Determined to crack it, I employed a Randomized Guessing Algorithm, attempting various combinations until success finally smiled upon me. The correct combination, as fate would have it, was 601.

Inside the safe, I discovered two letters that appeared strikingly similar but were not identical. Realizing the importance of the differences, I turned to the Longest Common Subsequences Algorithm to identify the variations. The first letter contained words like "fun!", "day," "summer," and "fine," while the second letter had differences such as "time!," "fun," "night," "winter," and "great."

The variations in the letters hinted at a potential book title. Armed with this information, I applied the Binary Search Algorithm to swiftly navigate through the library's countless volumes. After scanning the shelves diligently, I located the book with the keywords "Winter Night: A Great Time For Fun." It was the 203rd book in the collection.

As I flipped through the pages of the book, a folded piece of paper caught my attention. It contained a secret message, cleverly encoded to protect vital information. Utilizing my knowledge of codes and ciphers, I deciphered the message using the Caesar Cipher. The decoded words sent a chill down my spine: "That person is coming for me! If you find this note, look around my property. Hint: I visit the area with my trolley from the garden shed."

Intrigued by this revelation, I made my way to the garden shed, where I discovered various items. Utilizing a Knapsack Algorithm, I deduced that the victim had taken a sack of corn for the chicken at the barn, an oil tank filled with fuel for the boat at lake and two pieces of tires for the car in the garage, based on the trolley's weight capacity.

With each step, the puzzle grew closer to being solved. My journey took me to different areas of the property until I stumbled upon a bottle containing yet another secret message. This time, the message was jumbled, resembling an anagram. Applying the Anagram Algorithm, the hidden words emerged: "That Person Has Motive."

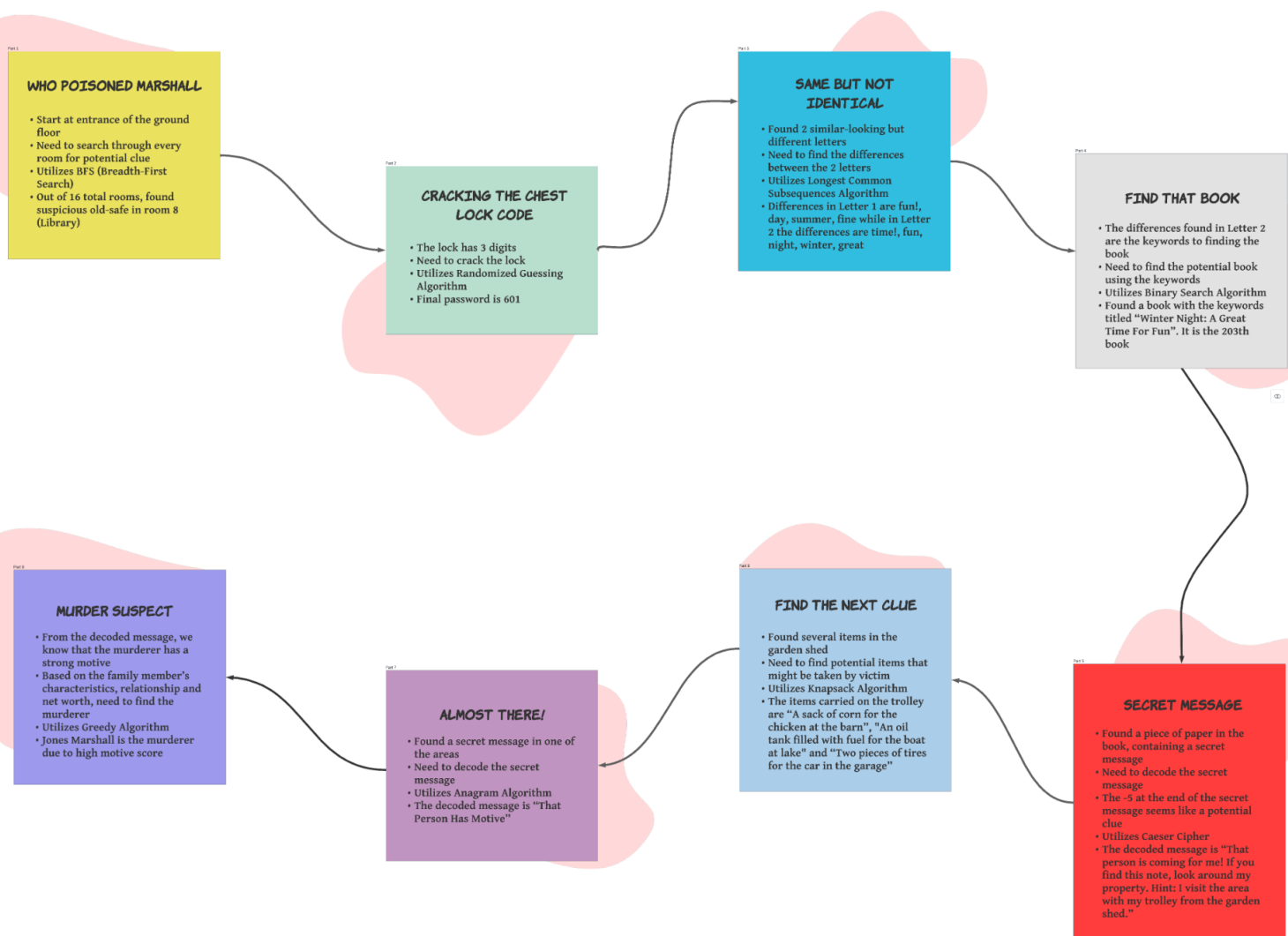
As the pieces of the puzzle fell into place, it became evident that the murderer had a strong motive. Drawing upon the characteristics, relationships, and net worth of the family

members, I employed the Greedy Algorithm once more. The individual who stood out with the most significant motive was none other than Jones Marshall, Mr. Phillip Marshall's son.

Jones, despite his outward appearance, harbored deep resentment towards his father. Known for his rude behavior, it seemed that his animosity had finally pushed him to commit this heinous act. His large inheritance further fueled his motive, making him the prime suspect.

With the evidence gathered and the murderer identified, justice could now be served. Jones Marshall was apprehended, and the truth behind the murder of Mr. Phillip Marshall was finally unveiled, bringing closure to the once-shrouded mystery that had gripped Marshallville.

And so, the story of the Marshall Mansion and its tragic events came to a close. The residents of Marshallville could now rest assured that justice had prevailed, and the darkness that had loomed over their town had been lifted. The memory of this captivating tale would forever linger, a testament to the resilience of truth and the unwavering pursuit of justice.



Flowchart (Part 1 - Part 8)

Appendices

Appendix A - Group Contract

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

Each member role and contract

Group Leader : Hazeeq Syakirin bin Azahar

Contract Item: As a Team we agree to

• Participation	<ol style="list-style-type: none">1. All members are expected to actively contribute to the group's objectives.2. Attend and actively engage in any group discussion or meetings held by either the leader or any other group member.3. Get involved in any form of group work. For example, presentation, report writing, making slides, etc.4. Members must complete assigned tasks promptly and with dedication.
• Communication	<ol style="list-style-type: none">1. Communication should be respectful and professional at all times.2. All group members should actively listen to other's ideas and perspectives.3. Feedback and suggestions are encouraged to foster a collaborative environment.4. Group members can communicate with each other via Whatsapp group.
• Meetings	<ol style="list-style-type: none">1. Meetings will be scheduled in advance.2. All members are expected to attend the meeting.3. If any of the group members are not able to attend said meeting, prior notice and reasoning should be given.4. Agendas will be prepared for each meeting, and members are encouraged to contribute agenda items in advance.5. Minutes or meeting summaries will be documented and shared with all members.
• Conduct	<ol style="list-style-type: none">1. Respect for diversity, inclusivity, and differing opinions is paramount.2. Members should be punctual, prepared, and organized for meetings and assigned tasks.

	<p>3. Members should maintain confidentiality and respect the privacy of others.</p> <p>4. Discrimination, harassment, or any form of disrespectful behavior will not be tolerated.</p> <p>5. Members should help out other members that are struggling with their part.</p>
<ul style="list-style-type: none"> Deadlines 	<p>1. Members must agree on deadlines for individual tasks and contribute to overall project deadlines.</p> <p>2. If unforeseen circumstances prevent meeting a deadline, group members should inform the group as soon as possible.</p> <p>3. Members should strive to deliver high-quality work that meets the group's standards.</p>
<ul style="list-style-type: none"> Conflict 	<p>1. Any form of conflict should be addressed promptly and respectfully, focusing on finding mutually beneficial solutions.</p> <p>2. Members will actively listen to one another, striving to understand different perspectives.</p> <p>3. If conflicts cannot be resolved within the group, a designated mediator or facilitator will be sought.</p>

<p>Clause</p> <p>In any violation of the above, we agree</p>	<p>1. Address the issue. The issue will be addressed promptly and properly by the group should any member violate the guideline described in this contract.</p> <p>2. Resolution Attempts. The group will make reasonable attempts to address the problem internally through productive discussion, attentive listening, and appreciation of various viewpoints.</p> <p>3. Open Communication. The offending member(s) will be approached politely and informed of the specific infringement and how it affects the dynamics or development of the group.</p> <p>4. Escalation. The issue may be escalated to a higher authority, such as a project supervisor or teacher, if necessary, in circumstances when violations continue or are severe and internal resolution attempts fail.</p> <p>5. Mediation. A designated mediator or facilitator will be brought in to help resolve the disagreement if the first attempts at resolution are unsuccessful. The mediator will maintain objectivity and assist in fostering honest dialogue and amicable resolutions.</p> <p>6. Consequences. Consequences may include, but are not limited to, verbal/written warnings, responsibility redistribution,</p>
---	--

adjustment of group roles, or, in severe circumstances, removal from the group and project depending on the gravity and frequency of the infraction.

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

No	Matric No	Name	Team Role	Signature
1.	U2000687	HAZEEQ SYAKIRIN BIN AZAHAR	Group Leader	
2.	U2102733	MUHAMAD IZZUL IZZANI BIN ABU BAKAR	Communication Coordinator	
3.	U2102865	MOHAMAD QHALISH BIN MOHD HAROMA	Creative/design expert	
4.	U2102743	MUHAMMAD ARIF EZUAN BIN MOHD FAUZI	Content Editor	
5.	U2001853	MUHAMMAD ADAM AIMAN BIN HELMI	Quality Tester	
6.	U2000726	MUHAMMAD MUQRI QAWIEM BIN HANIZAM	Research Specialist	

(Assessor:

Date Received:

)

Appendix B - FILA form

FILA FORM – University of Malaya

Part	FACTS	IDEAS	LEARNING ISSUES	ACTION	DATELINE
P1	What do we know about the task? Explore a mansion map. One of the rooms has a clue.	What do we need to find out? Algorithms to search all rooms in the mansion without missing any room.		Who is going to do it? Arif	
	Phases Discussion. Research. Comparison. Implementation. Reporting. Documentation.	Theory into practical The issue entails navigating a mansion, providing a practical illustration of how to use search and investigating various techniques, such as grid-based or graph-based		Activities Group discussion & brainstorming to generate ideas. Online research to find possible solutions. Comparing all 3 algorithms to find the best one.	Date 11th May

P2	What do we know about the task? Crack an old safe.	What do we need to find out? Algorithms to crack an old safe that has a 3-digit lock.	Who is going to do it? Hazeeq	
	Phases Discussion. Research. Comparison. Implementation. Reporting. Documentation.	Theory into practical The real-world challenge of breaking a three-digit code or any other form of secret password involves numerous CS fields. We need to solve a straightforward problem, but the answers can be scaled to handle more complicated problems.	Activities Group discussion & brainstorming to generate ideas. Online research to find possible solutions. Comparing all 3 algorithms to find the best one.	Date 11th May
P3	What do we know about the task? Find the differences between 2 similar messages.	What do we need to find out? Algorithms to compare strings.	Who is going to do it? Adam	
	Phases Discussion. Research. Comparison.	Theory into practical Comparing messages is employed in a variety of real-world situations, including social media platforms,	Activities Group discussion & brainstorming to generate ideas.	Date 11th May

	Implementation. Reporting. Documentation.	compression, and encryption.	Online research to find possible solutions. Comparing all 3 algorithms to find the best one.	
P4	What do we know about the task? Search for a book title in a sorted bookshelf.	What do we need to find out? Algorithms to search for a specific element in a sorted collection.	Who is going to do it? Muqri	
	Phases Discussion. Research. Comparison. Implementation. Reporting. Documentation.	Theory into practical Numerous real-world uses of searching include search engines, dictionaries, and library platforms. A great example is searching for any keyword using Ctrl-F.	Activities Group discussion & brainstorming to generate ideas. Online research to find possible solutions. Comparing all 3 algorithms to find the best one.	Date 11th May

P5	What do we know about the task? Decode an encrypted message.	What do we need to find out? Algorithms to decode an encrypted message.	Who is going to do it? Izzul	
	Phases Discussion. Research. Comparison. Implementation. Reporting. Documentation.	Theory into practical A vast area of computer science called cryptography deals with encrypted messages and is employed in many real-world situations, particularly in achieving secure communication.	Activities Group discussion & brainstorming to generate ideas. Online research to find possible solutions. Comparing all 3 algorithms to find the best one.	Date 11th May
P6	What do we know about the task? Find the most optimal way to choose items from the shed.	What do we need to find out? Algorithms to determine the most optimal way to solve problems.	Who is going to do it? Qhalish	
	Phases Discussion. Research. Comparison. Implementation.	Theory into practical The majority of real-world challenges that are quite comparable to this one are in the business and commerce sectors, where one would	Activities Group discussion & brainstorming to generate ideas. Online research to find possible solutions.	Date 11th May

	Reporting. Documentation.	typically aim to maximize earnings or minimize costs.	Comparing all 3 algorithms to find the best one.	
P7	What do we know about the task? Decode some jumbled up words.	What do we need to find out? Algorithms to decode jumbled up words.	Who is going to do it? Hazeeq	
	Phases Discussion. Research. Comparison. Implementation. Reporting. Documentation.	Theory into practical In real-world circumstances, the issue of organized word jumbles is employed in and of itself to repair broken messages and data.	Activities Group discussion & brainstorming to generate ideas. Online research to find possible solutions. Comparing all 3 algorithms to find the best one.	Date 11th May
P8	What do we know about the task? Find the person with the most motive based on their characteristics.	What do we need to find out? Algorithms to calculate the level of human motivation based on their characteristics	Who is going to do it? Muqri	
	Phases Discussion.	Theory into practical The answer to this issue can be applied to similar situations in the	Activities	Date 11th May

	Research. Comparison. Implementation. Reporting. Documentation.	real world. An example would be the use of statistics and numerical traits to identify potential criminals.	Group discussion & brainstorming to generate ideas. Online research to find possible solutions. Comparing all 3 algorithms to find the best one.	
--	---	---	--	--

References

1. *Breadth First Search or BFS for a Graph* - GeeksforGeeks. (2012, March 20). GeeksforGeeks. <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>
2. *Longest Common Subsequence (LCS)* - GeeksforGeeks. (2011, June 14). GeeksforGeeks. <https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/>
3. *Binary Search - Data Structure and Algorithm Tutorials* - GeeksforGeeks. (2014, January 28). GeeksforGeeks. <https://www.geeksforgeeks.org/binary-search/>
4. *Caesar Cipher in Cryptography* - GeeksforGeeks. (2016, June 2). GeeksforGeeks. <https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/>
5. *Dynamic Programming* - GeeksforGeeks. (n.d.). GeeksforGeeks. <https://www.geeksforgeeks.org/dynamic-programming/>
6. *Python | Program to implement Jumbled word game* - GeeksforGeeks. (2018, September 13). GeeksforGeeks. <https://www.geeksforgeeks.org/python-program-to-implement-jumbled-word-game/>
7. *Greedy Algorithms* - GeeksforGeeks. (n.d.). GeeksforGeeks. <https://www.geeksforgeeks.org/greedy-algorithms/>

Source Code

Link to our Google Colab:

<https://colab.research.google.com/drive/1M-spZgEaJfiuDdWDIVBIDKkZ0yd7j86n?usp=sharing>