Practical-1

Q1. Caesar Cypher and Improved Caesar Cypher

Caesar Cypher: An Overview

The Caesar Cypher, named after Julius Caesar who used it in his private correspondence, is a type of substitution Cypher where each letter in the plaintext is shifted a certain number of places down or up the alphabet. It is one of the simplest and most widely known encryption techniques.

Key Characteristics:

1. **Shift Value (Key):** The number of positions each letter in the plaintext is shifted. For example, with a shift of 3, A becomes D, B becomes E, and so on.
2. **Alphabet Wrap-Around:** The alphabet is treated as circular, so after Z comes A again. This means a shift of 1 on Z would result in A.
3. **Case Sensitivity:** Traditionally, the Cypher is case-sensitive, meaning 'A' and 'a' are considered distinct and are encrypted separately.

Encryption Process:

1. **Input:** A plaintext message and a shift value (key).
2. **Shift:** Each letter in the plaintext is shifted by the specified key. Non-alphabetic characters remain unchanged.
3. **Output:** The resulting Cyphertext.

Decryption Process:

1. **Input:** A Cyphertext message and the same shift value (key) used for encryption.
2. **Shift Back:** Each letter in the Cyphertext is shifted backward by the specified key to retrieve the original plaintext.
3. **Output:** The original plaintext message.

Example:

- **Plaintext:** HELLO

- **Key:** 3
- **Encryption:**
    - H (shift by 3) -> K
    - E (shift by 3) -> H
    - L (shift by 3) -> O
    - L (shift by 3) -> O
    - O (shift by 3) -> R
    - **Cyphertext:** KHOOR
- **Decryption:**
    - K (shift back by 3) -> H
    - H (shift back by 3) -> E
    - O (shift back by 3) -> L
    - O (shift back by 3) -> L
    - R (shift back by 3) -> O
    - **Plaintext:** HELLO

Applications:

- Historically used in military and government communication.
- Educational purposes to demonstrate basic encryption techniques.
- Simple puzzles and games for recreational cryptography.

Limitations of the Caesar Cypher

1. **Susceptibility to Brute-Force Attacks:**
    - With only 25 possible shifts, it is easy for an attacker to try all possible keys and decrypt the message.
2. **Frequency Analysis Vulnerability:**
    - The Cypher does not alter the frequency of letters, allowing attackers to use frequency analysis to break the encryption based on the known frequency of letters in the language.
3. **Lack of Complexity:**
    - The simplicity of the Cypher means it provides very little security and can be easily broken with minimal computational effort.
4. **Fixed Shift Key:**

- The use of a single shift key for the entire message makes it easy to deCypher once the key is known.

5. **Not Suitable for Modern Communications:**
   - Given its weaknesses, the Caesar Cypher cannot protect sensitive information against modern cryptographic analysis and attacks.
6. **No Integrity or Authentication:**
   - The Cypher provides no mechanisms to ensure the integrity of the message or authenticate the sender, making it vulnerable to tampering and impersonation.

Code:

```
print("\nCaesar Cypher Encryption/Decryption\n")
choice = input("Enter the operation you want to perform:
Encryption(1)/Decryption(0): ")

# Populating the alphabet table before hand without loop to avoid any overhead
alphabetTable = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7,
'I': 8, 'J': 9, 'K': 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14, 'P': 15, 'Q': 16, 'R':
17, 'S': 18, 'T': 19, 'U': 20, 'V': 21, 'W': 22, 'X': 23, 'Y': 24, 'Z': 25}
reverseAlphabetTable = {v: k for k, v in alphabetTable.items()}

if choice == "1":
    input_message = input("\nEnter the message you want to encrypt: ")
    key = int(input("\nEnter the encryption key you want to use: "))
    encrypted_message = ""

    for c in input_message:
        if (65 <= ord(c) <= 90):
            encrypted_message += reverseAlphabetTable[(alphabetTable[c] + key) %
26]
        elif (97 <= ord(c) <= 122):
            temp = ord(c) - 32
            encrypted_message += reverseAlphabetTable[(alphabetTable[chr(temp)] +
key) % 26].lower()
        else:
            encrypted_message += c

    print("\nEncrypted message: ", encrypted_message, end="\n\n")

elif choice == "0":
    input_message = input("\nEnter the message you want to decrypt: ")
    key = int(input("\nEnter the decryption key you want to use: "))
    decrypted_message = ""

    for c in input_message:
        if (65 <= ord(c) <= 90):
```

```
            decrypted_message += reverseAlphabetTable[(alphabetTable[c] - key) %
26]
        elif (97 <= ord(c) <= 122):
            temp = ord(c) - 32
            decrypted_message += reverseAlphabetTable[(alphabetTable[chr(temp)] -
key) % 26].lower()
        else:
            decrypted_message += c

    print("\nDecrypted message: ", decrypted_message, end="\n\n")

else:
    print("\nInvalid choice! Please enter 1 for encryption or 0 for decryption.")
```

Output:

```
(base) tirthshah@Tirths-MacBook-Pro PDEU-Sem-5 % python3 -u "/Users/tirthshah/Doc
Caesar Cypher Encryption/Decryption
Enter the operation you want to perform: Encryption(1)/Decryption(0): 1
Enter the message you want to encrypt: Tirth
Enter the encryption key you want to use: 14
Encrypted message:  Hwfhv
```

Improved Caesar Cypher: An Overview

The Improved Caesar Cypher enhances the traditional Caesar Cypher by incorporating additional security measures, making it more robust against attacks. This version uses a keyword to create a variable shift pattern, combined with a simple hash function to determine the shift value, thereby increasing the complexity and security of the encryption process.

Key Improvements:

1. **Keyword-Length Adjustment:**
    - The keyword is adjusted to match the length of the input message, ensuring a consistent shift pattern throughout the entire message.
2. **Hash Function for Shift Value:**
    - A simple hash function based on the keyword generates a shift value, adding an extra layer of security and variability to the encryption process.

Example:

- **Plaintext:** HELLO
- **Keyword:** KEY
- **Key:** 3

Step-by-Step Encryption Process:

1. **Adjust the Keyword Length:**
    - The keyword "KEY" needs to be adjusted to match the length of the plaintext "HELLO".
    - Adjusted Keyword: "KEYKE"
    - This is done by repeating the keyword until it matches the length of the plaintext.
2. **Calculate the Shift Value:**
    - The shift value is calculated using a simple hash function based on the adjusted keyword and the provided key.
    - The hash value is the sum of the ASCII values of the characters in the keyword "KEYKE":
        - K: 75
        - E: 69
        - Y: 89
        - K: 75
        - E: 69
        - Hash Value = 75 + 69 + 89 + 75 + 69 = 377
    - The key is adjusted: Key = Key * 17
        - Key = 3 * 17 = 51
    - The shift value is calculated as: Hash Value % Key
        - Shift Value = 377 % 51 = 20
3. **Encrypt Each Character:**
    - Now, each character of the plaintext "HELLO" is shifted by the calculated shift value (20).
    - **H (shift by 20) -> B**
        - 'H' is at index 7 in the alphabet.
        - New index = (7 + 20) % 26 = 1
        - The character at index 1 is 'B'.

- **E (shift by 20) -> Y**
  - 'E' is at index 4 in the alphabet.
  - New index = (4 + 20) % 26 = 24
  - The character at index 24 is 'Y'.
- **L (shift by 20) -> F**
  - 'L' is at index 11 in the alphabet.
  - New index = (11 + 20) % 26 = 5
  - The character at index 5 is 'F'.
- **L (shift by 20) -> F**
  - 'L' is at index 11 in the alphabet.
  - New index = (11 + 20) % 26 = 5
  - The character at index 5 is 'F'.
- **O (shift by 20) -> I**
  - 'O' is at index 14 in the alphabet.
  - New index = (14 + 20) % 26 = 8
  - The character at index 8 is 'I'.

4. **Cyphertext:**
   - After shifting each character, the resulting Cyphertext is "BYFFI".

Summary of Encryption:

- **Plaintext:** HELLO
- **Adjusted Keyword:** KEYKE
- **Shift Value:** 20
- **Cyphertext:** BYFFI

Step-by-Step Decryption Process:

1. **Use the Same Adjusted Keyword and Shift Value:**
   - Adjusted Keyword: "KEYKE"
   - Shift Value: 20
2. **Decrypt Each Character:**
   - Now, each character of the Cyphertext "BYFFI" is shifted back by the calculated shift value (20).
   - **B (shift back by 20) -> H**

        - 'B' is at index 1 in the alphabet.
        - New index = (1 - 20 + 26) % 26 = 7
        - The character at index 7 is 'H'.
    - **Y (shift back by 20) -> E**
        - 'Y' is at index 24 in the alphabet.
        - New index = (24 - 20 + 26) % 26 = 4
        - The character at index 4 is 'E'.
    - **F (shift back by 20) -> L**
        - 'F' is at index 5 in the alphabet.
        - New index = (5 - 20 + 26) % 26 = 11
        - The character at index 11 is 'L'.
    - **F (shift back by 20) -> L**
        - 'F' is at index 5 in the alphabet.
        - New index = (5 - 20 + 26) % 26 = 11
        - The character at index 11 is 'L'.
    - **I (shift back by 20) -> O**
        - 'I' is at index 8 in the alphabet.
        - New index = (8 - 20 + 26) % 26 = 14
        - The character at index 14 is 'O'.
3. **Plaintext:**
    - After shifting each character back, the resulting plaintext is "HELLO".

Summary of Decryption:

- **Cyphertext:** BYFFI
- **Adjusted Keyword:** KEYKE
- **Shift Value:** 20
- **Plaintext:** HELLO

Code:

```
print("\nCaesar Cypher Encryption/Decryption\n")
choice = input("Enter the operation you want to perform:
Encryption(1)/Decryption(0): ")
```

```
# Populating the alphabet table beforehand without loop to avoid any overhead
alphabetTable = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7,
'I': 8, 'J': 9, 'K': 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14, 'P': 15, 'Q': 16, 'R':
17, 'S': 18, 'T': 19, 'U': 20, 'V': 21, 'W': 22, 'X': 23, 'Y': 24, 'Z': 25}
reverseAlphabetTable = {v: k for k, v in alphabetTable.items()}

def adjustLength(keyword, input_message):
    diff = len(input_message) - len(keyword)
    newKeyword = ""
    if diff < 0:
        for i in range(len(input_message)):
            newKeyword += keyword[i]
    elif diff == 0:
        newKeyword = keyword
    else:
        for i in range(len(input_message)):
            newKeyword += keyword[i % len(keyword)]
    return newKeyword

def simpleHash(keyword, key):
    hashValue = 0
    for i in range(len(keyword)):
        hashValue += ord(keyword[i])
    key = key * 17
    return hashValue % key

def imporvedCaesarEncrypt(input_message, key, keyword, alphabetTable,
reverseAlphabetTable):
    sameLengthKeyword = adjustLength(keyword, input_message)
    shiftValue = simpleHash(sameLengthKeyword, key)
    encrypted_message = ""
    for c in input_message:
        if (65 <= ord(c) <= 90):
            encrypted_message += reverseAlphabetTable[(alphabetTable[c] +
shiftValue) % 26]
        elif (97 <= ord(c) <= 122):
            temp = ord(c) - 32
            encrypted_message += reverseAlphabetTable[(alphabetTable[chr(temp)] +
shiftValue) % 26].lower()
        else:
            encrypted_message += c
    return encrypted_message

def imporvedCaesarDecrypt(input_message, key, keyword, alphabetTable,
reverseAlphabetTable):
    sameLengthKeyword = adjustLength(keyword, input_message)
    shiftValue = simpleHash(sameLengthKeyword, key)
    decrypted_message = ""
    for c in input_message:
        if (65 <= ord(c) <= 90):
```

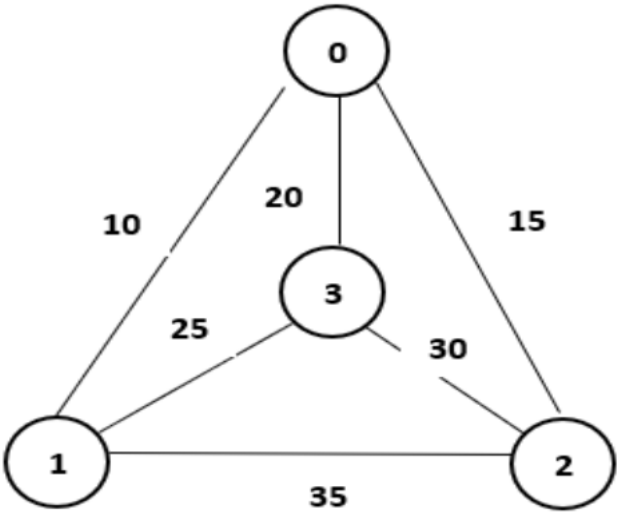
```
            decrypted_message += reverseAlphabetTable[(alphabetTable[c] -
shiftValue) % 26]
        elif (97 <= ord(c) <= 122):
            temp = ord(c) - 32
            decrypted_message += reverseAlphabetTable[(alphabetTable[chr(temp)] -
shiftValue) % 26].lower()
        else:
            decrypted_message += c
    return decrypted_message

if choice == "1":
    input_message = input("\nEnter the message you want to encrypt: ")
    key = int(input("\nEnter the encryption key you want to use: "))
    keyword = input("\nEnter the keyword you want to use: ")
    encrypted_message = imporvedCaesarEncrypt(input_message, key, keyword,
alphabetTable, reverseAlphabetTable)
    print("\nEncrypted message: ", encrypted_message, end="\n\n")
elif choice == "0":
    input_message = input("\nEnter the message you want to decrypt: ")
    key = int(input("\nEnter the decryption key you want to use: "))
    keyword = input("\nEnter the keyword you want to use: ")
    decrypted_message = imporvedCaesarDecrypt(input_message, key, keyword,
alphabetTable, reverseAlphabetTable)
    print("\nDecrypted message: ", decrypted_message, end="\n\n")
else:
    print("\nInvalid choice! Please enter 1 for encryption or 0 for decryption.")
```

Output:

```
(base) tirthshah@Tirths-MacBook-Pro PDEU-Sem-5 % python3 -u "/Users/tirthshah/Docume

Caesar Cypher Encryption/Decryption

Enter the operation you want to perform: Encryption(1)/Decryption(0): 1

Enter the message you want to encrypt: HELLO

Enter the encryption key you want to use: 3

Enter the keyword you want to use: KEY

Encrypted message:  BYFFI
```

Conclusion:

The traditional Caesar Cypher uses a fixed shift for encryption, making it simple but easily breakable. The Improved Caesar Cypher adds complexity by using a keyword-based hash to determine a variable shift, enhancing security. This added complexity makes it more resistant to basic attacks, though both Cyphers remain vulnerable due to the only 25 possible shifts for both of them.



A Travelling Salesman Problem (TSP) tour in the graph is 0 − 1 − 3 − 2 − 0. The cost of the tour is 10 + 25 + 30 + 15 = 80

A1. The Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem in computer science and operations research. The problem involves finding the shortest possible tour that visits all cities exactly once and returns to the starting city, with the objective of minimizing the total distance traveled.

TSP is an NP-hard problem, meaning that there is no known efficient algorithm to solve it optimally for large instances in polynomial time. However, various heuristic and approximate algorithms have been developed to find near-optimal solutions within a reasonable time frame.

Branch and Bound Approach: The Branch and Bound algorithm is a widely used technique for solving various optimization problems, including TSP. It is a systematic approach that explores the solution space by branching out and bounding the potential solutions based on certain criteria.

In the context of TSP, the Branch and Bound algorithm works as follows:

1. **Branching**: At each step, the algorithm creates branches by considering the possible choices for the next city to be visited. For example, if the current partial tour includes cities A and B, the algorithm will create branches for visiting city C, city D, and so on.
2. **Bounding**: For each branch, the algorithm computes a lower bound on the total cost (distance) of any tour that could be constructed by extending the current partial tour. This lower bound is obtained by using heuristics, such as the minimum spanning tree or the nearest neighbor approach.
3. **Pruning**: If the lower bound for a branch is greater than or equal to the cost of the best solution found so far, the algorithm prunes (discards) that branch, as it cannot lead to a better solution.

4. **Backtracking**: The algorithm continues exploring the remaining branches by recursively applying the branching and bounding steps until all branches have been explored or a complete tour has been found.
5. **Updating the Best Solution**: If a complete tour is found with a cost lower than the current best solution, the algorithm updates the best solution with the new tour and its cost.

Time Complexity:

The time complexity of the Branch and Bound algorithm for TSP is heavily dependent on the effectiveness of the bounding function and the order in which the branches are explored. In the worst case, where the bounding function is ineffective and all branches need to be explored, the time complexity is O(n!), which is exponential in the number of cities.

However, in practice, the Branch and Bound algorithm can significantly reduce the search space by effectively bounding and pruning branches, leading to much better performance than the brute-force approach for many instances.

Code:

```
#include <stdio.h>
#include <limits.h>

#define N 4 // Number of cities

int graph[N][N] = {
    {0, 10, 15, 20},
    {10, 0, 35, 25},
    {15, 35, 0, 30},
    {20, 25, 30, 0}
};

int minCost = INT_MAX; // Variable to store the minimum cost of TSP tour
int minPath[N + 1]; // Array to store the best path found so far

void copyArray(int dest[], int src[], int n) {
    for (int i = 0; i < n; i++) {
        dest[i] = src[i];
    }
}

int calculateBound(int path[], int level, int visited[]) {
    // Calculate lower bound using minimum spanning tree
    // For simplicity, we use the nearest neighbor heuristic here
    int bound = 0;
    int lastCity = path[level - 1];
    int minEdge = INT_MAX;
    for (int i = 0; i < N; i++) {
        if (!visited[i]) {
            if (graph[lastCity][i] < minEdge && lastCity != i) {
                minEdge = graph[lastCity][i];
```

```
            }
        }
    }
    bound += minEdge;

    // Add cost of the minimum edge from the last city to the starting city
    bound += graph[path[0]][path[level - 1]];
    return bound;
}

void tsp(int path[], int level, int cost, int visited[]) {
    if (level == N) {
        // If all cities have been visited
        if (cost + graph[path[level - 1]][path[0]] < minCost) {
            // Update minCost if this path is better
            minCost = cost + graph[path[level - 1]][path[0]];
            // Copy the path to minPath
            copyArray(minPath, path, N);
            minPath[N] = path[0]; // Add the starting city to the end to complete
the loop
        }
        return;
    }

    // Calculate lower bound
    int bound = calculateBound(path, level, visited);
    if (bound >= minCost) {
        return; // Prune this branch
    }

    for (int i = 0; i < N; i++) {
        if (!visited[i]) {
            int newPath[N];
            int newVisited[N];
            copyArray(newPath, path, level);
            copyArray(newVisited, visited, N);

            newPath[level] = i;
            newVisited[i] = 1;

            tsp(newPath, level + 1, cost + graph[path[level - 1]][i], newVisited);
        }
    }
}

int main() {
    int path[N]; // Path array to store the current TSP tour
    int visited[N] = {0}; // Array to mark visited cities
    path[0] = 0; // Start from city 0
    visited[0] = 1; // Mark city 0 as visited
```

```
    tsp(path, 1, 0, visited); // Start the Branch and Bound algorithm

    printf("Minimum cost of TSP tour: %d\n", minCost);
    printf("Path taken: ");
    for (int i = 0; i <= N; i++) {
        printf("%d ", minPath[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter the number of edges: 5
Enter the edges (start, destination, weight):
1 2 1
2 3 2
3 4 5
4 5 6
5 1 3
Number of edges in MST is 4
Weight of Minimum Spanning Tree is 11
Selected Edges are :
(1, 2) Weight: 1
(2, 3) Weight: 2
(5, 1) Weight: 3
(3, 4) Weight: 5
```

```c
#include <stdio.h>
#include <stdlib.h>

// Structure to store license information
typedef struct {
    int LicenseNo;
    float r;
    float price;
} LicenseInfo;

void printLicenseInfo(LicenseInfo licenses[], int n) {
    printf("License No.\tGrowth Rate\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%.2f\n", licenses[i].LicenseNo, licenses[i].r);
    }
}

void minimize_total_cost(LicenseInfo licenses[], int n) {

    // Sort licenses based on growth rate
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (licenses[j].r < licenses[j + 1].r) {
                LicenseInfo temp = licenses[j];
                licenses[j] = licenses[j + 1];
                licenses[j + 1] = temp;
            }
        }
    }

    // Calculate total cost
    float total_cost = 100.0;
    for (int i = 1; i < n; i++) {
        float temp = 100 * (licenses[i].r * i);
        licenses[i].price = temp;
        total_cost += temp;
    }

    printf("\nOptimal license purchase order:\n\n");

    for(int i = 0; i < n; i++) {

        if(i == n-1) {
            printf("License %d\n\n", licenses[i].LicenseNo);
        }
        else {
            printf("License %d -> ", licenses[i].LicenseNo);
        }
    }
```

```
    printf("License No.\tGrowth Rate\tPrice\n\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t\t%.2f\t\t%.2f\n", licenses[i].LicenseNo, licenses[i].r,
licenses[i].price);
    }



    printf("\nTotal cost: %.2f\n", total_cost);

}

int main() {
    int n;
    printf("Enter the number of licenses: ");
    scanf("%d", &n);

    LicenseInfo licenses[n];

    // Input growth rates for each license
    printf("Enter the growth rates for each license:\n");
    for (int i = 0; i < n; i++) {
        licenses[i].LicenseNo = i + 1;
        licenses[i].price = 100.0;
        scanf("%f", &licenses[i].r);
    }

    // Calculate and print total cost
    minimize_total_cost(licenses, n);

    return 0;
}
```

Output:

```
Enter the number of licenses: 4
Enter the growth rates for each license:
1 4 2 3

Optimal license purchase order:

License 2 -> License 4 -> License 3 -> License 1

License No.     Growth Rate     Price

2               4.00            100.00
4               3.00            300.00
3               2.00            400.00
1               1.00            300.00

Total cost: 1100.00
```

Q2. Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is unimodal. That is, for some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry $A[j]$ on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum value, and then fall from there on). You'd like to find the "peak entry" p without having to read the entire array - in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(log\ n)$ entries of A.

A2. A unimodal array is a type of array where the elements are initially in increasing order, and then at some point, they start decreasing. This means that there exists an index p such that the elements before p are in increasing order, and the elements after p are in decreasing order. The array is said to have a "peak" at index p.

Problem Statement: Given an array A with n entries, where each entry holds a distinct number, and the sequence of values A[1], A[2], ..., A[n] is unimodal, the problem is to find the "peak entry" p by reading as few entries of A as possible, with a time complexity of O(log n).

Algorithm: The algorithm to find the peak entry in a unimodal array is based on the idea of binary search. Since the array is unimodal, we can divide the search space into two parts at the midpoint and check if the midpoint is the peak entry or if the peak lies on the left or right side of the midpoint.

1. Initialize two pointers, low and high, to the first and last indices of the array, respectively.
2. Calculate the midpoint mid = (low + high) / 2.
3. Compare the values at mid, mid - 1, and mid + 1:
    - If A[mid] > A[mid + 1] and A[mid] > A[mid - 1], then mid is the peak entry. Return mid.
    - If A[mid] < A[mid + 1], then the peak entry lies on the right side of mid. Update low = mid + 1.
    - If A[mid] < A[mid - 1], then the peak entry lies on the left side of mid. Update high = mid - 1.
4. Repeat steps 2 and 3 until the peak entry is found.

Time Complexity: The time complexity of this algorithm is O(log n). In each iteration, the search space is reduced by half due to the binary search approach. Therefore, the number of iterations required to find the peak entry is proportional to log n.

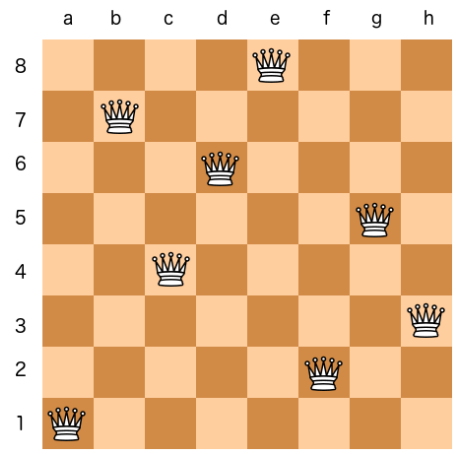
Code:

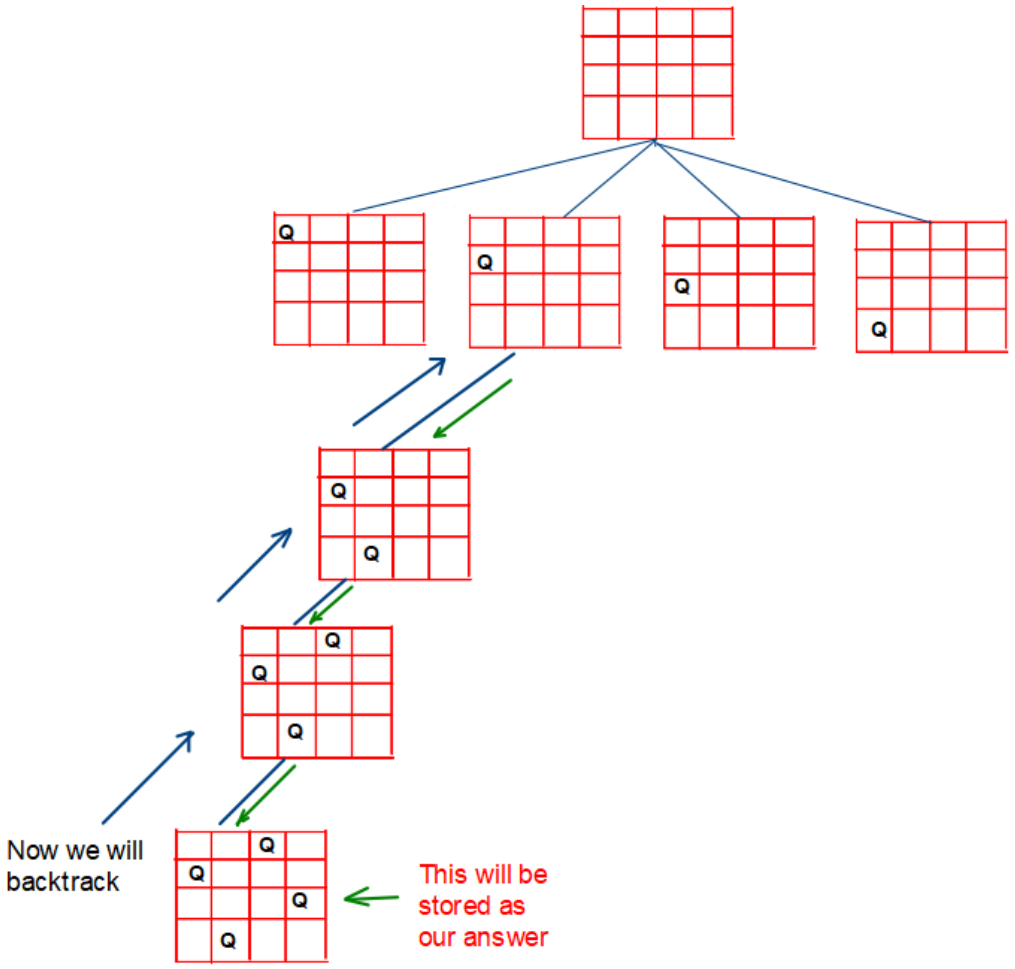
```
#include <stdio.h>

int findPeak(int A[], int n) {
    int low = 0, high = n - 1;

    while (low < high) {
```

```
        int mid = low + (high - low) / 2;

        if (A[mid] < A[mid + 1]) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }

    return low;  // or high, since low == high when the loop ends
}

int main() {
    int A[] = {1, 3, 5, 7, 6, 4, 2}; // Example unimodal array
    int n = sizeof(A) / sizeof(A[0]);

    int peakIndex = findPeak(A, n);
    int peakValue = A[peakIndex];

    printf("Peak entry index: %d\n", peakIndex);
    printf("Peak entry value: %d\n", peakValue);

    return 0;
}
```

Output:

```
Peak entry index: 3
Peak entry value: 7
```





Code:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_SIZE 20
```

```
bool isSafe(int board[MAX_SIZE][MAX_SIZE], int row, int col, int n) {
    // Check this row on the left side
    for (int i = 0; i < col; i++) {
        if (board[row][i] == 1) {
            return false;
        }
    }

    // Check upper diagonal on the left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // Check lower diagonal on the left side
    for (int i = row, j = col; i < n && j >= 0; i++, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}

bool solveNQueens(int board[MAX_SIZE][MAX_SIZE], int col, int n) {
    // Base case: If all queens are placed, return true
    if (col == n) {
        return true;
    }

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < n; i++) {
        if (isSafe(board, i, col, n)) {
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQueens(board, col + 1, n)) {
                return true;
            }

            // If placing queen in board[i][col] doesn't lead to a solution, remove
the queen
            board[i][col] = 0;
        }
    }

    // If the queen cannot be placed in any row in this column, return false
    return false;
}
```

```c
void printSolution(int board[MAX_SIZE][MAX_SIZE], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int n;
    printf("Enter the number of queens: ");
    scanf("%d", &n);

    int board[MAX_SIZE][MAX_SIZE] = {0};

    if (solveNQueens(board, 0, n)) {
        printf("Solution:\n");
        printSolution(board, n);
    } else {
        printf("No solution exists for %d queens.\n", n);
    }

    return 0;
}
```

Output:

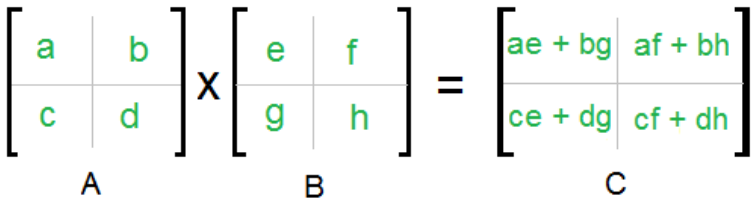
```
Enter the number of queens: 6
Solution:
0 0 0 1 0 0
1 0 0 0 0 0
0 0 0 0 1 0
0 1 0 0 0 0
0 0 0 0 0 1
0 0 1 0 0 0
```

Time Complexity:

The time complexity of the backtracking solution with dynamic programming for the N-Queens problem is O(N!), as in the worst case, the algorithm needs to explore all possible permutations of queen placements.

However, the dynamic programming optimization can significantly reduce the number of redundant computations, leading to improved efficiency, especially for larger board sizes.

Traditional Matrix Multiplication Algorithm Using Divide and Conquer:



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Recurrence Relation: $T(n) = 8\,T\left(\frac{n}{2}\right) + O(n^2)$
Time Complexity: $T(n) = O(n^3) = \theta\,(n^3) = \Omega\,(n^3)$

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int **matrix_allocate(int row, int column) {
    int **matrix = malloc(row * sizeof(int*));
    for (int i = 0; i < row; i++) {
        matrix[i] = calloc(column, sizeof(int));
    }
    return matrix;
}

void matrix_free(int **matrix, int row) {
    for (int i = 0; i < row; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void matrix_print(int **a, int row) {

    int i, j;

    for (i=0 ; i<row ; i++) {
        for (j=0 ; j<row ; j++)
            printf("%3d ", a[i][j]);
        printf("\n");
    }
```

```
    printf("\n");
}

int **matrix_add(int **a, int **b, int row) {

    int **c = matrix_allocate(row, row);

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < row; j++) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }

    return c;
}

int **matrix_subtract(int **a, int **b, int row) {

    int **c = matrix_allocate(row, row);

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < row; j++) {
            c[i][j] = a[i][j] - b[i][j];
        }
    }

    return c;
}

int **matrix_multiply(int **A, int **B, int row) {

    int **C = matrix_allocate(row, row);

    if (row == 1) {
        C[0][0] = A[0][0] * B[0][0];
    }

    else {
        int row2 = row / 2;
        int **a11 = matrix_allocate(row2, row2);
        int **a12 = matrix_allocate(row2, row2);
        int **a21 = matrix_allocate(row2, row2);
        int **a22 = matrix_allocate(row2, row2);
        int **b11 = matrix_allocate(row2, row2);
        int **b12 = matrix_allocate(row2, row2);
        int **b21 = matrix_allocate(row2, row2);
        int **b22 = matrix_allocate(row2, row2);

        for (int i = 0; i < row2; i++) {
            for (int j = 0; j < row2; j++) {
                a11[i][j] = A[i][j];
```

```
                a12[i][j] = A[i][j + row2];
                a21[i][j] = A[i + row2][j];
                a22[i][j] = A[i + row2][j + row2];
                b11[i][j] = B[i][j];
                b12[i][j] = B[i][j + row2];
                b21[i][j] = B[i + row2][j];
                b22[i][j] = B[i + row2][j + row2];
            }
        }

        int **c11 = matrix_add(matrix_multiply(a11, b11, row2),
                               matrix_multiply(a12, b21, row2), row2);
        int **c12 = matrix_add(matrix_multiply(a11, b12, row2),
                               matrix_multiply(a12, b22, row2), row2);
        int **c21 = matrix_add(matrix_multiply(a21, b11, row2),
                               matrix_multiply(a22, b21, row2), row2);
        int **c22 = matrix_add(matrix_multiply(a21, b12, row2),
                               matrix_multiply(a22, b22, row2), row2);

        for (int i = 0; i < row2; i++) {
            for (int j = 0; j < row2; j++) {
                C[i][j] = c11[i][j];
                C[i][j + row2] = c12[i][j];
                C[i + row2][j] = c21[i][j];
                C[i + row2][j + row2] = c22[i][j];
            }
        }
    }

    return C;
}
```

Strassen Matrix Multiplication Algorithm:

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Recurrence Relation: $T(n) = 7\,T\left(\frac{n}{2}\right) + O(n^2)$

Time Complexity: $T(n) = O\left(n^{\log_2 7}\right) = O(n^{2.8074}) = \theta\,(n^{2.8074}) = \Omega\,(n^{2.8074})$

Code:

```
int** strassen_mult(int **A, int **B, int row) {

    int **C = matrix_allocate(row, row);

    if (row == 1) {
        C[0][0] = A[0][0] * B[0][0];
    }

    else {

        int row2 = row / 2;
        int **a11 = matrix_allocate(row2, row2);
        int **a12 = matrix_allocate(row2, row2);
        int **a21 = matrix_allocate(row2, row2);
        int **a22 = matrix_allocate(row2, row2);
        int **b11 = matrix_allocate(row2, row2);
        int **b12 = matrix_allocate(row2, row2);
        int **b21 = matrix_allocate(row2, row2);
        int **b22 = matrix_allocate(row2, row2);

        for (int i = 0; i < row2; i++) {
            for (int j = 0; j < row2; j++) {
                a11[i][j] = A[i][j];
                a12[i][j] = A[i][j + row2];
                a21[i][j] = A[i + row2][j];
                a22[i][j] = A[i + row2][j + row2];
```

```
                b11[i][j] = B[i][j];
                b12[i][j] = B[i][j + row2];
                b21[i][j] = B[i + row2][j];
                b22[i][j] = B[i + row2][j + row2];
            }
        }

        int **s1 = matrix_subtract(b12, b22, row2);
        int **s2 = matrix_add(a11, a12, row2);
        int **s3 = matrix_add(a21, a22, row2);
        int **s4 = matrix_subtract(b21, b11, row2);
        int **s5 = matrix_add(a11, a22, row2);
        int **s6 = matrix_add(b11, b22, row2);
        int **s7 = matrix_subtract(a12, a22, row2);
        int **s8 = matrix_add(b21, b22, row2);
        int **s9 = matrix_subtract(a11, a21, row2);
        int **s10 = matrix_add(b11, b12, row2);

        int **p1 = strassen_mult(a11, s1, row2);
        int **p2 = strassen_mult(s2, b22, row2);
        int **p3 = strassen_mult(s3, b11, row2);
        int **p4 = strassen_mult(a22, s4, row2);
        int **p5 = strassen_mult(s5, s6, row2);
        int **p6 = strassen_mult(s7, s8, row2);
        int **p7 = strassen_mult(s9, s10, row2);

        int **c11 = matrix_add(matrix_subtract(matrix_add(p5, p4, row2), p2, row2),
p6, row2);
        int **c12 = matrix_add(p1, p2, row2);
        int **c21 = matrix_add(p3, p4, row2);
        int **c22 = matrix_subtract(matrix_subtract(matrix_add(p5, p1, row2), p3,
row2), p7, row2);

        for (int i = 0; i < row2; i++) {
            for (int j = 0; j < row2; j++) {
                C[i][j] = c11[i][j];
                C[i][j + row2] = c12[i][j];
                C[i + row2][j] = c21[i][j];
                C[i + row2][j + row2] = c22[i][j];
            }
        }

        matrix_free(a11, row2);
        matrix_free(a12, row2);
        matrix_free(a21, row2);
        matrix_free(a22, row2);
        matrix_free(b11, row2);
        matrix_free(b12, row2);
        matrix_free(b21, row2);
        matrix_free(b22, row2);
        matrix_free(s1, row2);
```

```
        matrix_free(s2, row2);
        matrix_free(s3, row2);
        matrix_free(s4, row2);
        matrix_free(s5, row2);
        matrix_free(s6, row2);
        matrix_free(s7, row2);
        matrix_free(s8, row2);
        matrix_free(s9, row2);
        matrix_free(s10, row2);
        matrix_free(p1, row2);
        matrix_free(p2, row2);
        matrix_free(p3, row2);
        matrix_free(p4, row2);
        matrix_free(p5, row2);
        matrix_free(p6, row2);
        matrix_free(p7, row2);
    }

    return C;
}

int main() {

    int n = 8;
    int **A = matrix_allocate(n, n);
    int **B = matrix_allocate(n, n);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = i + j;
            B[i][j] = i + j;
        }
}

    printf("Matrix A:\n");
    matrix_print(A, n);

    printf("Matrix B:\n");
    matrix_print(B, n);

    int **C = matrix_multiply(A, B, n);
    int **D = strassen_mult(A, B, n);

    printf("Matrix C:\n");
    matrix_print(C, n);

    printf("Matrix D:\n");
    matrix_print(D, n);

    matrix_free(A, n);
    matrix_free(B, n);
```

```
    matrix_free(C, n);
    matrix_free(D, n);

    return 0;
}
```

Output:

```
Matrix A:
  0   1   2   3   4   5   6   7
  1   2   3   4   5   6   7   8
  2   3   4   5   6   7   8   9
  3   4   5   6   7   8   9  10
  4   5   6   7   8   9  10  11
  5   6   7   8   9  10  11  12
  6   7   8   9  10  11  12  13
  7   8   9  10  11  12  13  14

Matrix B:
  0   1   2   3   4   5   6   7
  1   2   3   4   5   6   7   8
  2   3   4   5   6   7   8   9
  3   4   5   6   7   8   9  10
  4   5   6   7   8   9  10  11
  5   6   7   8   9  10  11  12
  6   7   8   9  10  11  12  13
  7   8   9  10  11  12  13  14

Matrix C:
140 168 196 224 252 280 308 336
168 204 240 276 312 348 384 420
196 240 284 328 372 416 460 504
224 276 328 380 432 484 536 588
252 312 372 432 492 552 612 672
280 348 416 484 552 620 688 756
308 384 460 536 612 688 764 840
336 420 504 588 672 756 840 924

Matrix D:
140 168 196 224 252 280 308 336
168 204 240 276 312 348 384 420
196 240 284 328 372 416 460 504
224 276 328 380 432 484 536 588
252 312 372 432 492 552 612 672
280 348 416 484 552 620 688 756
308 384 460 536 612 688 764 840
336 420 504 588 672 756 840 924
```