CAP538: ALGORITHM DESIGN AND ANALYSIS

CONTINUOUS ASSESSMENTS (C.A)-2

ST_NAME	**: -**	**EKHLAKH AHMAD**
REG NO.	**: -**	**12209166**
ROLL NO.	**: -**	**03**
SECTION	**: -**	**REC72**
GROUP	**: -**	**01**

Q1. Explain the concept of Optimal Binary Search Trees (OBST) in the context of dynamic programming. Describe the key steps involved in constructing an OBST using dynamic programming. Provide a detailed example to illustrate how dynamic programming is applied to solve the OBST problem, including the formulation of the recurrence relations and the construction of the optimal tree.

Introduction

Optimal Binary Search Trees (OBST) are a fascinating topic within the realm of data structures and algorithms, specifically in the context of dynamic programming. The primary goal of an OBST is to construct a binary search tree (BST) that minimizes the expected search time for a given set of keys and their corresponding access probabilities. This assignment will explore the concept of OBST, the key steps involved in constructing an OBST using dynamic programming and provide a detailed example to illustrate the application of dynamic programming to solve the OBST problem.

Concept of Optimal Binary Search Trees

In a typical binary search tree, the expected search time depends on the structure of the tree. For a given set of keys, different BST structures can have different expected search times. An OBST is a BST that ensures the minimum expected search time, considering the access probabilities of the keys.

The OBST problem can be formally defined as follows:

- We are given n keys, K={k1,k2,…,kn}K = \{k_1, k_2, \ldots, k_n\}K={k1,k2,…,kn}, in sorted order.
- Each key kik_iki has a probability pip_ipi of being searched.
- There are also n+1 dummy keys d0,d1,…,dnd_0, d_1, \ldots, d_nd0,d1,…,dn, where did_idi represents the probability of searches falling between ki−1k_{i-1}ki−1 and kik_iki.

The goal is to construct a BST such that the expected search cost is minimized.
Key Steps in Constructing an OBST Using Dynamic Programming

1. Initialization: Create two tables, e and w, of size (n+1) x (n+1) to store the expected search costs and probabilities, respectively. Additionally, create a root table r to store the roots of subtrees.

2. Base Case: Initialize the diagonal entries of e and w. For i=1i = 1i=1 to n+1n+1n+1:

o e[i][i−1]=qi−1e[i][i-1] = q_{i-1}e[i][i−1]=qi−1 (cost when there are no keys)
o w[i][i−1]=qi−1w[i][i-1] = q_{i-1}w[i][i−1]=qi−1 (probability of dummy keys)

3. Recursive Case: Fill the tables for increasing lengths of subtrees. For each subtree length lll from 1 to n:

o For i=1i = 1i=1 to n−l+1n-l+1n−l+1:
  Let j=i+l−1j = i + l - 1j=i+l−1
  Set e[i][j]=∞e[i][j] = \inftye[i][j]=∞
  Calculate w[i][j]=w[i][j−1]+pj+qjw[i][j] = w[i][j-1] + p_j + q_jw[i][j]=w[i][j−1]+pj+qj
  For each possible root kkk from iii to jjj:
  Compute the cost t=e[i][k−1]+e[k+1][j]+w[i][j]t = e[i][k-1] + e[k+1][j] + w[i][j]t=e[i][k−1]+e[k+1][j]+w[i][j]
  Update e[i][j]e[i][j]e[i][j] and r[i][j]r[i][j]r[i][j] if ttt is smaller than the current e[i][j]e[i][j]e[i][j]

4. Construction of OBST: Use the root table r to construct the tree by recursively choosing the root nodes and their left and right subtrees.

Detailed Example
Let's consider an example with 3 keys: K={k1,k2,k3}K = \{k_1, k_2, k_3\}K={k1,k2,k3}, and their probabilities P={0.2,0.5,0.3}P = \{0.2, 0.5, 0.3\}P={0.2,0.5,0.3}. The dummy key probabilities are Q={0.1,0.1,0.1,0.1}Q = \{0.1, 0.1, 0.1, 0.1\}Q={0.1,0.1,0.1,0.1}.

Step-by-Step Construction

1. Initialization:

o e[i][i−1]=qi−1e[i][i-1] = q_{i-1}e[i][i−1]=qi−1
o w[i][i−1]=qi−1w[i][i-1] = q_{i-1}w[i][i−1]=qi−1

```
e[1][0] = 0.1
e[2][1] = 0.1
e[3][2] = 0.1
e[4][3] = 0.1

w[1][0] = 0.1
w[2][1] = 0.1
w[3][2] = 0.1
w[4][3] = 0.1
```

2. **Recursive Case:**

o For length l=1l = 1l=1:

```
w[1][1] = w[1][0] + p_1 + q_1 = 0.1 + 0.2 + 0.1 = 0.4
w[2][2] = w[2][1] + p_2 + q_2 = 0.1 + 0.5 + 0.1 = 0.7
w[3][3] = w[3][2] + p_3 + q_3 = 0.1 + 0.3 + 0.1 = 0.5

e[1][1] = e[1][0] + e[2][1] + w[1][1] = 0.1 + 0.1 + 0.4 = 0.6
e[2][2] = e[2][1] + e[3][2] + w[2][2] = 0.1 + 0.1 + 0.7 = 0.9
e[3][3] = e[3][2] + e[4][3] + w[3][3] = 0.1 + 0.1 + 0.5 = 0.7

r[1][1] = 1
r[2][2] = 2
r[3][3] = 3
```

```
w[1][2] = w[1][1] + p_2 + q_2 = 0.4 + 0.5 + 0.1 = 1.0
w[2][3] = w[2][2] + p_3 + q_3 = 0.7 + 0.3 + 0.1 = 1.1

For e[1][2]:
k = 1: t = e[1][0] + e[2][2] + w[1][2] = 0.1 + 0.9 + 1.0 = 2.0
k = 2: t = e[1][1] + e[3][2] + w[1][2] = 0.6 + 0.1 + 1.0 = 1.7
e[1][2] = 1.7, r[1][2] = 2

For e[2][3]:
k = 2: t = e[2][1] + e[3][3] + w[2][3] = 0.1 + 0.7 + 1.1 = 1.9
k = 3: t = e[2][2] + e[4][3] + w[2][3] = 0.9 + 0.1 + 1.1 = 2.1
e[2][3] = 1.9, r[2][3] = 2
```

- For length l=2l = 2l=2:

```
w[1][3] = w[1][2] + p_3 + q_3 = 1.0 + 0.3 + 0.1 = 1.4

For e[1][3]:
k = 1: t = e[1][0] + e[2][3] + w[1][3] = 0.1 + 1.9 + 1.4 = 3.4
k = 2: t = e[1][1] + e[3][3] + w[1][3] = 0.6 + 0.7 + 1.4 = 2.7
k = 3: t = e[1][2] + e[4][3] + w[1][3] = 1.7 + 0.1 + 1.4 = 3.2
e[1][3] = 2.7, r[1][3] = 2
```

3. Construction of OBST:
   - o From r[1][3] = 2, k2k_2k2 is the root.
   - o Left subtree: k1k_1k1 (since r[1][1]=1r[1][1] = 1r[1][1]=1)
   - o Right subtree: k3k_3k3 (since r[3][3]=3r[3][3] = 3r[3][3]=3)

Q2. Discuss the application of the Branch and Bound technique to solve the Travelling Salesperson Problem (TSP). Outline the general approach of Branch and Bound and explain how it can be tailored to tackle the TSP efficiently. Provide a step-by-step explanation of the Branch and Bound algorithm for TSP, including the selection of bounding functions, pruning strategies, and the backtracking mechanism.

Introduction

The Travelling Salesperson Problem (TSP) is a classic optimization problem in which a salesperson must visit a set of cities exactly once and return to the starting city, minimizing the total travel distance or cost. The Branch and Bound technique is a

powerful method to solve TSP, especially for small to medium-sized instances, by systematically exploring and pruning the search space.
Branch and Bound Technique

The Branch and Bound technique is a general algorithmic method for solving combinatorial optimization problems. It involves the following key components:
1. Branching: Decomposing the problem into smaller subproblems.
2. Bounding: Calculating a bound on the optimal solution for a subproblem.
3. Pruning: Discarding subproblems that cannot yield a better solution than the current best.
4. Backtracking: Exploring different branches and backtracking when necessary to find the optimal solution.

Application of Branch and Bound to TSP

To apply Branch and Bound to the TSP, we follow these steps:
1. Branching: Generate subproblems by selecting edges to include in the tour.
2. Bounding: Calculate a lower bound on the total cost of any tour that includes the selected edges.
3. Pruning: Discard subproblems with bounds greater than the current best solution.
4. Backtracking: Explore other subproblems and backtrack if necessary to find the optimal solution.

Step-by-Step Explanation of the Branch and Bound Algorithm for TSP
Step 1: Initialization
- Start with an initial upper bound on the tour cost, typically obtained using a heuristic (e.g., the nearest neighbor algorithm).
- Create a priority queue to store subproblems, prioritized by their lower bounds.

Step 2: Branching
- At each node, generate subproblems by including and excluding an edge (i, j).
- For each subproblem, calculate a lower bound on the tour cost.

Step 3: Bounding
- Use a bounding function to calculate a lower bound for each subproblem. A common method is to use the Minimum Spanning Tree (MST) of the remaining cities plus the cost of the selected edges.

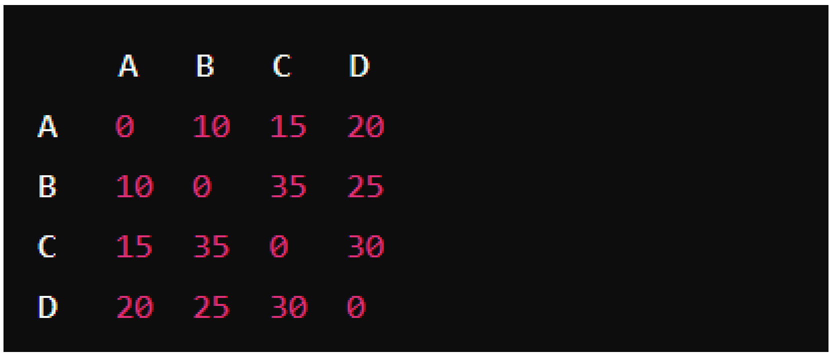
Step 4: Pruning
- If the lower bound of a subproblem exceeds the current best solution, prune (discard) that subproblem.
- If a complete tour is found with a cost lower than the current best, update the best solution.

Step 5: Backtracking

- Continue exploring subproblems in the priority queue.
- Use backtracking to explore different branches, ensuring all potential tours are considered.

Example

Let's consider a TSP instance with 4 cities (A, B, C, D) and the following distance matrix:



```
     A   B   C   D
A    0   10  15  20
B    10  0   35  25
C    15  35  0   30
D    20  25  30  0
```

Step-by-Step Process:

1. **Initialization:**
   - o Initial upper bound using a heuristic (e.g., nearest neighbor): A -> B -> D -> C -> A with a cost of 80.
   - o Priority queue initialized with the root node (no edges selected).

2. **Branching:**
   - o From the root node, branch to subproblems by including edges (A, B) and (A, C).

3. **Bounding:**
   - o Calculate bounds for each subproblem:
     For (A, B): Calculate MST for the remaining cities (B, C, D) and add the cost of (A, B).
     For (A, C): Calculate MST for the remaining cities (C, B, D) and add the cost of (A, C).

4. **Pruning:**
   - o Prune subproblems with bounds greater than 80 (initial upper bound).

5. Backtracking:
   - o Explore remaining subproblems, updating the best solution and pruning as necessary.

Detailed Example:

1. Initialization:
   - o Heuristic tour: A -> B -> D -> C -> A with a cost of 80.
   - o Priority queue: [Root node].
2. Branching and Bounding:

o Node 0 (root):
Branch: Include (A, B).
Bound: Lower bound using MST (A, B, C, D): A -> B + MST(B, C, D) = 10 + MST(10, 35, 25) = 10 + 45 = 55.
Branch: Include (A, C).
Bound: Lower bound using MST (A, C, B, D): A -> C + MST(C, B, D) = 15 + MST(35, 25, 30) = 15 + 55 = 70.
o Priority queue: [Node (A, B) with bound 55, Node (A, C) with bound 70].
3. Pruning and Backtracking:
o Node (A, B) with bound 55:
Branch: Include (B, C).
Bound: Lower bound using MST (A, B, C, D): A -> B -> C + MST(C, D) = 10 + 35 + 25 = 70.
Branch: Include (B, D).
Bound: Lower bound using MST (A, B, D, C): A -> B -> D + MST(D, C) = 10 + 25 + 30 = 65.
o Priority queue: [Node (A, C) with bound 70, Node (A, B, B, C) with bound 70, Node (A, B, B, D) with bound 65].
o Continue exploring nodes, updating the best solution, and pruning as necessary.

Q3. Compare and contrast the Knuth-Morris-Pratt (KMP) algorithm with the brute-force approach for pattern matching. Explain the rationale behind the development of the KMP algorithm and how it improves upon the efficiency of the brute-force method. Provide a detailed walkthrough of the KMP algorithm, highlighting its key components such as the preprocessing phase (failure function computation) and the matching phase. Illustrate the application of the KMP algorithm with a concrete example, demonstrating its advantages over brute-force in terms of time complexity.

Introduction
Pattern matching is a fundamental problem in computer science, where the goal is to find occurrences of a "pattern" within a "text". The brute-force approach and the Knuth-Morris-Pratt (KMP) algorithm are two well-known methods for solving this problem.

Brute-Force Approach
Description
The brute-force approach, also known as the naive approach, involves checking for the pattern at every possible position in the text. It starts from the beginning of the text and attempts to match the pattern. If a mismatch is found, the pattern is shifted one position to the right and the matching process is repeated.
Time Complexity

The worst-case time complexity of the brute-force approach is O(m·n)O(m \cdot n)O(m·n), where mmm is the length of the text and nnn is the length of the pattern. This is because, in the worst case, the algorithm may have to check each character of the text mmm times.

Knuth-Morris-Pratt (KMP) Algorithm

Rationale

The KMP algorithm was developed to improve the efficiency of the brute-force method by avoiding unnecessary comparisons. It achieves this by pre-processing the pattern to create a "failure function" (also known as the "partial match table" or "prefix function"), which provides information about how the pattern itself matches against shifts of itself.

Improvements over Brute-Force

The KMP algorithm improves upon the brute-force approach by ensuring that each character of the text is only scanned once, leading to a time complexity of O(m+n)O(m + n)O(m+n). This is achieved by using the failure function to skip over portions of the text that have already been matched.

Detailed Walkthrough of the KMP Algorithm

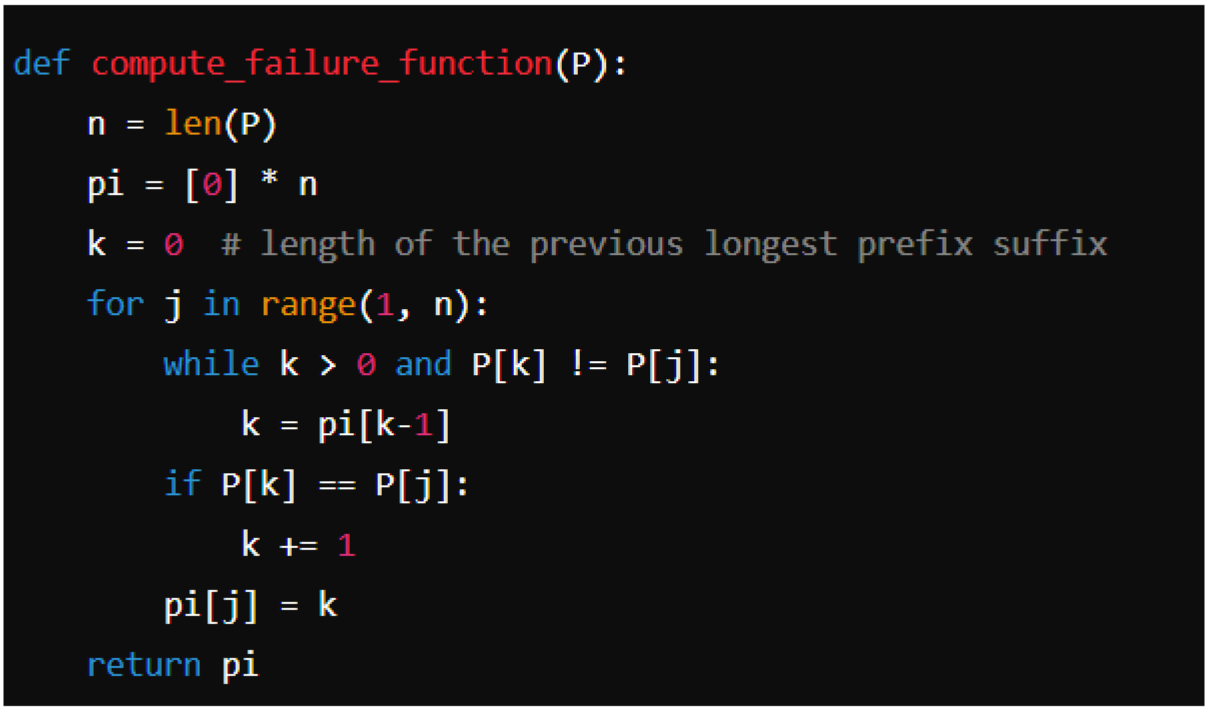
Key Components

1. Preprocessing Phase (Failure Function Computation):
   - o The failure function π\piπ for the pattern is computed to determine the longest proper prefix of the pattern that is also a suffix for each prefix of the pattern.
   - o This function is used to avoid unnecessary re-checking of characters in the text.
2. Matching Phase:
   - o The pattern is matched against the text using the failure function to skip unnecessary comparisons.

Preprocessing Phase (Failure Function Computation)

The failure function π\piπ is an array where π[j]\pi[j]π[j] indicates the length of the longest proper prefix of the pattern P[0…j]P[0 \ldots j]P[0…j] which is also a suffix of this substring.

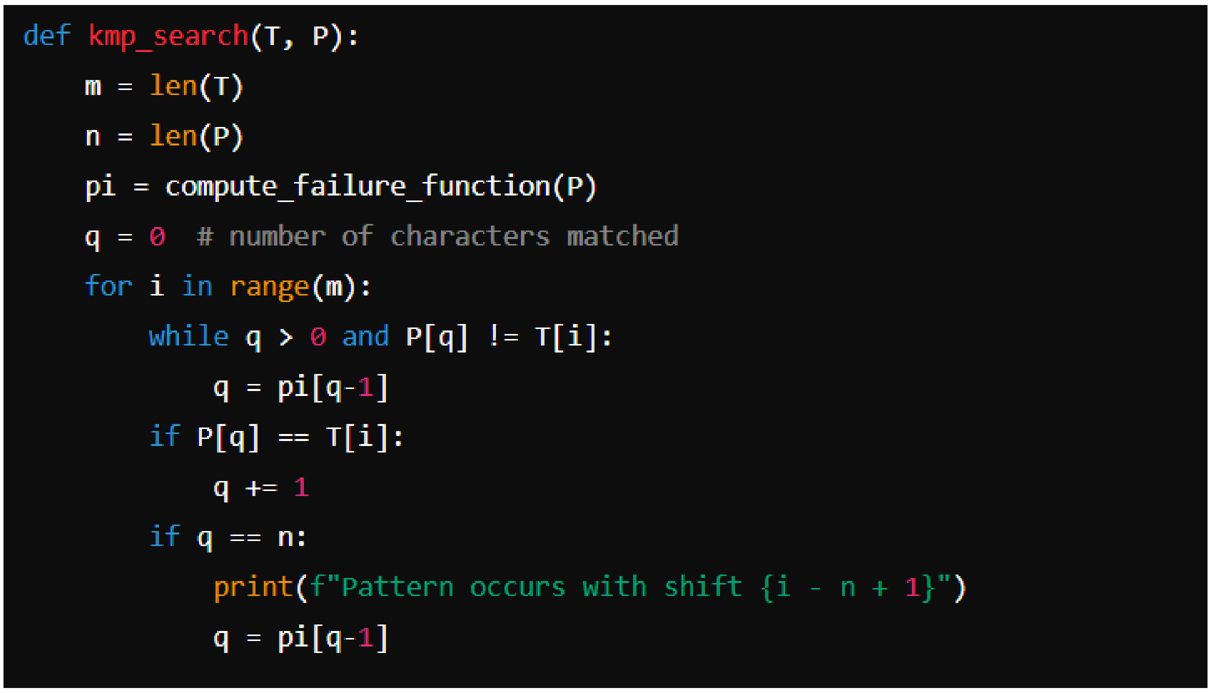
Algorithm for computing the failure function:



```python
def compute_failure_function(P):
    n = len(P)
    pi = [0] * n
    k = 0  # length of the previous longest prefix suffix
    for j in range(1, n):
        while k > 0 and P[k] != P[j]:
            k = pi[k-1]
        if P[k] == P[j]:
            k += 1
        pi[j] = k
    return pi
```

Matching Phase
Using the failure function, the pattern is matched against the text:
Python

Copy code



```python
def kmp_search(T, P):
    m = len(T)
    n = len(P)
    pi = compute_failure_function(P)
    q = 0  # number of characters matched
    for i in range(m):
        while q > 0 and P[q] != T[i]:
            q = pi[q-1]
        if P[q] == T[i]:
            q += 1
        if q == n:
            print(f"Pattern occurs with shift {i - n + 1}")
            q = pi[q-1]
```

Example to Illustrate KMP Algorithm
Consider the text T="ABABDABACDABABCABAB"T = "ABABDABACDABABCABAB"T="ABABDABACDABABCABAB" and the pattern P="ABABCABAB"P = "ABABCABAB"P="ABABCABAB".

1. **Compute the Failure Function:**
o For the pattern "ABABCABAB""ABABCABAB""ABABCABAB":

```
P:  A B A B C A B A B
π:  0 0 1 2 0 1 2 3 4
```

2. **Match the Pattern against the Text:**
o Text: "ABABDABACDABABCABAB"
o Pattern: "ABABCABAB"

```
Initial state: q = 0, i = 0
T[0] matches P[0], q = 1
T[1] matches P[1], q = 2
T[2] matches P[2], q = 3
T[3] matches P[3], q = 4
T[4] doesn't match P[4], q = π[3] = 2
...
Continue matching...
Pattern found at index 10
```

Comparison of Time Complexity

- Brute-Force: O(m·n)O(m \cdot n)O(m·n)
- KMP: O(m+n)O(m + n)O(m+n)

The KMP algorithm is significantly more efficient than the brute-force approach, especially for large texts and patterns.