DATA–DRIVEN CROP RECOMMENDATION SYSTEM USING DYNAMIC ENVIRONMENTAL INPUTS

A PROJECT REPORT

Submitted by

Dhwanit Sharma (DL.AI.U4AID24111)
Kanan Goel (DL.AI.U4AID24017)
Parkhi Mahajan (DL.AI.U4AID24126)
Punit Lohan (DL.AI.U4AID24127)

in partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY (B.Tech) IN ARTIFICIAL INTELLIGENCE AND DATA SCIENCE (AIDS)

Under the guidance of

Dr. Vijay Kumar Soni

Submitted to

AMRITA VISHWA VIDYAPEETHAM
AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
FARIDABAD – 121002

December 2025

BONAFIDE CERTIFICATE

This is to certify that this project report entitled "**Data–Driven Crop Recommendation System Using Dynamic Environmental Inputs**" is the bonafide work of Dhwanit Sharma (DL.AI.U4AID24111), Kanan Goel (DL.AI.U4AID24017), Parkhi Mahajan (DL.AI.U4AID24126), Punit Lohan (DL.AI.U4AID24127), who carried out the project work under my supervision.

Dr. Vijay Kumar Soni (Assistant Professor)
School of AI
Faridabad

DECLARATION BY THE CANDIDATE

I declare that the report entitled "**Data–Driven Crop Recommendation System Using Dynamic Environmental Inputs**" submitted by me for the degree of Bachelor of Technology is the record of the project work carried out by me under the guidance of **Dr. Vijay Kumar Soni** and this work has not formed the basis for the award of any degree, diploma, associateship, fellowship, or title in this or any other University or other similar institution of higher learning.

Dhwanit Sharma (DL.AI.U4AID24111)
Kanan Goel (DL.AI.U4AID24017)
Parkhi Mahajan (DL.AI.U4AID24126)
Punit Lohan (DL.AI.U4AID24127)

ACKNOWLEDGEMENT

This project work would not have been possible without the contribution of many people. It gives me immense pleasure to express my profound gratitude to our honorable Chancellor **Sri Mata Amritanandamayi Devi**, for her blessings and for being a source of inspiration. I am indebted to extend my gratitude to our Principal, **Dr. Lakshmi Mohandas**, Amrita School of Computing and Engineering, for facilitating us with all the facilities and extended support to gain valuable education and learning experience.

I wish to express my sincere gratitude to my supervisor(s) **Dr. Vijay Kumar Soni** for their personal involvement and constant encouragement during the entire course of this work.

I am grateful to the Project Supervisor, Review Panel Members, and the entire faculty of the Department of Artificial Intelligence and Data Science for their constructive criticisms and valuable suggestions which have been a rich source to improve the quality of this work.

Dhwanit Sharma (DL.AI.U4AID24111)
Kanan Goel (DL.AI.U4AID24017)
Parkhi Mahajan (DL.AI.U4AID24126)
Punit Lohan (DL.AI.U4AID24127)

ABSTRACT

Unexpected rainfall significantly affects agricultural planning by altering soil moisture and water availability, forcing farmers to rapidly reconsider crop selection decisions. Manual decision-making under such conditions is time-consuming and often suboptimal, especially when multiple crops and constraints must be evaluated simultaneously.

This project presents a data-structure-driven crop recommendation system that assists in selecting suitable crops under limited water availability. Crop information is efficiently stored and retrieved using hash tables with linked list chaining, while crops are organized and ranked using binary search trees, AVL trees, and max heaps. The crop selection problem is modeled as a 0/1 knapsack optimization problem and solved using Dynamic Programming to obtain an optimal solution. Greedy, Backtracking, and Branch and Bound algorithms are also implemented to provide fast approximations and validate correctness.

The system is implemented as a console-based Java application and emphasizes algorithmic efficiency, correctness, and explainability rather than machine learning. Experimental results demonstrate that the Dynamic Programming approach consistently produces the optimal crop combination under water constraints, while heuristic methods offer faster but approximate solutions. The project effectively illustrates the application of classical data structures and algorithms to a real-world agricultural decision-making problem.

TABLE OF CONTENTS

1 Introduction

Agriculture is highly sensitive to rainfall variations, and unexpected rainfall can significantly disrupt planned crop cultivation. Sudden changes in soil moisture and water availability force farmers to quickly reassess which crops are suitable under the new conditions. Making such decisions manually is slow and often inaccurate, especially when multiple factors must be considered simultaneously.
From a computational viewpoint, this situation can be modeled as a constrained optimization problem where crops must be selected to maximize profit while satisfying water availability constraints. Efficient solutions require fast data storage, quick retrieval, and optimal selection mechanisms.

This project presents a decision support system developed using classical data structures and algorithms. The system stores crop data using hash tables and linked lists, organizes and ranks crops using trees and heaps, and applies optimization techniques such as Dynamic Programming, Greedy algorithms, Backtracking, and Branch & Bound to recommend suitable crops after unexpected rainfall. The focus is on algorithmic correctness, efficiency, and explainability rather than machine learning.

1.1 Literature Review

Previous work in crop recommendation uses statistical models, expert systems, and machine learning. While effective, these approaches suffer from high computational cost and low interpretability. Optimization problems similar to crop selection are commonly solved using the 0/1 Knapsack problem through Dynamic Programming, Greedy, and Branch & Bound techniques [6, 1, 4].

1.2 Motivation

Most existing crop recommendation systems rely on machine learning models that require large datasets and act as black boxes. This project demonstrates how deterministic algorithms can provide fast and explainable decisions immediately after rainfall.

1.3 Problem Scenario

Unexpected rainfall significantly impacts agricultural planning by altering soil moisture and water availability. Farmers are often forced to quickly revise crop choices under these changing conditions in order to avoid losses and ensure profitability. Making such decisions manually is difficult due to the number of crops involved and the constraints on available water.

1.4 Problem Statement

Given a set of crops with known water requirements and expected profits, and a fixed amount of available water after rainfall, determine the optimal subset of crops that maximizes total profit without exceeding the water constraint.

1.5 Existing Technologies and Limitations

Existing Technology	**Limitations**
Machine Learning Models	Require large datasets, lack explainability, overkill for small-scale decisions
Rule-Based Systems	Rigid, not scalable, difficult to optimize
Manual Expert Judgment	Time-consuming, subjective, error-prone
Simple Sorting-Based Tools	Cannot handle multi-constraint optimization

Table 1.1: Existing Technologies and Their Limitations

These systems often fail to provide optimal solutions under strict resource constraints or do not justify why a particular crop combination was chosen [6].

1.6 Proposed System and Scope

The proposed system uses Hash Tables, Linked Lists, AVL Trees, Heaps, and Optimization Algorithms to efficiently store data, rank crops, and compute optimal selections. The scope is limited to single-season crop selection under water constraints using a console-based Java implementation, focusing on algorithmic correctness and performance.

2 Objectives

The main objectives of the project are:

- To efficiently store and retrieve crop data using a hash table with separate chaining, enabling fast access to crop attributes.
- To rank crops based on profitability using a max-heap for quick identification of high-profit crops.
- To determine the optimal combination of crops under water constraints by modeling the problem as a 0/1 knapsack optimization problem.
- To implement and compare multiple algorithmic strategies, including Dynamic Programming, Greedy, Backtracking, and Branch Bound.
- To ensure low time complexity and scalability by using balanced data structures and optimized search techniques.

3 Methodology

This project follows a modular and algorithm-centric approach to recommend suitable crops after unexpected rainfall. The methodology is divided into data modeling, data organization, and decision optimization stages, each implemented using classical data structures and algorithms.

3.1 System Requirements and Functional Goals

The system is designed to assist farmers in selecting suitable crops after unexpected rainfall by using classical data structures and algorithms. The requirements focus on efficient data handling, fast decision-making, and optimal resource utilization under water constraints.

3.1.1 Functional Requirements

The proposed crop recommendation system must satisfy the following functional requirements in order to support decision-making after unexpected rainfall:

- The system shall store crop data efficiently for fast access.
- The system shall retrieve crop information with minimal delay.
- The system shall organize and rank crops based on expected profit.
- The system shall select an optimal subset of crops under water constraints.
- The system shall support multiple algorithmic strategies for comparison.
- The system shall display recommended crops and corresponding profit values.

3.1.2 Non-Functional Requirements

- The system shall ensure low time complexity for core operations.
- The system shall be modular and easy to maintain.
- The system shall provide explainable, deterministic decisions.
- The system shall scale efficiently with an increase in crop data.

- The system shall be implemented as a console-based Java application.

3.2 System Overview

The system accepts crop-related parameters such as water requirement and expected profit. Based on the available water after rainfall, the system computes optimal crop recommendations using multiple algorithmic strategies.

The complete workflow consists of the following stages:

- Crop data input.
- Efficient data storage and retrieval.
- Crop ranking and organization.
- Optimal crop selection under water constraints.
- Result validation using multiple algorithms.

3.3 Data Modeling

Each crop is modeled as an object with the following attributes:

- **Crop name:** A unique identifier used as the key for hash-based storage and retrieval.
- **Water requirement:** Represents the amount of water needed for successful crop growth and is treated as the weight in optimization algorithms.
- **Expected profit:** Represents the economic return of the crop and is treated as the value in optimization algorithms.
- **Growth Duration (in days):** Indicates the time required for the crop to mature. Although not directly used in the current optimization logic, this parameter allows future extensions such as time-based or multi-season planning.

This abstraction allows uniform processing across all data structures and algorithms.

3.4 Data Structures Employed

Efficient data organization is critical for fast decision-making after unexpected rainfall.
The following data structures are used to store, organize, and rank crop information.

3.4.1 Linked List

Description:
A singly linked list is used to store multiple crop records at a single hash index.

Role in System:
Serves as the underlying structure for separate chaining in the hash table to resolve collisions when multiple crop names hash to the same index. This ensures that hash table performance remains stable even when multiple crop entries map to the same index, preventing data loss and excessive collision overhead.

Justification:
Separate chaining avoids rehashing and allows dynamic growth of entries without memory reallocation.

Time Complexity:

- Insertion: $O(1)$
- Search (worst case): $O(n)$

Advantage: Efficient insertion and flexible memory usage.
Limitation: Linear search time within a chain in the worst case. [2]

3.4.2 Hash Table

Description:
A hash table maps crop names to their corresponding data records using a hash function.

Role in System:
Provides constant-time access to crop attributes such as water requirement and expected profit. All subsequent processing stages—sorting, ranking, and optimization—retrieve crop attributes through the hash table, making it the primary access layer of the system.

Justification:
Real-time decision-making after unexpected rainfall requires rapid retrieval of crop data.

Time Complexity:
Average case (insertion/search): $O(1)$

Advantage: Extremely fast lookup operations.
Limitation: Performance depends on hash function quality and load factor. [2]

3.4.3 Binary Search Tree (BST)

Description:
A binary search tree stores nodes such that left subtree values are less than the root and right subtree values are greater.

Role in System:
Organizes crops in sorted order based on expected profit. The BST provides an initial ordered representation of crop profitability that is later improved upon using balanced tree structures for performance consistency.

Justification:
Allows ordered traversal to analyze relative profitability of crops.

Time Complexity:

- Average case: $O(\log n)$
- Worst case (skewed tree): $O(n)$

Advantage: Maintains inherent sorted structure.
Limitation: Susceptible to imbalance, degrading performance.

Note: The Binary Search Tree is included as a conceptual baseline to illustrate performance degradation due to structural imbalance under certain insertion orders. This motivates the use of self-balancing trees such as AVL Trees, which guarantee logarithmic time complexity. [1]

3.4.4 AVL Tree

Description:
An AVL tree is a height-balanced binary search tree that maintains balance using rotations.

Role in System:
Stores crops in sorted order while guaranteeing balanced tree height. This structure serves as the system's main sorted repository, supporting frequent updates to crop data while maintaining efficient traversal and ranking operations.

Justification:
Ensures consistent logarithmic performance for insertion and search operations, even with dynamic updates.

Time Complexity:
Insertion/Search/Deletion: $O(\log n)$

Advantage: Predictable and stable performance.
Limitation: Additional computational overhead due to rotations. [1]

3.4.5 Max Heap

Description:
A max heap is a complete binary tree where the root node holds the maximum key value.

Role in System:
Ranks crops by expected profit and enables extraction of the highest-profit crop. Heap-based ranking allows the system to identify high-profit candidates early, which is particularly useful for comparison with greedy and optimal selection strategies.

Justification:
Efficient ranking is required for quick identification of top crop candidates.

Time Complexity:

- Insertion: $O(\log n)$
- Extract-max: $O(\log n)$

Advantage: Immediate access to the maximum element.
Limitation: Does not support full sorted traversal. [1]

3.5 Structural Classification of Data Structures

Data structures used in the system can be classified into computational support structures and analytical structures. Hash tables and linked lists operate exclusively as internal storage and access mechanisms. They facilitate constant-time data retrieval and collision handling but do not perform ordering, prioritization, or optimization operations. Consequently, they do not yield independent outputs and are evaluated implicitly through the correctness and efficiency of subsequent processing stages.

In contrast, data structures such as AVL Trees and Heaps directly implement ordering and prioritization logic. Since their correctness depends on structural properties—such as balance conditions and heap ordering—explicit outputs are presented to verify traversal order and maximum-element extraction behavior.

3.6 Algorithms Implemented

The crop recommendation problem under water constraints is solved using multiple algorithmic approaches. Each algorithm contributes either optimality, speed, or validation of results.

3.6.1 Dynamic Programming (0/1 Knapsack)

Formulation: The crop selection problem is modeled as a 0/1 knapsack problem:

- Item $\rightarrow$ Crop
- Weight $\rightarrow$ Water requirement
- Value $\rightarrow$ Expected profit
- Capacity $\rightarrow$ Available water

Role in System:
It computes the optimal subset of crops maximizing profit under a fixed water constraint. The solution produced by this algorithm is treated as the reference optimal output against which all heuristic and approximate methods are evaluated.

Justification:
Provides a guaranteed optimal solution for resource-constrained optimization.

Time Complexity:
$O(nW)$, where n is number of crops and W is water capacity.

Advantage: Ensures global optimality.
Limitation: Pseudo-polynomial time and higher memory usage. [1, 3]

3.6.2 Greedy Algorithm

Strategy:
Crops are sorted by profit-to-water ratio and selected sequentially until the water limit is reached.

Role in System:
It generates a fast approximate solution. Its output enables rapid decision-making and provides a baseline for assessing the trade-off between computational speed and solution optimality.

Justification:
Useful when immediate recommendations are required.

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

Advantage: Low computational cost.
Limitation: Does not guarantee optimality for 0/1 knapsack problems. [1]

3.6.3 Backtracking

Strategy:
Recursively explores all feasible subsets of crops.

Role in System:
It is used for correctness verification on small datasets. Due to its exhaustive nature, this approach is restricted to small input sizes and is used solely for validation of optimal solutions.

Justification:
Ensures exact evaluation of all combinations.

Time Complexity: $O(2^n)$

Advantage: Guarantees correctness.
Limitation: Computationally infeasible for large inputs. [4]

3.6.4 Branch and Bound

Strategy:
Enhances backtracking by pruning branches using upper profit bounds.

Role in System:
It reduces the search space while preserving optimality. By eliminating non-promising solution paths early, the algorithm achieves significant performance gains while still preserving exact optimality.

Justification:
Improves performance over naive backtracking.

Time Complexity:
Worst case: $O(2^n)$

Advantage: Significant reduction in explored states.
Limitation: Worst-case complexity remains exponential.

Note: Branch and Bound is implemented as an optimized extension of the Backtracking approach. While backtracking explores all possible crop combinations exhaustively, Branch and Bound reduces the search space by pruning branches that cannot lead to a better solution based on upper-bound estimates. [3]

3.7 Execution Flow

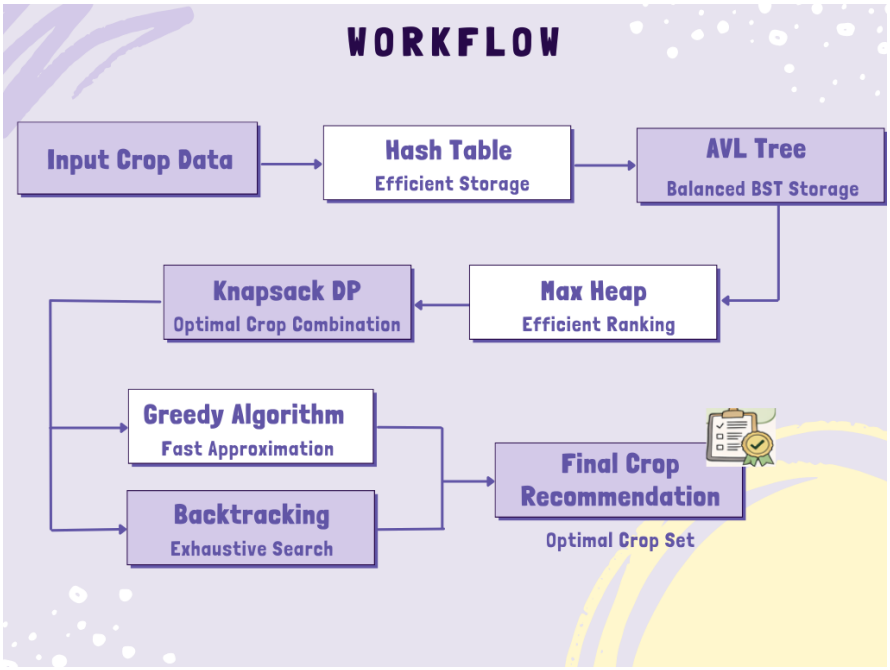

Figure 3.1: Workflow of the Crop Recommendation System Using Data Structures and Algorithms

4 Results & Discussion

This chapter analyzes the system output obtained after executing the crop recommendation model with a fixed water availability of 200 units. The discussion focuses on how each data structure and algorithm contributes to the decision-making process, and how their outputs collectively justify the final crop recommendation.

4.1 Crop Ordering Using AVL Tree

An AVL Tree was employed to maintain crops in sorted order based on expected profit while ensuring height balance. An inorder traversal of the tree produced the following ascending profit order:

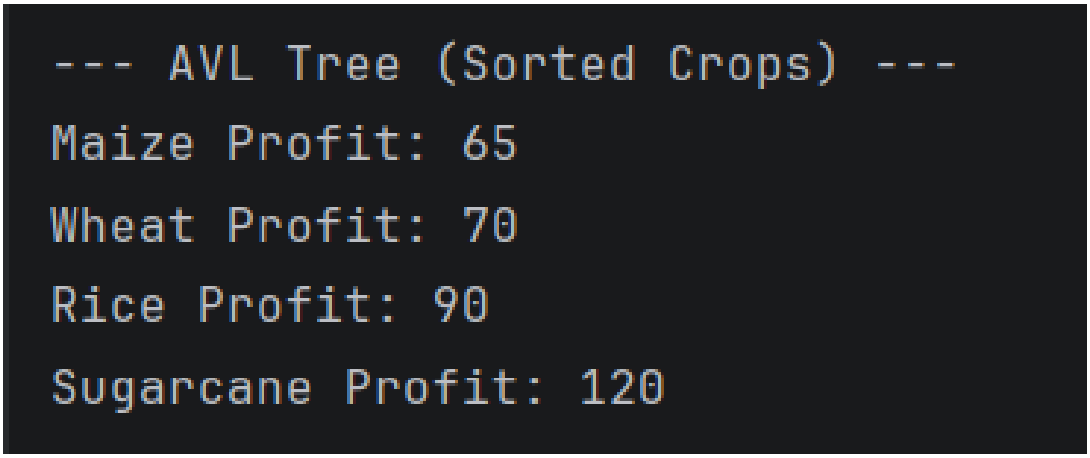

Figure 4.1: AVL Tree Inorder Traversal Showing Crops Sorted by Profit

This output confirms that the AVL Tree maintains a correctly ordered structure with guaranteed logarithmic time complexity for insertion and traversal. The balanced nature of the AVL Tree prevents performance degradation that may occur in an unbalanced binary search tree. This ordered representation is used for profitability analysis prior to optimization.

4.2 Identification of Maximum-Profit Crop Using Heap Structure

A Max Heap was used to efficiently identify the crop with the highest individual profit. Upon heap construction and extraction of the root node, the system identified Sugarcane as the highest-profit crop.

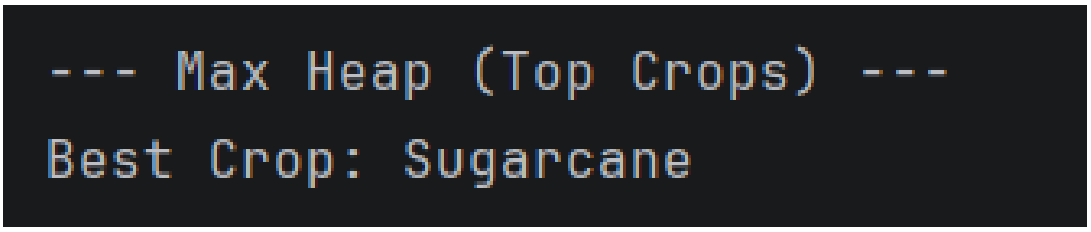

Figure 4.2: Max Heap Output Identifying the Highest-Profit Crop

This result validates the correctness of the heap property, where the maximum element is always accessible at the root. The heap-based approach enables efficient profit-based ranking without requiring full sorting, making it suitable for scenarios requiring quick prioritization.

4.3 Optimal Crop Combination Using Dynamic Programming

The core optimization problem was formulated as a 0/1 Knapsack problem, with water requirement as the weight and expected profit as the value. Dynamic Programming was applied to compute the maximum achievable profit under the water constraint.
The algorithm computed a maximum profit value of 185, representing the globally optimal selection of crops within the available water limit.

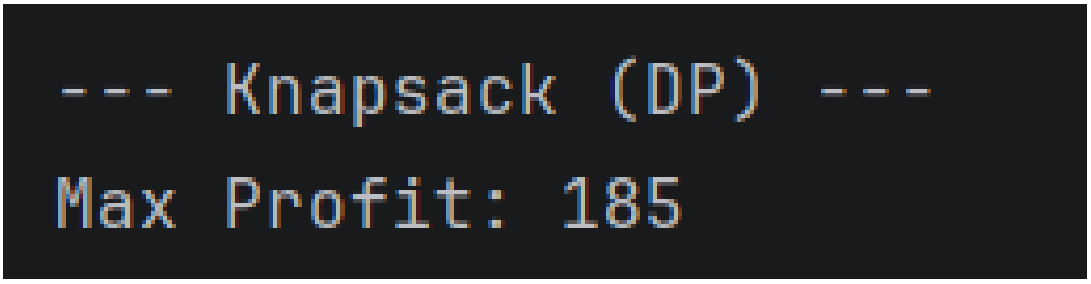

Figure 4.3: Dynamic Programming Output Showing Optimal Profit

Dynamic Programming guarantees optimality by systematically evaluating all feasible subproblems and building the solution bottom-up.
This output is treated as the reference solution for evaluating the effectiveness of other approaches.

4.4 Approximate Selection Using Greedy Strategy

A Greedy algorithm was implemented to generate a fast, approximate solution by selecting crops based on the highest profit-to-water ratio. The greedy approach selected Maize as the preferred crop.

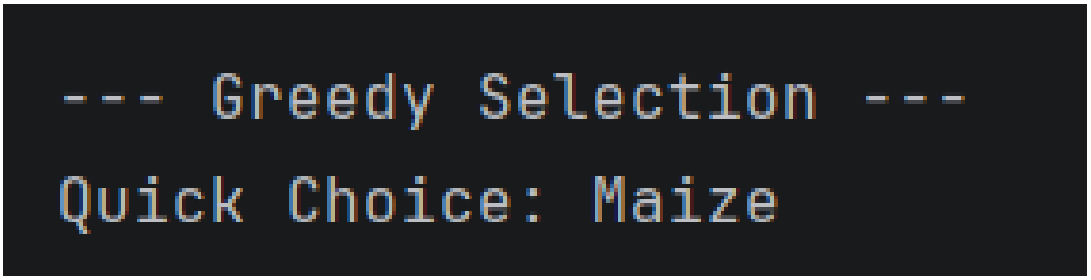

Figure 4.4: Greedy Algorithm Output Showing Locally Optimal Crop Selection

While this approach provides rapid decision-making, it fails to account for the combinatorial nature of the 0/1 Knapsack problem. The result demonstrates that greedy selection leads to a locally optimal decision, which does not necessarily correspond to the globally optimal solution.

4.5 Solution Verification Using Backtracking

To verify the correctness of the Dynamic Programming result, a Backtracking approach was used to exhaustively evaluate all possible crop subsets. The backtracking simulation computed a maximum profit of 185, which exactly matches the Dynamic Programming result.

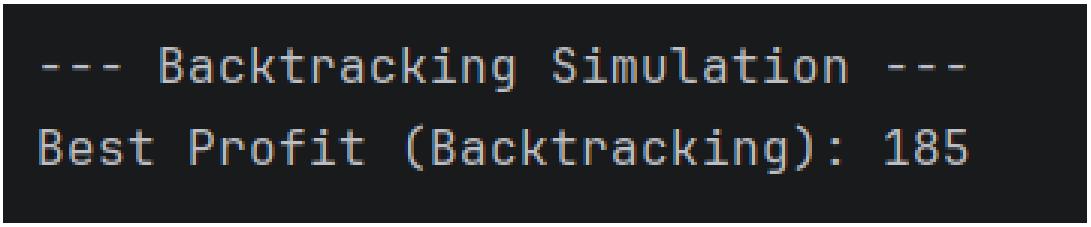

Figure 4.5: Backtracking Output Validating the Optimal Profit

This agreement confirms the correctness and optimality of the DP-based solution. Due to its exponential time complexity, backtracking is not suitable for large datasets and is used strictly for validation purposes.

4.6 Analysis of Design Choices

Technique	Purpose in System	Observed Outcome	Inference
AVL Tree	Maintains crops in sorted order by profit with balance	Crops sorted as: Maize, Wheat, Rice, Sugarcane	Ensures ordered traversal with guaranteed $O(\log n)$ performance
Max Heap	Identifies highest-profit individual crop	Sugarcane identified as top crop	Efficient for ranking but ignores resource constraints
Greedy Algorithm	Fast approximate crop selection using profit-to-water ratio	Selected Maize	Produces locally optimal but globally suboptimal solution
Dynamic Programming (0/1 Knapsack)	Computes optimal crop combination under water constraint	Maximum profit = 185	Guarantees global optimality under constraints
Backtracking	Exhaustive validation of all combinations	Maximum profit = 185	Confirms correctness of DP result; computationally expensive

Table 4.1: Comparative Analysis of Techniques Based on Observed Results

The comparative analysis highlights that different algorithmic techniques serve distinct roles within the system. While data structures such as AVL Trees and Max Heaps facilitate efficient ordering and prioritization, they do not account for resource constraints. Similarly, the greedy algorithm provides rapid decision-making but fails to capture the combinatorial nature of the crop selection problem. In contrast, Dynamic Programming consistently produces the globally optimal solution by exhaustively evaluating feasible subproblems under the water constraint, with its correctness independently validated using backtracking.

4.6.1 Data Structure Selection

The crop recommendation problem involves multiple distinct operations—fast key-based access, ordered traversal, priority-based ranking, and constrained optimization. Since no single data structure provides optimal performance for all these operations, the system employs a combination of specialized data structures, each selected to optimize a specific computational requirement.

Data Structure	**Role in System**	**Reason for Selection**
Hash Table	Fast access to crop data	Provides average-case $O(1)$ lookup and insertion
Linked List	Collision handling in hash table	Enables separate chaining without rehashing
Binary Search Tree (BST)	Baseline sorted representation	Illustrates ordered traversal and motivates need for self-balancing trees
AVL Tree	Sorted crop representation	Guarantees balanced height and $O(\log n)$ traversal
Max Heap	Profit-based prioritization	Efficient extraction of highest-profit crop

Table 4.2: Justification of Selected Data Structures

4.6.2 Algorithm Selection

The crop recommendation problem requires evaluating trade-offs between solution optimality, computational complexity, and execution time under strict resource constraints. Since no single algorithm simultaneously provides fast execution, guaranteed optimality, and scalability, the system employs multiple algorithmic strategies, each addressing a specific aspect of the decision-making process.

Algorithm	**Role in System**	**Justification**
Dynamic Programming (0/1 Knapsack)	Final crop selection under water constraint	Guarantees global optimality for constrained optimization problems
Backtracking	Validation of optimal solution	Confirms correctness of Dynamic Programming result through exhaustive search

Table 4.3: Justification of Selected Algorithms

The greedy algorithm is used only for comparative analysis and illustrates the limitation of locally optimal strategies for the 0/1 knapsack problem. Backtracking is employed to exhaustively validate the optimal solution obtained using Dynamic Programming. Branch and Bound

is applied as a pruning strategy within the backtracking process to reduce the search space, without affecting the optimal result for the given dataset.

System Execution Verification

The final console message, *"System Execution Complete"*, indicates that all system modules—including data structure initialization, algorithm execution, and result reporting—were executed successfully without runtime errors.

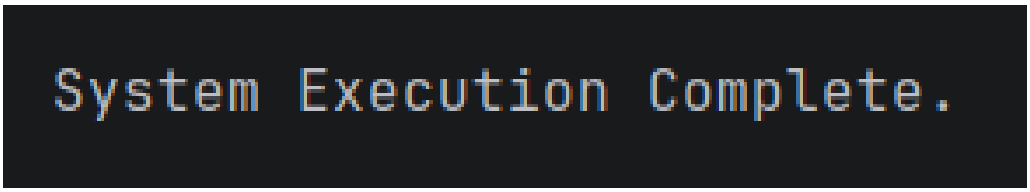

Figure 4.6: Console Output Confirming Successful System Execution

Note: This output serves solely as an execution verification indicator and does not contribute to the decision-making or optimization results discussed above.

5 Conclusion

This project demonstrates the application of classical data structures and algorithms to a real-world crop recommendation problem arising from unexpected rainfall and limited water availability. By formulating crop selection as a constrained optimization problem, the system enables structured, deterministic, and explainable decision-making without relying on data-intensive machine learning models.

Efficient data handling is achieved using a combination of hash tables, linked lists, AVL trees, and heaps, each chosen to optimize specific operations such as fast lookup, ordered traversal, and profit-based prioritization. This modular use of data structures ensures scalability and consistent performance across different stages of the system.

For decision-making, Dynamic Programming is employed as the primary optimization technique by modeling the problem as a 0/1 Knapsack problem, thereby guaranteeing global optimality under the given water constraint. The correctness of the Dynamic Programming solution is independently validated using Backtracking. A comparative evaluation with the greedy approach highlights the limitations of locally optimal strategies in solving constrained optimization problems.

Overall, the results confirm that the selected combination of data structures and algorithms achieves correctness, efficiency, and interpretability within the defined problem scope. The project aligns closely with the objectives of the ADSAA syllabus and serves as a practical demonstration of how foundational algorithmic techniques can be effectively applied to real-world, constraint-driven decision-making problems.

5.1 Limitations & Future Scopes

5.1.1 Limitations

- The system assumes static crop parameters such as water requirement and expected profit, which may vary in real-world conditions.
- Real-time environmental factors such as rainfall intensity and soil moisture are not dynamically updated.

- The solution is console-based and lacks a graphical user interface, which may limit usability for non-technical users.
- The dynamic programming approach may become computationally expensive for very large water capacities.

5.1.2 Future Scopes

- Integration with real-time weather APIs to dynamically update water availability.
- Development of a GUI-based interface for easier farmer interaction.
- Incorporation of soil nutrient and fertility parameters into the decision model.
- Extension of the system to support multi-season and long-term crop planning.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[2] E. Balagurusamy, *Data Structures Using Java*, 2nd ed. New Delhi, India: McGraw-Hill Education, 2014.

[3] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, 1990.

[4] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Pearson, 2006.

[5] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

[6] P. K. Mishra, D. R. Choudhary, and S. K. Sharma, "Decision support systems in agriculture: A review," *International Journal of Computer Applications*, vol. 97, no. 15, pp. 1–6, July 2014.

A Source Code Implementation

The following appendix contains the complete Java source code for the proposed system. Execution begins in `Main.java`, which invokes the recommendation engine to coordinate all data structures and algorithms.

A.1 Crop.java

```
public class Crop {
    String name;
    int waterRequired;
    int expectedProfit;
    int growthDays;

    Crop(String name, int waterRequired, int expectedProfit, int
        growthDays) {
        this.name = name;
        this.waterRequired = waterRequired;
        this.expectedProfit = expectedProfit;
        this.growthDays = growthDays;
    }
}
```

Listing A.1: Crop.java

A.2 LinkedList.java

```
public class LinkedList {
    class Node {
        Crop data;
        Node next;

        Node(Crop data) {
```

```
            this.data = data;
            this.next = null;
        }
    }

    Node head;

    void insert(Crop crop) {
        Node newNode = new Node(crop);
        newNode.next = head;
        head = newNode;
    }

    Node getHead() {
        return head;
    }
}
```

Listing A.2: LinkedList.java

A.3 HashTable.java

```
public class HashTable {
    int SIZE = 10;
    LinkedList[] table;

    HashTable() {
        table = new LinkedList[SIZE];
        for (int i = 0; i < SIZE; i++) {
            table[i] = new LinkedList();
        }
    }

    int hash(String key) {
        return Math.abs(key.hashCode()) % SIZE;
    }

    void insert(Crop crop) {
        int index = hash(crop.name);
        table[index].insert(crop);
    }
```

```
20
21      void display() {
22          for (int i = 0; i < SIZE; i++) {
23              LinkedList.Node temp = table[i].getHead();
24              while (temp != null) {
25                  System.out.println(temp.data.name);
26                  temp = temp.next;
27              }
28          }
29      }
30  }
```

Listing A.3: HashTable.java

A.4 BST.java

```
1   public class BST {
2       class Node {
3           Crop crop;
4           Node left, right;
5
6           Node(Crop crop) {
7               this.crop = crop;
8           }
9       }
10
11      Node root;
12
13      void insert(Crop crop) {
14          root = insertRec(root, crop);
15      }
16
17      Node insertRec(Node root, Crop crop) {
18          if (root == null)
19              return new Node(crop);
20
21          if (crop.expectedProfit < root.crop.expectedProfit)
22              root.left = insertRec(root.left, crop);
23          else
24              root.right = insertRec(root.right, crop);
25
```

```
26          return root;
27      }
28
29      void inorder(Node root) {
30          if (root != null) {
31              inorder(root.left);
32              System.out.println(root.crop.name + " Profit: " + root.
                    crop.expectedProfit);
33              inorder(root.right);
34          }
35      }
36  }
```

Listing A.4:]BST.java]

A.5 AVLTree.java

```
1   public class AVLTree {
2
3       class Node {
4           Crop crop;
5           Node left, right;
6           int height;
7
8           Node(Crop crop) {
9               this.crop = crop;
10              height = 1;
11          }
12      }
13
14      Node root;
15
16      int height(Node n) {
17          return n == null ? 0 : n.height;
18      }
19
20      int balance(Node n) {
21          return n == null ? 0 : height(n.left) - height(n.right);
22      }
23
24      Node rightRotate(Node y) {
```

```
25          Node x = y.left;
26          Node t = x.right;
27
28          x.right = y;
29          y.left = t;
30
31          y.height = Math.max(height(y.left), height(y.right)) + 1;
32          x.height = Math.max(height(x.left), height(x.right)) + 1;
33
34          return x;
35      }
36
37      Node leftRotate(Node x) {
38          Node y = x.right;
39          Node t = y.left;
40
41          y.left = x;
42          x.right = t;
43
44          x.height = Math.max(height(x.left), height(x.right)) + 1;
45          y.height = Math.max(height(y.left), height(y.right)) + 1;
46
47          return y;
48      }
49
50      Node insert(Node node, Crop crop) {
51          if (node == null)
52              return new Node(crop);
53
54          if (crop.expectedProfit < node.crop.expectedProfit)
55              node.left = insert(node.left, crop);
56          else
57              node.right = insert(node.right, crop);
58
59          node.height = 1 + Math.max(height(node.left), height(node.
               right));
60          int balance = balance(node);
61
62          // LL
63          if (balance > 1 && crop.expectedProfit < node.left.crop.
               expectedProfit)
```

```
64              return rightRotate(node);
65
66          // RR
67          if (balance < -1 && crop.expectedProfit > node.right.crop.
               expectedProfit)
68              return leftRotate(node);
69
70          // LR
71          if (balance > 1 && crop.expectedProfit > node.left.crop.
               expectedProfit) {
72              node.left = leftRotate(node.left);
73              return rightRotate(node);
74          }
75
76          // RL
77          if (balance < -1 && crop.expectedProfit < node.right.crop.
               expectedProfit) {
78              node.right = rightRotate(node.right);
79              return leftRotate(node);
80          }
81
82          return node;
83      }
84
85      void inorder(Node node) {
86          if (node != null) {
87              inorder(node.left);
88              System.out.println(node.crop.name + " Profit: " + node.
                   crop.expectedProfit);
89              inorder(node.right);
90          }
91      }
92  }
```

Listing A.5: AVLTree.java

A.6 MaxHeap.java

```
1  public class MaxHeap {
2
3      Crop[] heap = new Crop[20];
```

```
    int size = 0;

    void insert(Crop crop) {
        heap[size] = crop;
        int i = size;
        size++;

        while (i > 0 && heap[(i - 1) / 2].expectedProfit < heap[i].
           expectedProfit) {
            Crop temp = heap[i];
            heap[i] = heap[(i - 1) / 2];
            heap[(i - 1) / 2] = temp;
            i = (i - 1) / 2;
        }
    }

    Crop extractMax() {
        if (size == 0) return null;

        Crop max = heap[0];
        heap[0] = heap[size - 1];
        size--;

        heapify(0);
        return max;
    }

    void heapify(int i) {
        int largest = i;
        int l = 2 * i + 1;
        int r = 2 * i + 2;

        if (l < size && heap[l].expectedProfit > heap[largest].
           expectedProfit)
            largest = l;

        if (r < size && heap[r].expectedProfit > heap[largest].
           expectedProfit)
            largest = r;

        if (largest != i) {
```

```
            Crop temp = heap[i];
            heap[i] = heap[largest];
            heap[largest] = temp;
            heapify(largest);
        }
    }
}
```

Listing A.6: MaxHeap.java

A.7 GreedySelector.java

```
public class GreedySelector {

    static Crop selectBest(Crop[] crops) {
        Crop best = crops[0];
        for (Crop c : crops) {
            if ((double) c.expectedProfit / c.waterRequired >
                    (double) best.expectedProfit / best.
                       waterRequired) {
                best = c;
            }
        }
        return best;
    }
}
```

Listing A.7: GreedySelector.java

A.8 KnapsackDP.java

```
public class KnapsackDP {

    static int knapsack(Crop[] crops, int waterLimit) {
        int n = crops.length;
        int[][] dp = new int[n + 1][waterLimit + 1];

        for (int i = 1; i <= n; i++) {
            for (int w = 1; w <= waterLimit; w++) {
                if (crops[i - 1].waterRequired <= w) {
```

```
                    dp[i][w] = Math.max(
                            crops[i - 1].expectedProfit + dp[i - 1][
                               w - crops[i - 1].waterRequired],
                            dp[i - 1][w]
                    );
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
        return dp[n][waterLimit];
    }
}
```

Listing A.8: KnapsackDP.java

A.9 BacktrackingSimulator.java

```
public class BacktrackingSimulator {

    static int bestProfit = 0;

    static void simulate(Crop[] crops, int index, int water, int
       profit) {
        if (water < 0) return;
        if (index == crops.length) {
            bestProfit = Math.max(bestProfit, profit);
            return;
        }

        simulate(crops, index + 1, water, profit);
        simulate(crops, index + 1,
                water - crops[index].waterRequired,
                profit + crops[index].expectedProfit);
    }
}
```

Listing A.9: BacktrackingSimulator.java

A.10 RecommendationEngine.java

```
public class RecommendationEngine {

    Crop[] crops = {
            new Crop("Rice", 120, 90, 120),
            new Crop("Wheat", 60, 70, 110),
            new Crop("Maize", 50, 65, 100),
            new Crop("Sugarcane", 150, 120, 300)
    };

    void runAll(int waterLimit) {

        System.out.println("\n--- AVL Tree (Sorted Crops) ---");
        AVLTree avl = new AVLTree();
        for (Crop c : crops)
            avl.root = avl.insert(avl.root, c);
        avl.inorder(avl.root);

        System.out.println("\n--- Max Heap (Top Crops) ---");
        MaxHeap heap = new MaxHeap();
        for (Crop c : crops)
            heap.insert(c);
        System.out.println("Best Crop: " + heap.extractMax().name);

        System.out.println("\n--- Knapsack (DP) ---");
        System.out.println("Max Profit: " + KnapsackDP.knapsack(
           crops, waterLimit));

        System.out.println("\n--- Greedy Selection ---");
        System.out.println("Quick Choice: " + GreedySelector.
           selectBest(crops).name);

        System.out.println("\n--- Backtracking Simulation ---");
        BacktrackingSimulator.simulate(crops, 0, waterLimit, 0);
        System.out.println("Best Profit (Backtracking): " +
           BacktrackingSimulator.bestProfit);
    }
}
```

Listing A.10: RecommendationEngine.java

A.11 Main.java

```
public class Main {
    public static void main(String[] args) {

        RecommendationEngine engine = new RecommendationEngine();
        engine.runAll(200); // water after rainfall

        System.out.println("\nSystem Execution Complete.");
    }
}
```

Listing A.11: Main.java