THE AMERICAN UNIVERSITY IN CAIRO

CSCE 220302 – Analysis and Design of Algorithms Lab

Fall 2024

Coupon Selection for Maximum Savings

Algorithmic Solutions and Analysis

Author:

Saif Abd Elfattah (ID: 900225535)

Instructor:

Dr. May Shalaby

2/12/2024

Contents

Chapter 1

Introduction

In today's world, shoppers frequently encounter a variety of coupons, each offering discounts under specific terms and conditions. These coupons are often accompanied by constraints such as expiry dates, minimum spending requirements, and limits on the number of coupons that can be used simultaneously. Deciding which coupons to use for maximizing savings while adhering to these constraints presents a significant challenge, especially for cost-conscious consumers.

The problem of optimal coupon selection is not only relevant to individual shoppers but also to e-commerce platforms and retail stores that aim to optimize their coupon distribution strategies. Retailers often face scenarios where offering the right set of coupons could enhance customer satisfaction and boost sales while minimizing losses. For consumers, the ability to strategically select coupons can lead to substantial financial savings, especially in economies where inflation and rising costs are prevalent.

This problem is both significant and complex. From a real-world perspective, it has applications in personal budgeting, e-commerce, retail analytics, and marketing. Its complexity arises from the need to balance multiple constraints such as:

- Selecting a subset of coupons that maximizes savings.
- Ensuring the selected coupons meet minimum spending requirements.
- Accounting for limitations on the number of coupons that can be used at once.
- Managing expiry dates to avoid losing valuable discounts.

The problem is worth solving because it combines practical utility with algorithmic complexity. By approaching this issue algorithmically, it is possible to develop solutions that are not only efficient but also scalable to handle large datasets, such as a shopper's digital coupon collection or a retailer's coupon database.

This report investigates three algorithmic approaches—Greedy, Backtracking, and Dynamic Programming—for solving the coupon selection problem. Each method is evaluated for its effectiveness, efficiency, and suitability in real-world applications, offering a comprehensive perspective on addressing this significant challenge.

Chapter 2

Problem Description

The coupon selection problem involves identifying the optimal subset of coupons from a given list to maximize savings while satisfying multiple constraints. To abstract this real-world challenge and translate it into an algorithmic problem, the key components can be defined as follows:

2.1 Key Components

- **Coupons:** Each coupon is represented as an object with the following attributes:
  - `ID:` A unique identifier for the coupon.
  - `Discount Value:` The monetary value deducted from the total cost when the coupon is applied.
  - `Expiry Date:` The date beyond which the coupon is no longer valid.
  - `Cost Requirement:` The minimum spending required to use the coupon.
- **Constraints:** The problem incorporates the following constraints:
  - `Least Cost:` After applying the selected coupons, the total remaining cost must not fall below a predefined minimum.
  - `Maximum Coupons:` The user can apply at most a limited number of coupons.
  - `Validity:` Coupons must not have expired, and their cost requirements must be met.
- **Objective:** The goal is to maximize the total savings obtained by applying coupons while adhering to the given constraints.

2.2 Algorithmic Representation

The problem can be formulated as follows:

- **Input:**
  - A list of coupons, each with attributes `ID`, `Discount Value`, `Expiry Date`, and `Cost Requirement`.

  - The user's initial total cost (`userCost`).
  - The minimum allowable cost after discounts (`leastCost`).
  - The maximum number of coupons that can be applied (`maxCoupons`).
  - The current date to determine coupon validity (`todayDate`).

- **Output:**
  - A subset of valid coupons that maximizes savings.
  - The final cost after applying the selected coupons.

- **Constraints:**
  - The total number of selected coupons must not exceed `maxCoupons`.
  - The total remaining cost after applying discounts must not fall below `leastCost`.
  - Coupons with expiry dates earlier than `todayDate` are invalid.

Chapter 3

Code Explanation

3.1 Coupon Selection Program

The program aims to help shoppers maximize savings by selecting the most beneficial coupons from a list while considering several constraints.

3.1.1 Program Workflow

The program follows these main steps:

1. **Input:** Load coupons from a file, each containing:
   - `ID:` Unique identifier for each coupon.
   - `Discount Value:` Monetary value deducted from the total cost.
   - `Expiry Date:` The validity period of the coupon.
   - `Cost Requirement:` Minimum spending required to use the coupon.
2. **Preprocessing:**
   - Remove expired coupons based on the current date provided by the user.
   - Prompt the user for inputs, including:
     - `User Cost:` The total cost the user intends to spend.
     - `Least Cost:` The minimum amount the user must pay after discounts.
     - `Customer Score:` The maximum number of coupons the user can select.
3. **Algorithm Selection:**
   - The user chooses an approach based on preferences:
     - Prioritize retaining coupons nearing expiration (Greedy approach).
     - Maximize savings regardless of expiration (Backtracking or Dynamic Programming).
4. **Output:**
   - Display selected coupons, remaining coupons, and the final cost after discounts.

3.1.2 Shop Conditions

The program operates under the following constraints:

- Coupons must satisfy the cost requirement (`Cost Requirement` $\leq$ `User Cost`).
- The final cost after applying selected coupons must not fall below the `Least Cost`.
- The number of selected coupons must not exceed the user's `Customer Score`.

3.2 Display Functions

The following functions are responsible for managing the display and processing of coupon information.

3.2.1 `removeExpiredCoupons`

This function filters out coupons that have expired based on the current date provided by the user.

```cpp
void removeExpiredCoupons(vector<Coupon>& coupons, int today) {
    vector<Coupon> validCoupons;
    for (const auto& coupon : coupons) {
        if (coupon.expiryDate >= today) {
            validCoupons.push_back(coupon);
        }
    }
    coupons = validCoupons;  // Update the coupons list with valid ones
    cout << "Coupons that expired before today have been removed.\n";
}
```

Listing 3.1: removeExpiredCoupons Function

Explanation:

- Iterates through the list of coupons.
- Retains only those coupons whose expiry date is greater than or equal to the current date (`today`).
- Updates the list of coupons to include only valid ones.
- Outputs a message confirming the removal of expired coupons.

3.2.2 `displayCoupons`

This function displays the selected and remaining coupons after applying the chosen selection algorithm.

```cpp
void displayCoupons(const vector<Coupon>& coupons, const vector<
   Coupon>& selectedCoupons, int userCost) {
    double finalCost = userCost;
    for (const auto& coupon : selectedCoupons) {
        finalCost -= coupon.discountValue;
    }

    cout << "\nSelected Coupons:\n";
    for (const auto& coupon : selectedCoupons) {
        cout << "ID: " << coupon.id
            << ", Discount: " << coupon.discountValue
            << ", Expiry Date: " << coupon.expiryDate
            << ", Cost Requirement: " << coupon.costRequirement
               << endl;
    }
    cout << "Final Cost After Discounts: " << finalCost << endl;

    vector<Coupon> remainingCoupons;
    for (const auto& coupon : coupons) {
        bool used = false;
        for (const auto& selected : selectedCoupons) {
            if (coupon.id == selected.id) {
                used = true;
                break;
            }
        }
        if (!used) {
            remainingCoupons.push_back(coupon);
        }
    }

    cout << "\nRemaining Coupons (might expire soon):\n";
    for (const auto& coupon : remainingCoupons) {
        cout << "ID: " << coupon.id
            << ", Discount: " << coupon.discountValue
            << ", Expiry Date: " << coupon.expiryDate
            << ", Cost Requirement: " << coupon.costRequirement
               << endl;
    }
}
```

Listing 3.2: displayCoupons Function

Explanation:

- Computes the final cost after applying the discounts from the selected coupons.
- Outputs the details of the selected coupons, including:
  - `ID`, `Discount Value`, `Expiry Date`, and `Cost Requirement`.
- Identifies and displays the remaining coupons that were not selected.

- Provides insights into potential future savings or missed opportunities.

3.2.3 manageCoupons

This function handles user interactions for adding new coupons, selecting algorithms, and managing preferences.

```cpp
void manageCoupons(vector<Coupon>& coupons, const string&
   filename) {
    cout << "\nDo you want to add new coupons or run the program?
        (Add/Run): ";
    string choice;
    cin >> choice;

    if (choice == "Add") {
        int numCoupons;
        cout << "How many coupons would you like to add? ";
        cin >> numCoupons;

        for (int i = 0; i < numCoupons; ++i) {
            int id, expiry, costReq;
            double discount;
            cout << "Enter coupon ID: ";
            cin >> id;
            cout << "Enter discount value: ";
            cin >> discount;
            cout << "Enter expiry date (YYYYMMDD): ";
            cin >> expiry;
            cout << "Enter cost requirement: ";
            cin >> costReq;

            coupons.push_back(Coupon(id, discount, expiry,
               costReq));
        }
        saveCoupons(coupons, filename);
    }
}
```

Listing 3.3: manageCoupons Function

Explanation:

- Prompts the user to add new coupons or run the program.
- Accepts user inputs for coupon details and updates the list of coupons.
- Saves the updated list of coupons to a file for persistent storage.

Chapter 4

Algorithmic Approaches

This chapter explores three algorithmic approaches to solve the coupon selection problem. Each approach includes the rationale, detailed function explanation, code implementation, trade-offs, and an example trace using the provided test case.

4.1 Greedy Approach

The greedy approach prioritizes using coupons that are about to expire, ensuring they are not wasted, while attempting to maximize savings within the given constraints. It uses the `greedySelectCoupons` function and relies on `calculateAverageExpiry` to track the average expiry of selected coupons.

4.1.1 Code Implementation

```cpp
vector<Coupon> greedySelectCoupons(vector<Coupon>& coupons, int
   userCost, int maxCoupons, int leastCost) {
    vector<Coupon> validCoupons;

    // Filter coupons where costRequirement <= userCost
    for (const auto& coupon : coupons) {
        if (coupon.costRequirement <= userCost) {
            validCoupons.push_back(coupon);
        }
    }

    // Sort coupons by expiryDate (ascending) and discountValue (
       descending)
    sort(validCoupons.begin(), validCoupons.end(), [](const
       Coupon& a, const Coupon& b) {
        if (a.expiryDate != b.expiryDate) {
            return a.expiryDate < b.expiryDate;
        }
        return a.discountValue > b.discountValue;
    });

    vector<Coupon> selectedCoupons;
    double remainingCost = userCost;

    // Greedily select coupons
    for (const auto& coupon : validCoupons) {
        if (selectedCoupons.size() < maxCoupons && remainingCost
            - coupon.discountValue >= leastCost) {
            selectedCoupons.push_back(coupon);
            remainingCost -= coupon.discountValue;
        }
    }

    return selectedCoupons;
}
```

Listing 4.1: greedySelectCoupons Function

4.1.2 Explanation of Functions

- `calculateAverageExpiry`: This function computes the average expiry date of a given list of coupons. It:
  - Iterates through the subset of selected coupons.
  - Returns the average expiry or a very high value if the subset is empty.

  This metric ensures that the algorithm can track expiry trends and make informed selections.

- `greedySelectCoupons`: This function:
  1. Filters out coupons that do not meet the cost requirement (`costRequirement <= userCost`).
  2. Sorts coupons by:
     - Expiry date (earliest first).
     - Discount value (highest first for ties).
  3. Iteratively selects coupons that:
     - Do not exceed the user's maximum allowable coupons (`maxCoupons`).
     - Maintain the final cost above the specified least cost (`leastCost`).

4.1.3 Rationale

The greedy approach is built on the principle of making the best local decision at each step, prioritizing coupons about to expire to ensure minimal wastage. While it does not guarantee a globally optimal solution, it is efficient and aligns with user preferences to retain near-expiration coupons.

4.1.4 Example

```
Input File (coupons.txt):
1,50.000000,20241130,50
```

```
2,20.000000,20241125,100
3,30.000000,20241125,150
4,10.000000,20241123,160
5,60.000000,20241130,100
6,40.000000,20241129,100

Selected Coupons:
ID: 4, Discount: 10, Expiry Date: 20241123, Cost Requirement: 160
ID: 3, Discount: 30, Expiry Date: 20241125, Cost Requirement: 150
ID: 2, Discount: 20, Expiry Date: 20241125, Cost Requirement: 100

Final Cost After Discounts: 140
```

The greedy algorithm prioritizes using the coupons that are about to expire over maximizing savings. In this example:

- It selects **Coupon ID 4** (discount: 10, expiry: 20241123, cost requirement: 160) as it has the earliest expiry date.
- Then, it selects **Coupon ID 3** (discount: 30, expiry: 20241125, cost requirement: 150).
- Finally, it selects **Coupon ID 2** (discount: 20, expiry: 20241125, cost requirement: 100).

This results in a final cost of **140** after discounts. The algorithm prioritizes the early expiry date of coupons (**20241123 and 20241125**) over achieving maximum savings by ignoring higher discount coupons like **Coupon ID 5** (discount: 60, expiry: 20241130).

4.1.5 Trade-offs

- **Advantages:**
  - Fast and computationally efficient, especially for large datasets.
  - Simple to implement and easy to understand.
- **Disadvantages:**
  - Does not always find the globally optimal solution due to its myopic nature.
  - May prioritize expiry over potential savings, which can result in suboptimal total discounts.

4.2 Backtracking Approach

The backtracking approach explores all possible subsets of coupons to identify the combination that maximizes savings while respecting constraints like the least cost and maximum allowable coupons. It ensures an optimal solution by evaluating every potential selection.

4.2.1 Code Implementation

```cpp
double calculateAverageExpiry(const vector<Coupon>& coupons) {
    if (coupons.empty()) return numeric_limits<double>::max();
    double sum = 0;
    for (const auto& coupon : coupons) {
        sum += coupon.expiryDate;
    }
    return sum / coupons.size();
}
```

Listing 4.2: calculateAverageExpiry Function

```cpp
void findOptimalSubset(const vector<Coupon>& coupons, int
   maxCoupons, int leastCost, double remainingCost,
    vector<Coupon>& currentSubset, vector<Coupon>& bestSubset,
    double& closestCost, double& bestAvgExpiry) {
    double currentAvgExpiry = calculateAverageExpiry(
       currentSubset);

    // Check if the current subset is a better solution
    if (remainingCost >= leastCost &&
        (remainingCost < closestCost ||
         (remainingCost == closestCost && currentAvgExpiry <
            bestAvgExpiry))) {
        closestCost = remainingCost;
        bestAvgExpiry = currentAvgExpiry;
        bestSubset = currentSubset;
    }

    // Stop recursion if the subset size reaches the maximum
       allowable coupons
    if (currentSubset.size() == maxCoupons) {
        return;
    }

    // Iterate over the remaining coupons
    for (size_t i = 0; i < coupons.size(); ++i) {
        const Coupon& coupon = coupons[i];
        double newRemainingCost = remainingCost - coupon.
           discountValue;

        // Only proceed if the new remaining cost is above the
           least cost
        if (newRemainingCost >= leastCost) {
            currentSubset.push_back(coupon);
            vector<Coupon> remainingCoupons(coupons.begin() + i +
                 1, coupons.end());
            findOptimalSubset(remainingCoupons, maxCoupons,
               leastCost, newRemainingCost, currentSubset,
               bestSubset, closestCost, bestAvgExpiry);
            currentSubset.pop_back(); // Backtrack to explore
               other combinations
        }
    }
}

vector<Coupon> backtrackingSelectCoupons(vector<Coupon>& coupons,
    int userCost, int maxCoupons, int leastCost) {
    vector<Coupon> validCoupons;

    // Filter valid coupons based on the cost requirement
    for (const auto& coupon : coupons) {
        if (coupon.costRequirement <= userCost) {
            validCoupons.push_back(coupon);
        }
    }

    // Sort coupons by expiryDate (ascending) and discountValue (
       descending)
    sort(validCoupons.begin(), validCoupons.end(), [](const
       Coupon& a, const Coupon& b) {
        if (a.expiryDate != b.expiryDate) {
            return a.expiryDate < b.expiryDate;
        }
        return a.discountValue > b.discountValue;
    });

    vector<Coupon> currentSubset;
    vector<Coupon> bestSubset;
    double closestCost = numeric_limits<double>::max();
    double bestAvgExpiry = numeric_limits<double>::max();

    // Find the optimal subset using backtracking
    findOptimalSubset(validCoupons, maxCoupons, leastCost,
       userCost, currentSubset, bestSubset, closestCost,
       bestAvgExpiry);

    return bestSubset;
}
```

Listing 4.3: findOptimalSubset and backtrackingSelectCoupons Functions

4.2.2 Explanation of Functions

- `calculateAverageExpiry`:
  - This is a function that computes the average expiry date of a subset of coupons. This helps track how the selected subset performs in terms of expiry, ensuring the algorithm doesn't waste coupons nearing expiration.
  - For each subset:

- `findOptimalSubset`:
  - This recursive function evaluates all possible subsets of valid coupons.
  - For each subset:
    - It calculates the remaining cost and checks if the subset satisfies the constraints.
    - If the subset offers a better savings (or the same savings with a better average expiry), it updates the best subset.
  - It backtracks by removing the last added coupon after exploring deeper branches of recursion.
- `backtrackingSelectCoupons`:
  - Filters valid coupons based on the user's cost requirement.
  - Sorts coupons by expiry date and discount value to guide the search.
  - Calls `findOptimalSubset` to identify the subset that maximizes savings.

4.2.3 Rationale

Backtracking provides an exploration of all possible subsets, ensuring the optimal solution is found. This approach is particularly useful when constraints and trade-offs between savings and expiry dates need to be carefully balanced.

4.2.4 Example

```
Input File (coupons.txt):
1,50.000000,20241130,50
2,20.000000,20241125,100
3,30.000000,20241125,150
4,10.000000,20241123,160
5,60.000000,20241130,100
6,40.000000,20241129,100

Selected Coupons:
ID: 4, Discount: 10, Expiry Date: 20241123, Cost Requirement: 160
ID: 3, Discount: 30, Expiry Date: 20241125, Cost Requirement: 150
ID: 5, Discount: 60, Expiry Date: 20241130, Cost Requirement: 100

Final Cost After Discounts: 100
```

The backtracking algorithm maximizes savings by selecting the subset that provides the highest discount while considering the average expiry date. In this example:

- It selects **Coupon ID 4** (discount: 10, expiry: 20241123), **Coupon ID 3** (discount: 30, expiry: 20241125), and **Coupon ID 5** (discount: 60, expiry: 20241130).
- These coupons provide a total discount of **100**, resulting in a final cost of **100**.

The algorithm could have also taken **Coupon ID 1** (discount: 50, expiry: 20241130), **Coupon ID 2** (discount: 20, expiry: 20241125), and **Coupon ID 3** (discount: 30, expiry: 20241125), which would give the same total savings of **100**. However, the chosen subset (**Coupons 3, 4, 5**) has a better average expiry date, prioritizing coupons that expire sooner while maintaining the same level of savings.

4.2.5 Trade-offs

- **Advantages:**
  - Guarantees the optimal solution.
  - Handles complex trade-offs between savings and other constraints (e.g., expiry dates) more effectively than greedy algorithms.
- **Disadvantages:**
  - Computationally expensive due to its exhaustive search, especially for large datasets.
  - Slower than dynamic programming, which avoids redundant calculations.

4.2.6 Comparison to Other Approaches

- **Greedy Approach:** While faster, it may miss the globally optimal solution that backtracking guarantees.
- **Dynamic Programming:** Both ensure optimal solutions, but dynamic programming is more efficient for larger datasets due to memoization.

4.3 Dynamic Programming Approach

The dynamic programming (DP) approach focuses on efficiently exploring the possible subsets of coupons to maximize savings while adhering to constraints. Like the greedy approach, it utilizes `calculateAverageExpiry` to track subset expiry metrics.

4.3.1 Code Implementation

```cpp
vector<Coupon> dpSelectCoupons(const vector<Coupon>& coupons, int
    userCost, int maxCoupons, int leastCost) {
    vector<Coupon> validCoupons;
    for (const auto& coupon : coupons) {
        if (coupon.costRequirement <= userCost) {
            validCoupons.push_back(coupon);
        }
    }

    sort(validCoupons.begin(), validCoupons.end(), [](const
       Coupon& a, const Coupon& b) {
        if (a.expiryDate != b.expiryDate) {
            return a.expiryDate < b.expiryDate;
        }
        return a.discountValue > b.discountValue;
    });

    vector<vector<pair<double, tuple<double, vector<Coupon>>>>> dp(
        maxCoupons + 1, vector<pair<double, tuple<double, vector<Coupon>>>>(userCost + 1, { numeric_limits<double>::max(), {numeric_limits<double>::max(), {}} }));

    dp[0][userCost] = { userCost, {numeric_limits<double>::max(), {}} };

    for (const auto& coupon : validCoupons) {
        for (int usedCoupons = maxCoupons - 1; usedCoupons >= 0; --usedCoupons) {
            for (int currentCost = userCost; currentCost >= 0; --currentCost) {
                if (dp[usedCoupons][currentCost].first != numeric_limits<double>::max()) {
                    double newCost = dp[usedCoupons][currentCost].first - coupon.discountValue;
                    if (newCost >= leastCost) {
                        auto newSelection = get<1>(dp[usedCoupons][currentCost].second);
                        newSelection.push_back(coupon);

                        double newAverageExpiry = calculateAverageExpiry(newSelection);

                        if (newCost < dp[usedCoupons + 1][static_cast<int>(newCost)].first ||
                            (newCost == dp[usedCoupons + 1][static_cast<int>(newCost)].first &&
                             newAverageExpiry < get<0>(dp[usedCoupons + 1][static_cast<int>(newCost)].second))) {
                            dp[usedCoupons + 1][static_cast<int>(newCost)] = { newCost, { newAverageExpiry, newSelection} };
                        }
                    }
                }
            }
        }
    }

    pair<double, tuple<double, vector<Coupon>>> bestSolution = { numeric_limits<double>::max(), {numeric_limits<double>::max(), {}} };
    for (int usedCoupons = 1; usedCoupons <= maxCoupons; ++
        usedCoupons) {
        for (int currentCost = leastCost; currentCost <= userCost
            ; ++currentCost) {
            if (dp[usedCoupons][currentCost].first < bestSolution
                .first ||
                (dp[usedCoupons][currentCost].first ==
                    bestSolution.first &&
                     get<0>(dp[usedCoupons][currentCost].second) <
                         get<0>(bestSolution.second))) {
                bestSolution = dp[usedCoupons][currentCost];
            }
        }
    }

    return get<1>(bestSolution.second);
}
```

Listing 4.4: dpSelectCoupons Function

4.3.2 Explanation of Functions

- `calculateAverageExpiry`: As described earlier, it calculates the average expiry date for selected subsets.
- `dpSelectCoupons`:
  1. Filters valid coupons based on the cost requirement.
  2. Uses a DP table to store intermediate results for subsets of coupons.
  3. For each coupon, iteratively updates the DP table to track:
     - Remaining cost after applying the coupon.
     - Average expiry date of the current subset.
  4. Identifies the subset with the maximum savings and the most favorable average expiry date.

4.3.3 Rationale

The dynamic programming approach improves upon the backtracking method by avoiding redundant calculations through memoization. By building a DP table, the algorithm efficiently explores all possible subsets without repeatedly evaluating the same combinations.

4.3.4 Example

```
Input File (coupons.txt):
1,50.000000,20241130,50
2,20.000000,20241125,100
3,30.000000,20241125,150
```

```
4,10.000000,20241123,160
5,60.000000,20241130,100
6,40.000000,20241129,100

Selected Coupons:
ID: 4, Discount: 10, Expiry Date: 20241123, Cost Requirement: 160
ID: 3, Discount: 30, Expiry Date: 20241125, Cost Requirement: 150
ID: 5, Discount: 60, Expiry Date: 20241130, Cost Requirement: 100

Final Cost After Discounts: 100
```

The dynamic programming algorithm also maximizes savings by selecting the subset with the highest discount while optimizing for the average expiry date. In this example:

- It selects **Coupon ID 4** (discount: 10, expiry: 20241123), **Coupon ID 3** (discount: 30, expiry: 20241125), and **Coupon ID 5** (discount: 60, expiry: 20241130).
- These coupons provide a total discount of **100**, resulting in a final cost of **100**.

Like the backtracking approach, the algorithm could have chosen **Coupon ID 1** (discount: 50, expiry: 20241130), **Coupon ID 2** (discount: 20, expiry: 20241125), and **Coupon ID 3** (discount: 30, expiry: 20241125), which also give total savings of **100**. However, it selects the subset (**Coupons 3, 4, 5**) because it offers a better average expiry date.

4.3.5 Trade-offs

- **Advantages:**
  - Ensures an optimal solution by considering all valid subsets of coupons.
  - Reduces computational complexity compared to exhaustive search (backtracking).
  - Incorporates multiple criteria (e.g., savings and expiry) in decision-making.
- **Disadvantages:**
  - Requires significant memory to maintain the DP table, especially for large datasets.
  - Slightly more complex to implement and debug than greedy algorithms.

4.3.6 Comparison to Other Approaches

- The DP approach balances the precision of backtracking with improved efficiency.
- Unlike the greedy approach, DP ensures a globally optimal solution.
- It handles trade-offs between maximizing savings and retaining coupons with near expiration better than the greedy approach.

Chapter 5

Complexity Analysis

This chapter discusses the computational complexity of each algorithm, analyzing how they perform in terms of time and space requirements.

5.1 Greedy Approach

The greedy approach selects coupons based on their expiry date and discount value, prioritizing early expiry dates and higher discounts. The complexity of the `greedySelectCoupons` function is derived as follows:

5.1.1 Steps and Complexity Derivation

1. **Filtering Valid Coupons:**
   - The function iterates through all n coupons to filter those that satisfy the condition `costRequirement <= userCost`.
   - **Time Complexity:** $O(n)$.
2. **Sorting Coupons:**
   - The valid coupons are sorted by expiry date (ascending) and by discount value (descending) when expiry dates are the same.
   - Let m represent the number of valid coupons after filtering. Sorting takes $O(m \log m)$.
   - Since $m \leq n$, this can be expressed as $O(n \log n)$.
3. **Selecting Coupons:**
   - The algorithm iteratively selects coupons while maintaining the conditions:
     - The total number of selected coupons does not exceed `maxCoupons`.
     - The remaining cost after applying discounts does not fall below `leastCost`.
   - This process involves iterating through at most m valid coupons.
   - **Time Complexity:** $O(m)$, which can be expressed as $O(n)$ since $m \leq n$.

5.1.2 Overall Time Complexity

Combining all steps:

$$O(n) + O(n \log n) + O(n) = O(n \log n)$$

5.1.3 Space Complexity

- The algorithm stores the list of valid coupons, which has size $m \leq n$, and the selected subset, which has a maximum size of `maxCoupons`.
- **Space Complexity:** $O(n)$.

5.2 Backtracking Approach

The backtracking approach explores all possible subsets of coupons to find the optimal solution, maximizing savings while considering constraints such as least cost and maximum allowable coupons. The complexity is derived as follows:

5.2.1 Steps and Complexity Derivation

1. **Filtering Valid Coupons:**
   - The function `backtrackingSelectCoupons` filters valid coupons based on the condition `costRequirement <= userCost`.
   - This requires iterating over all n coupons.
   - **Time Complexity:** $O(n)$.
2. **Sorting Coupons:**
   - The valid coupons are sorted by expiry date (ascending) and by discount value (descending) when expiry dates are the same.
   - If there are m valid coupons after filtering, sorting takes $O(m \log m)$.
   - Since $m \leq n$, the sorting complexity can be expressed as $O(n \log n)$.
3. **Exploring Subsets via Recursion:**
   - The recursive function `findOptimalSubset` explores all possible subsets of valid coupons. For a set of m valid coupons, there are 2^m possible subsets.
   - For each subset, the algorithm evaluates the remaining cost and the average expiry date. Computing the average expiry date (`calculateAverageExpiry`) requires iterating through the current subset, which has a maximum size of `maxCoupons`.
   - Thus, the complexity of evaluating one subset is $O(\texttt{maxCoupons})$.
   - The total time complexity for exploring all subsets is:

$$O(2^m \cdot \texttt{maxCoupons})$$

5.2.2 Overall Time Complexity

Combining all steps:

$$O(n) + O(n \log n) + O(2^m \cdot \texttt{maxCoupons})$$

Since $O(2^m \cdot \texttt{maxCoupons})$ dominates, the overall time complexity is:

$$O(2^m \cdot \texttt{maxCoupons})$$

where $m \leq n$.

5.2.3 Space Complexity

- The maximum depth of the recursion stack is equal to the number of valid coupons m or the maximum allowable coupons (`maxCoupons`), whichever is smaller.
- The function also stores the current subset and best subset, each of size `maxCoupons`.
- **Space Complexity:**
  $$O(\texttt{maxCoupons})$$

5.3 Dynamic Programming Approach

The dynamic programming approach avoids exploring all subsets by using a table to store intermediate results, thereby optimizing the computation process. The time and space complexity for the `dpSelectCoupons` function are derived as follows:

5.3.1 Steps and Complexity Derivation

1. **Filtering Valid Coupons:**
   - The function iterates through all n coupons to filter those that satisfy the condition `costRequirement <= userCost`.
   - **Time Complexity:** $O(n)$.
2. **Sorting Coupons:**
   - The filtered coupons are sorted by expiry date (ascending) and by discount value (descending) when expiry dates are the same.
   - If there are m valid coupons after filtering, sorting takes $O(m \log m)$.
   - Since $m \leq n$, the sorting complexity can be expressed as $O(n \log n)$.
3. **Filling the DP Table:**
   - The DP table has dimensions `(maxCoupons + 1)` rows and `(userCost + 1)` columns.
   - For each valid coupon, the algorithm iterates over all rows (`maxCoupons`) and columns (`userCost`).

- For each cell, it updates the table values by evaluating the new remaining cost and selecting the optimal subset.
- **Time Complexity:**

$$O(m \cdot (\texttt{maxCoupons} + 1) \cdot (\texttt{userCost} + 1)) = O(n \cdot \texttt{maxCoupons} \cdot \texttt{userCost})$$

4. **Extracting the Best Subset:**
   - After the DP table is filled, extracting the best subset involves iterating over all rows and columns to find the optimal solution.
   - **Time Complexity:** $O(\texttt{maxCoupons} + \texttt{userCost})$.

5.3.2 Overall Time Complexity

Combining all steps:

$$O(n) + O(n \log n) + O(n \cdot \texttt{maxCoupons} \cdot \texttt{userCost}) + O(\texttt{maxCoupons} + \texttt{userCost})$$

Since $O(n \cdot \texttt{maxCoupons} \cdot \texttt{userCost})$ dominates, the overall time complexity is:

$$O(n \cdot \texttt{maxCoupons} \cdot \texttt{userCost})$$

5.3.3 Space Complexity

- The DP table requires storage for `(maxCoupons + 1)` rows and `(userCost + 1)` columns.
- **Space Complexity:**

$$O(\texttt{maxCoupons} \cdot \texttt{userCost})$$

Chapter 6

Comparison & Conclusion

6.1 Comparison of Approaches

Table 6.1 summarizes the key features, time complexity, space complexity, strengths, and weaknesses of the three approaches.

Approach	Time Complexity	Space Complexity	Strengths	Weaknesses
Greedy	$O(n \log n)$	$O(n)$	Simple and efficient for large datasets; prioritizes near-expiry coupons.	May not provide the optimal solution as it prioritizes expiry dates over savings.
Backtracking	$O(2^m \cdot \texttt{maxCoupons})$	$O(\texttt{maxCoupons})$	Finds the optimal solution by exploring all subsets; balances savings and expiry dates.	Exponential time complexity; impractical for large datasets.
Dynamic Programming	$O(n \cdot \texttt{maxCoupons} \cdot \texttt{userCost})$	$O(\texttt{maxCoupons} \cdot \texttt{userCost})$	Efficiently finds the optimal solution by avoiding redundant calculations; balances savings and expiry dates.	Requires more memory than greedy or backtracking approaches; implementation is complex.

Table 6.1: Comparison of Greedy, Backtracking, and Dynamic Programming Approaches

6.2 Conclusion

The choice of approach depends on the problem constraints, dataset size, and user preferences:

6.2.1 Greedy Approach

- **Best suited for:** Large datasets where speed is critical and near-expiry coupons are a priority.
- **Strengths:** Efficient and straightforward to implement.
- **Weaknesses:** Does not guarantee an optimal solution, especially when maximum savings are a priority.

6.2.2 Backtracking Approach

- **Best suited for:** Small datasets where achieving the optimal solution is essential.
- **Strengths:** Guarantees the optimal solution by evaluating all possible subsets.

- **Weaknesses:** Computationally expensive and impractical for large datasets due to exponential time complexity.

6.2.3 Dynamic Programming Approach

- **Best suited for:** Medium to large datasets where optimal solutions are required without the computational cost of backtracking.
- **Strengths:** Balances efficiency and optimality; avoids redundant calculations through memoization.
- **Weaknesses:** Requires significant memory and is more complex to implement compared to greedy or backtracking approaches.

6.3 Final Recommendations

- Use the **Greedy Approach** for applications prioritizing speed and simplicity, where near-expiry coupons are critical.
- Opt for **Dynamic Programming** when both optimality and efficiency are required for large datasets.
- Employ **Backtracking** only for small datasets or when exploring alternative solutions for understanding trade-offs.