Project Report

Shopping With Coupons

Zhao Menglei
Zhou Haowen
Pu Yuancan

December 1, 2025

Contents

1 Introduction **1**
1.1 Problem Description . . . 1
1.2 Objective . . . 1
1.3 Challenges . . . 1

2 System Design and Algorithms **1**
2.1 Data Structure Selection . . . 1
2.2 Algorithm Design . . . 1
2.2.1 Proof of Correctness (Greedy Strategy) . . . 1
2.2.2 Algorithm Implementation (Implicit Graph Search) . . . 2
2.3 Pseudocode . . . 2

3 Testing and Evaluation **2**
3.1 Test Sample . . . 2
3.2 Test Results . . . 2
3.3 Analysis . . . 2

4 Complexity Analysis and Discussion **2**
4.1 Time Complexity . . . 2
4.2 Space Complexity . . . 3
4.3 Discussion . . . 3

5 Declaration **4**

A Source Code **5**

1 Introduction

1.1 Problem Description

In this project, we need to solve a shopping optimization problem. The scenario is as follows:

- We have N items with different prices.
- We have N types of coupons with different values.
- We have the amount of money D.

The rule for buying an item is simple: we can pair any item with any coupon. The cost of one transaction is calculated as:

$$\text{Cost} = \text{Price} - \text{Coupon Value}$$

Notice: the highest value of coupons is less than the lowest price of items. So we do not need to consider the negative cost.

We can reuse the same type of coupon or buy the same item multiple times, but a specific pair (Item i, Coupon j) can only be used once. Our goal is to find a strategy to buy the **maximum number of items** without exceeding the budget D.

1.2 Objective

We want to select K pairs of $(item, coupon)$ such that the total cost is less than or equal to D, and K is maximized. This is a typical combinatorial optimization problem.

1.3 Challenges

The main difficulty of this problem is the data size.

- The number of items N can be up to 10^5.
- The total budget D can be up to 10^6.

If we try to list all possible combinations of items and coupons, there would be $N \times N = 10^{10}$ pairs. A simple brute-force solution (calculating all pairs and sorting them) requires too much memory and computation time. Therefore, we need to design an efficient greedy algorithm using a **Priority Queue** to find the solution within the time limit.

2 System Design and Algorithms

2.1 Data Structure Selection

Guideline: Discuss how data is stored.

2.2 Algorithm Design

2.2.1 Proof of Correctness (Greedy Strategy)

The core strategy of our algorithm is simple: **"Always choose the cheapest available combination of item and coupon."**

But why does this guarantee the maximum number of items? We can prove this using a logical method called the *Exchange Argument*.

The Proposition To maximize the number of purchased items (k) within a fixed budget D, we must select the k combinations with the **smallest costs**.

The Proof Suppose there exists an "Optimal Solution" that is different from our "Greedy Solution".

1. **Assumption:** The Greedy Solution picks a cheap combination A (Cost C_A), but the Optimal Solution decides **not** to pick A. Instead, to reach the same count, the Optimal Solution picks a more expensive combination B (Cost C_B).
2. **Inequality:** Since the Greedy algorithm always picks the minimum cost available, it must be true that:
$$C_A < C_B$$
3. **Exchange:** If we modify the Optimal Solution by swapping B for A:
$$\text{New Total Cost} = \text{Old Total Cost} - C_B + C_A$$
Since $C_A < C_B$, the New Total Cost is **smaller** than the Old Total Cost.
4. **Conclusion:** By choosing the cheaper item A instead of B, we save money $(C_B - C_A)$. With this extra money, we might be able to buy even more items later.

Therefore, replacing any expensive choice with a cheaper choice never hurts the result; it only saves budget. This proves that selecting the cheapest options first is always the best strategy to maximize the total count.

2.2.2 Algorithm Implementation (Implicit Graph Search)

2.3 Pseudocode

3 Testing and Evaluation

3.1 Test Sample

Guideline: List your OS and Compiler version.

3.2 Test Results

Guideline: Present a table of test cases.

3.3 Analysis

Guideline: Briefly analyze the results. Mention that the program passed the sample and handled large inputs within the time limit.

4 Complexity Analysis and Discussion

4.1 Time Complexity

To analyze the time complexity exactly, we define the following variables:

- N: The number of items and coupons ($N \leq 10^5$).
- D: The initial budget ($D \leq 10^6$).
- K: The actual number of items purchased. Since the minimum cost is at least 1, $K \leq D$.
- M: The number of nodes in the priority queue at any given time. Due to the implicit graph search strategy, the heap maintains a "wavefront" of candidates, where $M \approx O(N)$.

The total execution time consists of two parts: **Preprocessing** and the **Greedy Loop**.

1. Preprocessing (Sorting)

The algorithm begins by sorting the `prices` array in ascending order and the `coupons` array in descending order using QuickSort .
$$T_{sort} = 2 \times O(N \log N) = O(N \log N)$$

2. The Main Greedy Loop

The loop runs K times (once for each item purchased). Inside the loop, we perform operations on a 4-ary Heap:

- **DeleteMin (Shift Down):** In a 4-ary heap, the height is $\log_4 M$. Each step requires comparing 4 children. The complexity is $O(\log_4 M)$.
- **Pruning & Insertion (Shift Up):** We calculate the next potential state. If the cost exceeds the remaining budget D, we prune the node. Otherwise, we insert it into the heap. The insertion takes $O(\log_4 M)$.

Since the maximum heap size M is bounded by $O(N)$, and the loop runs K times, the total time for the loop is:

$$T_{loop} \approx K \times (O(\log_4 N) + O(\log_4 N)) = O(K \log N)$$

Conclusion

Adding both parts, the total time complexity is:

$$T_{total} = O(N \log N + K \log N)$$

Given the constraints $N = 10^5$ and $K \leq 10^6$, the algorithm performs well within the 100-200ms range, far below the 1-second time limit.

4.2 Space Complexity

The space complexity is determined by the storage required for input data and the dynamic data structures.

- **Static Storage:** We store the `prices` and `coupons` arrays, taking $O(N)$ space.
- **Dynamic Storage (Heap):** We use a structure-of-arrays approach (parallel arrays for costs and indices). Although there are N^2 possible combinations, our **implicit graph search** strategy only stores the candidate frontier. The maximum number of nodes in the heap is linear with respect to N.
- **Auxiliary Space:** The recursion stack for 'qsort' takes $O(\log N)$.

Thus, the total space complexity is:

$$S_{total} = O(N)$$

For $N = 10^5$, the memory usage is approximately 2.4 MB, which is significantly lower than the typical memory limit (64 MB or 128 MB).

4.3 Discussion

Why not $O(N^2)$?

A brute-force approach would generate all $N \times N$ combinations, sort them, and select the cheapest ones. With $N = 10^5$, this results in 10^{10} entries, which would immediately cause a Memory Limit Exceeded and Time Limit Exceeded. Our approach avoids this by dynamically generating states.

The Power of Pruning

The condition `if (next_cost <= D)` is a critical optimization. As the budget D decreases, fewer items satisfy this condition. This prevents the heap from growing unnecessarily, ensuring that we only store reachable states.

4-ary Heap vs. Binary Heap

We implemented a **4-ary Heap** instead of a standard Binary Heap. A 4-ary heap reduces the tree height by half ($\log_4 N = \frac{1}{2} \log_2 N$), reducing the number of levels to traverse during 'ShiftUp' and 'ShiftDown'. Additionally, 4-ary heaps exhibit better **cache locality** because child nodes are stored contiguously in memory, reducing CPU cache misses during large-scale operations.

5 Declaration

A Source Code