

PROJECT REPORT

Shopping With Coupons

ZHAO MINGLEI

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
2.1.1	Data Structure Details	1
2.1.2	Rationale for Data Structure Selection	2
2.2	Algorithm Design (Greedy Strategy)	2
2.2.1	Proof of Correctness (Greedy Strategy)	2
2.2.2	Algorithm Implementation	3
2.3	Pseudocode	3
2.4	Main Program Sketch	4
3	Testing and Evaluation	5
3.1	Test Sample	5
3.2	Test Results	5
3.3	Analysis	5
4	Complexity Analysis and Discussion	5
4.1	Time Complexity	5
4.2	Space Complexity	6
4.3	Discussion	6
5	Declaration	7
A	Source Code	8
A.1	Header File: solvers.h	8
A.2	Main Program: main.c	8
A.3	Basic Implementation: solve.c (v1)	9
A.4	Optimized Implementation: solve_v2.c (v2)	10

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

2.1.1 Data Structure Details

We implement two versions of data structures to store the candidates for the greedy selection. We also define a unified result structure.

- **Common Output Structure:**

```
Struct Result {
    Integer count;           // Total items purchased
    LongLong left;          // Remaining budget
}
```

- **Version 1 (Basic): Array of Structures (AoS)** Used in the standard implementation. Each node encapsulates all information for a specific purchase option.

```

Struct Node {
    Integer item_idx; // Index in sorted Prices array
    Integer coupon_idx; // Index in sorted Coupons array
    LongLong cost; // Cached value of (Prices[i] - Coupons[j])
}
MinHeap<Node> heap; // Standard Binary Heap

```

- **Version 2 (Optimized): Structure of Arrays (SoA)** Used for the high-performance implementation. We decouple the attributes into separate arrays to improve CPU cache locality and utilize a **4-ary Heap** to reduce tree height.

```

Array heap_cost[]; // Stores costs (Key for sorting)
Array heap_p_idx[]; // Stores item indices
Array heap_c_idx[]; // Stores coupon indices
Integer size; // Current heap size

```

2.1.2 Rationale for Data Structure Selection

Why Min-Heap (Priority Queue)? The fundamental requirement of our greedy strategy is to repeatedly retrieve the candidate with the **global minimum cost** and insert new candidates.

- A linear scan would take $O(N)$ per purchase, leading to $O(N^2)$ overall, which causes Time Limit Exceeded (TLE).
- A Min-Heap allows extraction of the minimum and insertion of new elements in $O(\log N)$ time. This ensures the total time complexity remains bounded by $O(K \log N)$ (where K is the number of items bought), which fits comfortably within the time limit.

Optimization: From Binary Heap (v1) to 4-ary Heap (v2) While the standard Binary Heap is efficient, we optimized it to a **4-ary Heap (Quad Heap)** in the final version based on the following architectural considerations:

1. **Reduced Tree Height (Theoretical Improvement):** A 4-ary heap is shallower than a binary heap. The height of the tree changes from $\log_2 N$ to $\log_4 N$, which equals $\frac{1}{2} \log_2 N$. This reduces the number of levels traversed during **push** (sift-up) and **pop** (sift-down) operations by 50%.
2. **Cache Locality (System-Level Improvement):** In modern CPU architectures, memory is accessed in cache lines (typically 64 bytes).
 - In a Binary Heap, the children of node i are at $2i + 1$ and $2i + 2$.
 - In a 4-ary Heap, the children are at $4i + 1 \dots 4i + 4$.

The four children in a 4-ary heap are stored contiguously in memory. Accessing one child likely brings the others into the L1/L2 cache, significantly reducing **Cache Misses** compared to the binary layout. Although a 4-ary heap requires more comparisons per level (finding the minimum of 4 children), the reduction in memory latency and tree height results in a net performance gain.

2.2 Algorithm Design (Greedy Strategy)

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

1. Core Logic

The fundamental approach relies on sorting and a priority queue:

- (a) **Sorting:** Sort `Prices` in ascending order and `Coupons` in descending order. This ensures optimal pairings are easily discoverable.
- (b) **Greedy Choice:** We maintain a Min-Heap of potential "buyable" combinations. In each step, we extract the combination with the minimal cost.
- (c) **Consumption:** If the budget allows, we buy the item and decrement the budget. The loop terminates when the heap is empty or the cheapest item is unaffordable.

2. Optimizations

To handle large datasets ($N = 10^5$) efficiently, we introduce two key improvements in the second version:

- (a) **Frontier Expansion:** Instead of initializing the heap with all N items (which costs $O(N)$), we strictly utilize the monotonicity of the sorted arrays.
 - We start with only the global minimum: $(Price_0, Coupon_0)$.
 - When a node (i, j) is extracted, we insert its "horizontal" neighbor $(i, j + 1)$.
 - Only when $j = 0$ (it is the first time item i is used), we insert the "vertical" neighbor $(i + 1, 0)$.

This keeps the heap size small, proportional to the number of purchases made, not N .

- (b) **Cost Pruning:** Before pushing any new candidate into the heap, we verify if `candidate.cost ≤ current_budget`. Impossible candidates are discarded immediately, saving memory and heap operations.

2.3 Pseudocode

The following algorithm illustrates the optimized workflow, integrating the greedy strategy with frontier expansion and pruning.

Algorithm 1: Optimized Greedy Shopping Strategy

```

Input: Item count  $N$ , Budget  $D$ , sorted Arrays  $P$  (asc) and  $C$  (desc)
Output: Total items bought, Remaining budget

// Initialize with the single best candidate
1 Heap.insert( $P\_idx = 0, C\_idx = 0, Cost = P[0] - C[0]$ )
2  $Count \leftarrow 0$ 
3 while Heap.isEmpty() do
    // Extract the cheapest option
4    $Current \leftarrow \text{Heap.popMin}()$ 
5   if  $Current.Cost > D$  then
6     | break                                // Budget exceeded, stop immediately
7   end
8    $D \leftarrow D - Current.Cost$ 
9    $Count \leftarrow Count + 1$ 
10  // Expand 1: Try same item with next coupon
11   $NextC \leftarrow Current.C\_idx + 1$ 
12  if  $NextC < N$  then
13    |  $NewCost \leftarrow P[Current.P\_idx] - C[NextC]$ 
14    | // Optimization: Pruning Check
15    | if  $NewCost \leq D$  then
16      |   | Heap.insert( $Current.P\_idx, NextC, NewCost$ )
17    | end
18  end
19  // Expand 2: Try next item (only if current is fresh)
20  if  $Current.C\_idx == 0$  then
21    |  $NextP \leftarrow Current.P\_idx + 1$ 
22    | if  $NextP < N$  then
23      |   |  $NewCost \leftarrow P[NextP] - C[0]$ 
24      |   | if  $NewCost \leq D$  then
25        |     | Heap.insert( $NextP, 0, NewCost$ )
26      |   | end
27    | end
28  end
29 return  $Count, D$ 

```

2.4 Main Program Sketch

The main program serves as the driver layer for the application. It is designed to strictly separate Input/Output (I/O) operations from the core algorithmic logic. This modular design ensures that the solver functions (`solve_v1` and `solve_v2`) remain pure, testable, and reusable.

The execution flow consists of three distinct phases:

1. **Data Ingestion & Memory Management:** Since the input size N can reach 10^5 , allocating arrays on the stack may cause a stack overflow. Therefore, we verify the input validity of N and D , and then use dynamic memory allocation (`malloc`) for the `prices` and `coupons` arrays.
2. **Modular Invocation:** The sorted arrays and budget parameters are passed to the solver function. We invoke the optimized solver `solve_v2` by default. The solver returns a `Result` structure, keeping the main function agnostic to the internal complexity of the greedy algorithm.
3. **Output & Resource Cleanup:** The results are formatted according to the specification. Crucially, all dynamically allocated memory is released using `free()` before termination to prevent memory leaks, adhering to strict memory safety standards.

Algorithm 2: Main Driver Flow

Input: Standard Input Stream (stdin)
Output: Standard Output Stream (stdout)

```

// Phase 1: Input and Allocation
1 Read integers  $N$  and  $D$ 
2 if Input is valid then
3   Allocate array  $P$  of size  $N$                                 // Prices
4   Allocate array  $C$  of size  $N$                                 // Coupons
5   Read elements into  $P$  and  $C$ 
     // Phase 2: Execution
     // Call the optimized solver
6    $FinalResult \leftarrow solve\_v2(N, D, P, C)$ 
     // Phase 3: Output and Cleanup
7   Print  $FinalResult.count$  and  $FinalResult.left$ 
8   Free memory for  $P$ 
9   Free memory for  $C$ 
10 end
11 return 0

```

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.

5 Declaration

A Source Code

This appendix contains the complete C implementation. The code is modularized into a header file, a main driver, and two separate solver implementations (Basic vs. Optimized). Detailed comments are included to explain the logic and optimizations.

A.1 Header File: solvers.h

Defines the unified data structures and function prototypes used across modules.

Listing 1: solvers.h

```

1 #ifndef SOLVERS_H
2 #define SOLVERS_H
3
4 // Maximum number of items as per problem specification (10^5)
5 #define MAX_N 100005
6
7 // Unified structure to return the final answer
8 typedef struct {
9     int count;          // Total number of items purchased
10    long long left;    // Remaining budget in the pocket
11 } Result;
12
13 // Function prototypes
14 // solve_v1: Basic Greedy with Binary Heap
15 Result solve_v1(int N, long long D, int* prices, int* coupons);
16
17 // solve_v2: Optimized Greedy with 4-ary Heap, SoA, and Frontier Expansion
18 Result solve_v2(int N, long long D, int* prices, int* coupons);
19
20 #endif

```

A.2 Main Program: main.c

Handles I/O operations and memory management. It isolates the algorithmic logic from data ingestion.

Listing 2: main.c

```

1 #include <stdio.h>
2 #include "solvers.h"
3 #include <stdlib.h>
4
5 int main(){
6     int N;
7     long long D;
8
9     // Read N (items) and D (budget)
10    // Return 0 if input format is incorrect
11    if (scanf("%d%lld", &N, &D) != 2) return 0;
12
13    // Use Dynamic Memory Allocation (Heap) instead of Stack
14    // Reason: N can be up to 10^5, which might cause Stack Overflow if simple arrays are used
15
16    int *prices = (int *)malloc(N * sizeof(int));
17    int *coupons = (int *)malloc(N * sizeof(int));
18
19    // Read input arrays
20    for (int i = 0; i < N; i++) scanf("%d", &prices[i]);
21    for (int i = 0; i < N; i++) scanf("%d", &coupons[i]);
22
23    // Invoke the optimized solver (v2)
24    // This modular design allows switching to solve_v1 easily for testing
25    Result res = solve_v2(N, D, prices, coupons);
26
27    // Output results separated by space
28    printf("%d %lld\n", res.count, res.left);
29
30    // Clean up memory to prevent memory leaks
31    free(prices);

```

```

31     free(coupons);
32
33     return 0;
34 }
```

A.3 Basic Implementation: solve.c (v1)

Implements the standard Greedy strategy using an **Array of Structures (AoS)** and a standard **Binary Heap**.

Listing 3: solve.c

```

1 #include "solvers.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 // Data Structure: Array of Structures (AoS)
6 typedef struct {
7     int item_idx;    // Index in prices array
8     int coupon_idx; // Index in coupons array
9     long long cost; // Cache cost: prices[i] - coupons[j]
10 } Node;
11
12 Node heap[MAX_N];
13 int heap_size = 0;
14
15 // Helper functions for qsort
16 const int compareAsc(const void *a, const void *b){
17     return (*(int *)a - *(int *)b);
18 }
19
20 const int compareDesc(const void *a, const void *b){
21     return (*(int *)b - *(int *)a);
22 }
23
24 // Standard Binary Heap Insert (Sift Up)
25 const void insert(Node item){
26     if (heap_size >= MAX_N){
27         printf("heap is fullfilled\n");
28         return;
29     }
30     heap[heap_size] = item;
31     int index = heap_size;
32
33     // Sift Up Logic
34     while (index > 0){
35         int parent = (index - 1) / 2;
36         if (heap[index].cost < heap[parent].cost){
37             Node temp = heap[index];
38             heap[index] = heap[parent];
39             heap[parent] = temp;
40             index = parent;
41         } else {
42             break;
43         }
44     }
45     heap_size++;
46 }
47
48 // Standard Binary Heap Sift Down
49 const void siftDown(int index){
50     while (2*index + 1 < heap_size){
51         int child = 2*index + 1; // Left child
52
53         // Check if right child exists and is smaller
54         if (child + 1 < heap_size && heap[child].cost > heap[child + 1].cost)
55             child++;
56
57         // Swap if child is smaller than parent
58         if (heap[child].cost < heap[index].cost){
59             Node temp = heap[child];
60             heap[child] = heap[index];
61             heap[index] = temp;
62         }
63     }
64 }
```

```

61         heap[index] = temp;
62         index = child;
63     } else {
64         break;
65     }
66 }
67 }
68
69 Node deleteMin(){
70     if (heap_size == 0){
71         printf("empty heap!\n");
72         exit(1);
73     }
74     Node min = heap[0];
75     heap[0] = heap[heap_size-1]; // Move last element to root
76     heap_size--;
77     siftDown(0); // Restore heap property
78     return min;
79 }
80
81 Result solve_v1(int N, long long D, int* prices, int* coupons){
82     heap_size = 0;
83
84     // Step 1: Sort arrays
85     qsort(prices, N, sizeof(int), compareAsc);
86     qsort(coupons, N, sizeof(int), compareDesc);
87
88     // Step 2: Full Initialization (The Naive Approach)
89     // We calculate the best cost for EVERY item and push all N items into heap.
90     for(int i = 0; i < N; i++){
91         Node temp;
92         temp.item_idx = i;
93         temp.coupon_idx = 0; // Pair with the best coupon
94         temp.cost = (long long)(prices[i] - coupons[0]);
95         insert(temp);
96     }
97
98     int count = 0;
99
100    // Step 3: Greedy Loop
101    while (heap_size > 0){
102        Node current = deleteMin(); // Get cheapest option
103
104        // Check budget
105        if (D >= current.cost){
106            D -= current.cost;
107            count++;
108        } else {
109            break; // Cannot afford the cheapest, stop.
110        }
111
112        // Expansion: Only expand horizontally (same item, next coupon)
113        int next_coupon_idx = current.coupon_idx + 1;
114        if (next_coupon_idx < N){
115            Node next;
116            next.item_idx = current.item_idx;
117            next.coupon_idx = next_coupon_idx;
118            next.cost = (long long)(prices[next.item_idx] - coupons[next.coupon_idx]);
119            insert(next);
120        }
121    }
122
123    Result res = {count, D};
124    return res;
125}

```

A.4 Optimized Implementation: solve_v2.c (v2)

Features extensive optimizations: **Structure of Arrays (SoA)**, **4-ary Heap**, **Frontier Expansion**, and **Cost Pruning**.

Listing 4: solve_v2.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "solvers.h"
4
5 // OPTIMIZATION 1: Structure of Arrays (SoA)
6 // Improves CPU cache locality compared to 'struct Node'.
7 long long heap_cost[MAX_N];
8 int heap_p_idx[MAX_N];
9 int heap_c_idx[MAX_N];
10 int size = 0;
11
12 // Macro for swapping elements across three arrays
13 #define SWAP(i, j) { \
14     long long temp_cost = heap_cost[i]; heap_cost[i] = heap_cost[j]; heap_cost[j] = temp_cost \
15     ;\ \
16     int temp_p_idx = heap_p_idx[i]; heap_p_idx[i] = heap_p_idx[j]; heap_p_idx[j] = temp_p_idx \
17     ;\ \
18     int temp_c_idx = heap_c_idx[i]; heap_c_idx[i] = heap_c_idx[j]; heap_c_idx[j] = temp_c_idx \
19 }
20
21 const int compareAsc(const void *a, const void *b){ \
22     return (*(int *)a - *(int *)b); \
23 }
24
25 const int compareDesc(const void *a, const void *b){ \
26     return (*(int *)b - *(int *)a); \
27 }
28
29 // OPTIMIZATION 2: 4-ary Heap (Quad Heap)
30 // Reduces tree height by half ( $\log_4 N = 0.5 * \log_2 N$ ), reducing sift-up/down depth.
31 void insert(int p_idx, int c_idx, long long cost){
32     if (size > MAX_N) {
33         printf("heap is fullfilled\n");
34         return;
35     }
36
37     heap_cost[size] = cost;
38     heap_c_idx[size] = c_idx;
39     heap_p_idx[size] = p_idx;
40
41     int index = size;
42     // Sift Up for 4-ary heap
43     while (index > 0){
44         int parent = (index - 1) / 4; // Parent index calculation changes
45         if (heap_cost[index] < heap_cost[parent]){
46             SWAP(index, parent);
47             index = parent;
48         } else {
49             break;
50         }
51     }
52     size++;
53 }
54
55 const void shiftDown(int index){
56     // While at least the first child exists
57     while (4*index + 1 < size){
58         int child = 4*index + 1;
59         int min_child = child;
60
61         // Find the minimum among up to 4 children
62         // Loop unrolled for performance
63         if (child + 1 < size && heap_cost[child + 1] < heap_cost[min_child]) min_child = child + 1;
64         if (child + 2 < size && heap_cost[child + 2] < heap_cost[min_child]) min_child = child + 2;
65         if (child + 3 < size && heap_cost[child + 3] < heap_cost[min_child]) min_child = child + 3;
66
67         if (heap_cost[index] > heap_cost[min_child]) {
68             SWAP(index, min_child);
69         }
70     }
71 }
```

```

67         index = min_child;
68     } else {
69         break;
70     }
71 }
72 }
73
74 const void deleteMin(){
75     if (size == 0){
76         printf("empty heap!\n");
77         exit(1);
78     }
79     // Move last element to root
80     heap_cost[0] = heap_cost[size - 1];
81     heap_p_idx[0] = heap_p_idx[size - 1];
82     heap_c_idx[0] = heap_c_idx[size - 1];
83
84     size--;
85     shiftDown(0);
86 }
87
88 Result solve_v2(int N, long long D, int* prices, int* coupons){
89     size = 0;
90     qsort(prices, N, sizeof(int), compareAsc);
91     qsort(coupons, N, sizeof(int), compareDesc);
92
93     // OPTIMIZATION 3: Frontier Expansion (Initialization)
94     // Only push the global minimum (0,0) initially.
95     // Heap size starts at 1 instead of N.
96     insert(0, 0, (long long)(prices[0] - coupons[0]));
97
98     int count = 0;
99
100    while (size > 0){
101        // Retrieve min element
102        long long current_cost = heap_cost[0];
103        int current_p_idx = heap_p_idx[0];
104        int current_c_idx = heap_c_idx[0];
105
106        deleteMin();
107
108        // Pruning Check 1: Can we afford it?
109        if (D >= current_cost){
110            D -= current_cost;
111            count++;
112        } else {
113            break; // Budget exceeded
114        }
115
116        // Expansion Strategy:
117        // 1. Horizontal: Same item, next coupon (always try to add)
118        int next_c_idx = current_c_idx + 1;
119        if (next_c_idx < N){
120            long long next_cost = (long long)(prices[current_p_idx] - coupons[next_c_idx]);
121            // OPTIMIZATION 4: Cost Pruning
122            // Only insert if we can theoretically afford it
123            if (next_cost <= D){
124                insert(current_p_idx, next_c_idx, next_cost);
125            }
126        }
127
128        // 2. Vertical: Next item, best coupon
129        // Only done when we just used the BEST coupon for the current item.
130        // This ensures every combination is added exactly once.
131        if (current_c_idx == 0){
132            int next_p_idx = current_p_idx + 1;
133            if (next_p_idx < N){
134                long long next_cost = (long long)(prices[next_p_idx] - coupons[0]);
135                if (next_cost <= D){ // Pruning
136                    insert(next_p_idx, 0, next_cost);
137                }
138            }
139        }
}

```

```
140     }
141
142     Result res = {count, D};
143     return res;
144 }
```