Project Report

Shopping With Coupons

Zhao Menglei
Zhou Haowen
Pu Yuancan

December 1, 2025

Contents

1 Introduction

Guideline: This section should provide an overview of the problem.

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)

The core problem is to maximize purchases with a limited budget. We employ a greedy strategy with specific optimizations for efficiency.

2.2.1 Core Logic

The fundamental approach relies on sorting and a priority queue:

1. **Sorting:** Sort `Prices` in ascending order and `Coupons` in descending order. This ensures optimal pairings are easily discoverable.
2. **Greedy Choice:** We maintain a Min-Heap of potential "buyable" combinations. In each step, we extract the combination with the minimal cost.
3. **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.

2.2.2 Optimizations

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

1. **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.
2. **Cost Pruning:** Before pushing any new candidate into the heap, we verify if `candidate.cost` $\leq$ `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
Heap.insert(P_idx = 0, C_idx = 0, Cost = P[0] − C[0])
Count ← 0
while Heap.isNotEmpty() do
    // Extract the cheapest option
    Current ← Heap.popMin()
    if Current.Cost > D then
        break                                   // Budget exceeded, stop immediately
    end
    D ← D − Current.Cost
    Count ← Count + 1
    // Expand 1: Try same item with next coupon
    NextC ← Current.C_idx + 1
    if NextC < N then
        NewCost ← P[Current.P_idx] − C[NextC]
        // Optimization: Pruning Check
        if NewCost ≤ D then
            Heap.insert(Current.P_idx, NextC, NewCost)
        end
    end
    // Expand 2: Try next item (only if current is fresh)
    if Current.C_idx == 0 then
        NextP ← Current.P_idx + 1
        if NextP < N then
            NewCost ← P[NextP] − C[0]
            if NewCost ≤ D then
                Heap.insert(NextP, 0, NewCost)
            end
        end
    end
end
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
Read integers N and D
if Input is valid then
    Allocate array P of size N                    // Prices
    Allocate array C of size N                    // Coupons
    Read elements into P and C
    // Phase 2: Execution
    // Call the optimized solver
    FinalResult ← solve_v2(N, D, P, C)
    // Phase 3: Output and Cleanup
    Print FinalResult.count and FinalResult.left
    Free memory for P
    Free memory for C
end
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

Guideline: Analyze the mathematical complexity.

4.2 Space Complexity

Guideline: Analyze memory usage.

4.3 Discussion

Guideline: Discuss any trade-offs or why a simple $O(N^2)$ loop would fail.

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

```c
#ifndef SOLVERS_H
#define SOLVERS_H

// Maximum number of items as per problem specification (10^5)
#define MAX_N 100005

// Unified structure to return the final answer
typedef struct {
    int count;      // Total number of items purchased
    long long left; // Remaining budget in the pocket
} Result;

// Function prototypes
// solve_v1: Basic Greedy with Binary Heap
Result solve_v1(int N, long long D, int* prices, int* coupons);

// solve_v2: Optimized Greedy with 4-ary Heap, SoA, and Frontier Expansion
Result solve_v2(int N, long long D, int* prices, int* coupons);

#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

```c
#include <stdio.h>
#include "solvers.h"
#include <stdlib.h>

int main(){
    int N;
    long long D;

    // Read N (items) and D (budget)
    // Return 0 if input format is incorrect
    if (scanf("%d%lld", &N, &D) != 2) return 0;

    // Use Dynamic Memory Allocation (Heap) instead of Stack
    // Reason: N can be up to 10^5, which might cause Stack Overflow if simple arrays are used
    .
    int *prices = (int *)malloc(N * sizeof(int));
    int *coupons = (int *)malloc(N * sizeof(int));

    // Read input arrays
    for (int i = 0; i < N; i++) scanf("%d", &prices[i]);
    for (int i = 0; i < N; i++) scanf("%d", &coupons[i]);

    // Invoke the optimized solver (v2)
    // This modular design allows switching to solve_v1 easily for testing
    Result res = solve_v2(N, D, prices, coupons);

    // Output results separated by space
    printf("%d %lld\n", res.count, res.left);

    // Clean up memory to prevent memory leaks
    free(prices);
```

```
    free(coupons);

    return 0;
}
```

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

```
#include "solvers.h"
#include <stdio.h>
#include <stdlib.h>

// Data Structure: Array of Structures (AoS)
typedef struct {
    int item_idx;   // Index in prices array
    int coupon_idx; // Index in coupons array
    long long cost; // Cache cost: prices[i] - coupons[j]
} Node;

Node heap[MAX_N];
int heap_size = 0;

// Helper functions for qsort
const int compareAsc(const void *a, const void *b){
    return (*(int *)a - *(int *)b);
}

const int compareDesc(const void *a, const void *b){
    return (*(int *)b - *(int *)a);
}

// Standard Binary Heap Insert (Sift Up)
const void insert(Node item){
    if (heap_size >= MAX_N){
        printf("heap is fullfilled\n");
        return;
    }
    heap[heap_size] = item;
    int index = heap_size;

    // Sift Up Logic
    while (index > 0){
        int parent = (index - 1) / 2;
        if (heap[index].cost < heap[parent].cost){
            Node temp = heap[index];
            heap[index] = heap[parent];
            heap[parent] = temp;
            index = parent;
        } else {
            break;
        }
    }
    heap_size++;
}

// Standard Binary Heap Sift Down
const void siftDown(int index){
    while (2*index + 1 < heap_size){
        int child = 2*index + 1; // Left child

        // Check if right child exists and is smaller
        if (child + 1 < heap_size && heap[child].cost > heap[child + 1].cost)
            child++;

        // Swap if child is smaller than parent
        if (heap[child].cost < heap[index].cost){
            Node temp = heap[child];
            heap[child] = heap[index];
```

```c
            heap[index] = temp;
            index = child;
        } else {
            break;
        }
    }
}

Node deleteMin(){
    if (heap_size == 0){
        printf("empty heap!\n");
        exit(1);
    }
    Node min = heap[0];
    heap[0] = heap[heap_size-1]; // Move last element to root
    heap_size--;
    siftDown(0); // Restore heap property
    return min;
}

Result solve_v1(int N, long long D, int* prices, int* coupons){
    heap_size = 0;

    // Step 1: Sort arrays
    qsort(prices, N, sizeof(int), compareAsc);
    qsort(coupons, N, sizeof(int), compareDesc);

    // Step 2: Full Initialization (The Naive Approach)
    // We calculate the best cost for EVERY item and push all N items into heap.
    for(int i = 0; i < N; i++){
        Node temp;
        temp.item_idx = i;
        temp.coupon_idx = 0; // Pair with the best coupon
        temp.cost = (long long)(prices[i] - coupons[0]);
        insert(temp);
    }

    int count = 0;

    // Step 3: Greedy Loop
    while (heap_size > 0){
        Node current = deleteMin(); // Get cheapest option

        // Check budget
        if (D >= current.cost){
            D -= current.cost;
            count++;
        } else {
            break; // Cannot afford the cheapest, stop.
        }

        // Expansion: Only expand horizontally (same item, next coupon)
        int next_coupon_idx = current.coupon_idx + 1;
        if (next_coupon_idx < N){
            Node next;
            next.item_idx = current.item_idx;
            next.coupon_idx = next_coupon_idx;
            next.cost = (long long)(prices[next.item_idx] - coupons[next.coupon_idx]);
            insert(next);
        }
    }

    Result res = {count, D};
    return res;
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>
#include "solvers.h"

// OPTIMIZATION 1: Structure of Arrays (SoA)
// Improves CPU cache locality compared to 'struct Node'.
long long heap_cost[MAX_N];
int heap_p_idx[MAX_N];
int heap_c_idx[MAX_N];
int size = 0;

// Macro for swapping elements across three arrays
#define SWAP(i, j) {\
    long long temp_cost = heap_cost[i]; heap_cost[i] = heap_cost[j]; heap_cost[j] = temp_cost;\
    int temp_p_idx = heap_p_idx[i]; heap_p_idx[i] = heap_p_idx[j]; heap_p_idx[j] = temp_p_idx;\
    int temp_c_idx = heap_c_idx[i]; heap_c_idx[i] = heap_c_idx[j]; heap_c_idx[j] = temp_c_idx;\
}

const int compareAsc(const void *a, const void *b){
    return (*(int *)a - *(int *)b);
}

const int compareDesc(const void *a, const void *b){
    return (*(int *)b - *(int *)a);
}

// OPTIMIZATION 2: 4-ary Heap (Quad Heap)
// Reduces tree height by half (log4 N = 0.5 * log2 N), reducing sift-up/down depth.
void insert(int p_idx, int c_idx, long long cost){
    if (size > MAX_N) {
        printf("heap is fullfilled\n");
        return;
    }

    heap_cost[size] = cost;
    heap_c_idx[size] = c_idx;
    heap_p_idx[size] = p_idx;

    int index = size;
    // Sift Up for 4-ary heap
    while (index > 0){
        int parent = (index - 1) / 4; // Parent index calculation changes
        if (heap_cost[index] < heap_cost[parent]){
            SWAP(index, parent);
            index = parent;
        } else {
            break;
        }
    }
    size++;
}

const void shiftDown(int index){
    // While at least the first child exists
    while (4*index + 1 < size){
        int child = 4*index + 1;
        int min_child = child;

        // Find the minimum among up to 4 children
        // Loop unrolled for performance
        if (child + 1 < size && heap_cost[child + 1] < heap_cost[min_child]) min_child = child + 1;
        if (child + 2 < size && heap_cost[child + 2] < heap_cost[min_child]) min_child = child + 2;
        if (child + 3 < size && heap_cost[child + 3] < heap_cost[min_child]) min_child = child + 3;

        if (heap_cost[index] > heap_cost[min_child]) {
            SWAP(index, min_child);
```

```
            index = min_child;
        } else {
            break;
        }
    }
}

const void deleteMin(){
    if (size == 0){
        printf("empty heap!\n");
        exit(1);
    }
    // Move last element to root
    heap_cost[0] = heap_cost[size - 1];
    heap_p_idx[0] = heap_p_idx[size - 1];
    heap_c_idx[0] = heap_c_idx[size - 1];

    size--;
    shiftDown(0);
}

Result solve_v2(int N, long long D, int* prices, int* coupons){
    size = 0;
    qsort(prices, N, sizeof(int), compareAsc);
    qsort(coupons, N, sizeof(int), compareDesc);

    // OPTIMIZATION 3: Frontier Expansion (Initialization)
    // Only push the global minimum (0,0) initially.
    // Heap size starts at 1 instead of N.
    insert(0, 0, (long long)(prices[0] - coupons[0]));

    int count = 0;

    while (size > 0){
        // Retrieve min element
        long long current_cost = heap_cost[0];
        int current_p_idx = heap_p_idx[0];
        int current_c_idx = heap_c_idx[0];

        deleteMin();

        // Pruning Check 1: Can we afford it?
        if (D >= current_cost){
            D -= current_cost;
            count++;
        } else {
            break; // Budget exceeded
        }

        // Expansion Strategy:
        // 1. Horizontal: Same item, next coupon (always try to add)
        int next_c_idx = current_c_idx + 1;
        if (next_c_idx < N){
            long long next_cost = (long long)(prices[current_p_idx] - coupons[next_c_idx]);
            // OPTIMIZATION 4: Cost Pruning
            // Only insert if we can theoretically afford it
            if (next_cost <= D){
                insert(current_p_idx, next_c_idx, next_cost);
            }
        }

        // 2. Vertical: Next item, best coupon
        // Only done when we just used the BEST coupon for the current item.
        // This ensures every combination is added exactly once.
        if (current_c_idx == 0){
            int next_p_idx = current_p_idx + 1;
            if (next_p_idx < N){
                long long next_cost = (long long)(prices[next_p_idx] - coupons[0]);
                if (next_cost <= D){ // Pruning
                    insert(next_p_idx, 0, next_cost);
                }
            }
        }
```

```
    }

    Result res = {count, D};
    return res;
}
```