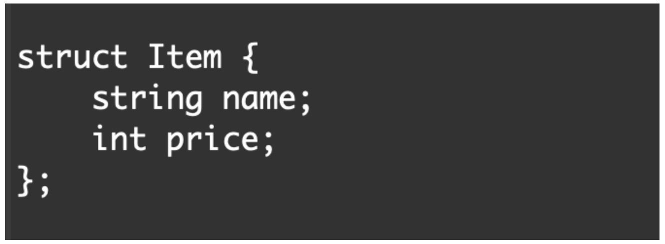
## Overview

This program calculates the maximum number of items that can be purchased without exceeding a user-specified budget. To achieve this, I implemented and analyzed two distinct algorithmic approaches: a Greedy selection method and a Dynamic Programming optimization. Each algorithm processes a list of items, where every item has a name and a price. The objective is to compare how each method functions, the reasoning behind its use, and its relative computational performance.

## Data Structure

Items are stored inside a simple structure:



```
struct Item {
    string name;
    int price;
};
```

This structure keeps the program organized by pairing each item's textual name with its price. Both algorithms operate on a vector<item> container.

# Greedy Algorithm

## Conceptual Explanation

The Greedy algorithm relies on a simple heuristic:

> Always purchase the cheapest items first until the budget runs out.

This strategy is computationally efficient and straightforward. However, because it makes locally optimal decisions at each step without evaluating future possibilities, it does **not** always guarantee an optimal global solution.

## Implementation Walkthrough

### 1. Preparing an index vector

```
vector<int> greedySolve(vector<Item> items, int budget) {
    vector<int> idx(items.size());
    for (int i = 0; i < items.size(); i++) idx[i] = i;
```

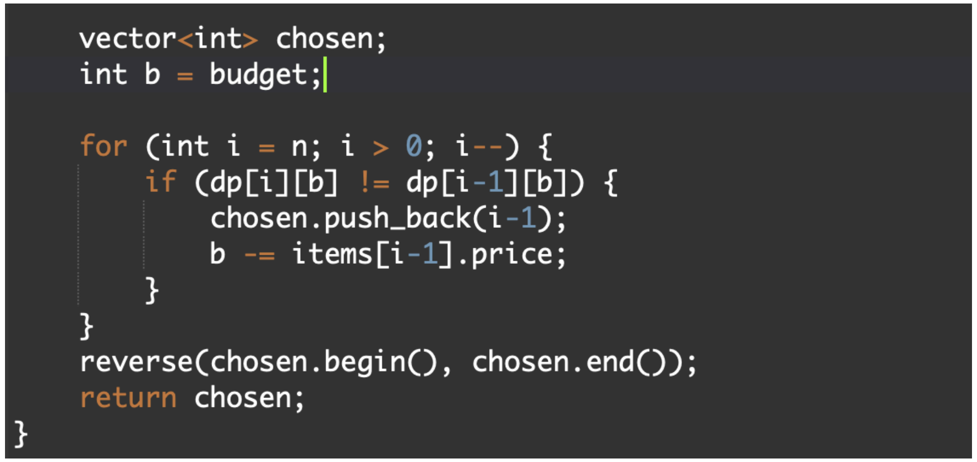
Instead of sorting the actual items, I sort their indices. This keeps the original vector unchanged while allowing price-based ordering.

### 2. Sorting items by ascending price

```
    sort(idx.begin(), idx.end(), [&](int a, int b){
        return items[a].price < items[b].price;
    });
```

This step ensures that the algorithm evaluates items from cheapest to most expensive.

### 3. Selecting items while the budget allows



```
    vector<int> chosen;
    int b = budget;

    for (int i = n; i > 0; i--) {
        if (dp[i][b] != dp[i-1][b]) {
            chosen.push_back(i-1);
            b -= items[i-1].price;
        }
    }
    reverse(chosen.begin(), chosen.end());
    return chosen;
}
```

The algorithm iterates through sorted items and selects any item that fits the remaining budget.

## Greedy Computational Complexity

- **Time Complexity:**
  The dominant operation is sorting → **O(n log n)**
- **Space Complexity:**
  The index array + chosen vector → **O(n)**

This makes the greedy algorithm efficient for large datasets but not guaranteed to produce an optimal solution.

# Dynamic Programming Algorithm

## Conceptual Explanation

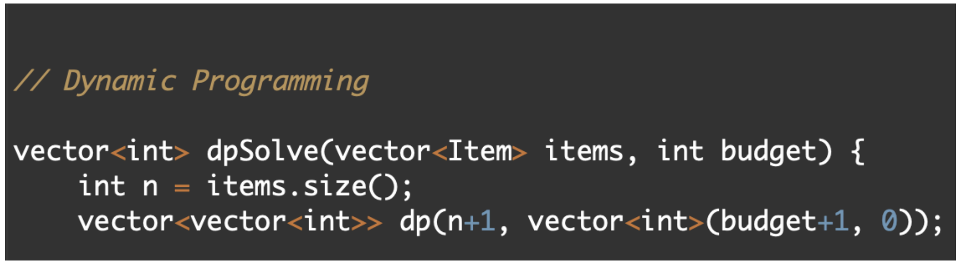
The DP algorithm models the problem as a 0/1 Knapsack problem, where:

- The weight of each item is its price
- The value of each item is always 1 (because we aim to maximize the count of purchased items)
- The capacity of the knapsack is the user's budget

Dynamic Programming evaluates all possible purchase combinations and therefore provides the **optimal solution**.

## Implementation Walkthrough

### 1. Creating the DP table

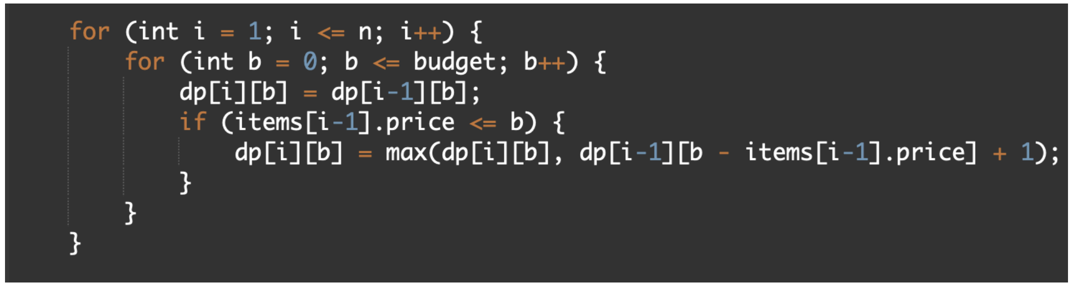


```
// Dynamic Programming

vector<int> dpSolve(vector<Item> items, int budget) {
    int n = items.size();
    vector<vector<int>> dp(n+1, vector<int>(budget+1, 0));
```

The DP table dp[i][b] represents the maximum number of items that can be purchased using the first i items with a budget of b.

### 2. Filling the DP table



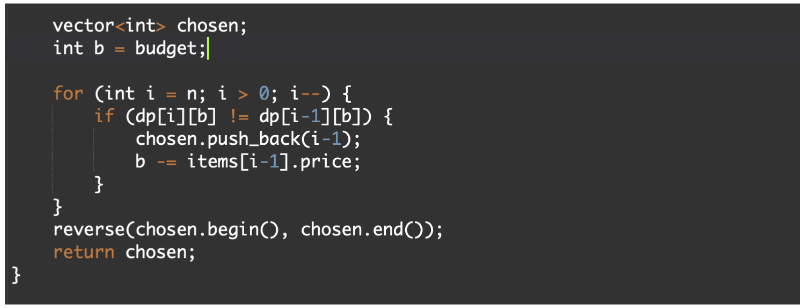
```
for (int i = 1; i <= n; i++) {
    for (int b = 0; b <= budget; b++) {
        dp[i][b] = dp[i-1][b];
        if (items[i-1].price <= b) {
            dp[i][b] = max(dp[i][b], dp[i-1][b - items[i-1].price] + 1);
        }
    }
}
```

For every item and every budget value, the algorithm explores two options:

1. **Exclude the item**
   dp[i-1][b]
2. **Include the item**
   dp[i-1][b - price] + 1

The DP value at each position stores the greater of these two choices.

## 3. Reconstructing the chosen items



```
    vector<int> chosen;
    int b = budget;

    for (int i = n; i > 0; i--) {
        if (dp[i][b] != dp[i-1][b]) {
            chosen.push_back(i-1);
            b -= items[i-1].price;
        }
    }
    reverse(chosen.begin(), chosen.end());
    return chosen;
}
```

Backtracking identifies which items contributed to the optimal solution.

### Dynamic Programming Complexity

Let:

- n = number of items
- B = budget

Then:

- **Time Complexity:** O(n × B)
- **Space Complexity:** O(n × B)

Although slower than the greedy method, DP guarantees the optimal solution for all cases.

## Output Function

```
void printItems(const vector<Item>& items, const vector<int>& chosen) {
    if (chosen.empty()) {
        cout << "No items can be bought.\n";
        return;
    }
    int total = 0;
    for (int i : chosen) {
        cout << items[i].name << " - " << items[i].price << endl;
        total += items[i].price;
    }
    cout << "Total price = " << total << endl;
}
```

This function prints the names and prices of selected items along with the total cost. If no items are chosen, it informs the user.

## Conclusion

This project demonstrates two contrasting strategies for budget-constrained selection:

- The Greedy algorithm provides speed and simplicity by prioritizing the lowest-cost items, but it does not always yield the best possible solution.
- The Dynamic Programming algorithm systematically evaluates all combinations to ensure an optimal selection, at the cost of greater computational resources.

This comparison effectively illustrates the fundamental trade-off between computational efficiency and solution optimality in algorithm design.