Week 12 - Knapsack Problem

Problem Definition

Given a target weight and set of items. Each objects has a value and weight. Determine the subset of items such that the sum of the weights is less than or equal to the target weight and the sum of their values is maximized.

Approach

Backtrack recursion. The idea is to enumerate all combinations and pick the one with best total value.

Code

```
#include <iostream>

#include <vector>

#include <chrono>

using namespace std;

using namespace std::chrono;

// Structure to represent an item

struct item {

    int weight;

    int value;

};

// Function to solve the Knapsack problem using backtracking recursion

int backtrack(const vector<item>& items, int capacity, int currentIndex) {

    // Base case: if either the capacity becomes negative or we have
considered all items

    if (capacity <= 0 || currentIndex < 0) {

        return 0;

    }

    // If the weight of the current item is more than the remaining capacity,
skip it

    if (items[currentIndex].weight > capacity) {
```

```
        return backtrack(items, capacity, currentIndex - 1);
    }
    // Try both including and excluding the current item
    int includeCurrent = items[currentIndex].value +
                         backtrack(items, capacity -
items[currentIndex].weight, currentIndex - 1);
    int excludeCurrent = backtrack(items, capacity, currentIndex - 1);
    // Return the maximum of the two values
    return max(includeCurrent, excludeCurrent);
}
// Function to solve the 0/1 Knapsack problem and measure running time
int knapsack(const vector<item>& items, int capacity) {
    // Start the timer
    auto start_time = high_resolution_clock::now();
    // Start from the last item
    int currentIndex = items.size() - 1;
    int maxValue = backtrack(items, capacity, currentIndex);
    // Stop the timer
    auto stop_time = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop_time - start_time);
    // Print the running time in seconds and microseconds
    cout << "Running time: " << duration.count() / 1e6 << " seconds" << endl;
    // Print the theoretical big O notation
    cout << "Big O notation: O(2^n)" << endl;
    return maxValue;
}
int main() {
    vector<item> items = {{2, 10}, {5, 25}, {1, 7}, {3, 15}};
```

```
    int capacity = 7;

    int maxValue = knapsack(items, capacity);

    cout << "Maximum value that can be obtained: " << maxValue << endl;
    return 0;
}
```

Important Steps

- **1: Define Item Structure**
  - Create a structure (`struct`) to represent items, with attributes for weight and value.
- **2: Recursive Function**
  - Implement a recursive helper function (`backtrack`) that explores combinations of items.
  - Base Case:
    - If remaining capacity is negative or all items are considered, return 0.
  - Decision:
    - If the weight of the current item exceeds remaining capacity, skip to the next item.
  - Explore:
    - Try both including and excluding the current item through recursive calls.
  - Return:
    - Return the maximum value obtained from the two possibilities.

References

(n.d.). Stanford University. https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1228/section/section4/Knapsack%20Problem%20(Optional%20Section%20Slides).pdf