The 0/1 Knapsack Problem: Brute Force Approach in C++

A comprehensive exploration of solving the classic 0/1 Knapsack Problem using brute force techniques in C++, designed for the Algorithms and Data Structures course.

What is the 0/1 Knapsack Problem?

The 0/1 Knapsack Problem is a fundamental optimization challenge in computer science with numerous real-world applications:

- Given a set of **n** items, each with a weight and a value
- A knapsack with a maximum weight capacity W
- Goal: Select items to **maximize total value** without exceeding capacity
- The "0/1" constraint: Each item can only be taken completely (1) or left behind (0)

This problem belongs to the NP-hard complexity class, making it computationally intensive for large inputs.

Brute Force Approach: Conceptual Overview

Generate All Combinations

Create all 2^n possible subsets of the n items (each item is either included or excluded)

Evaluate Each Combination

For each subset, calculate total weight and value

Filter Valid Solutions

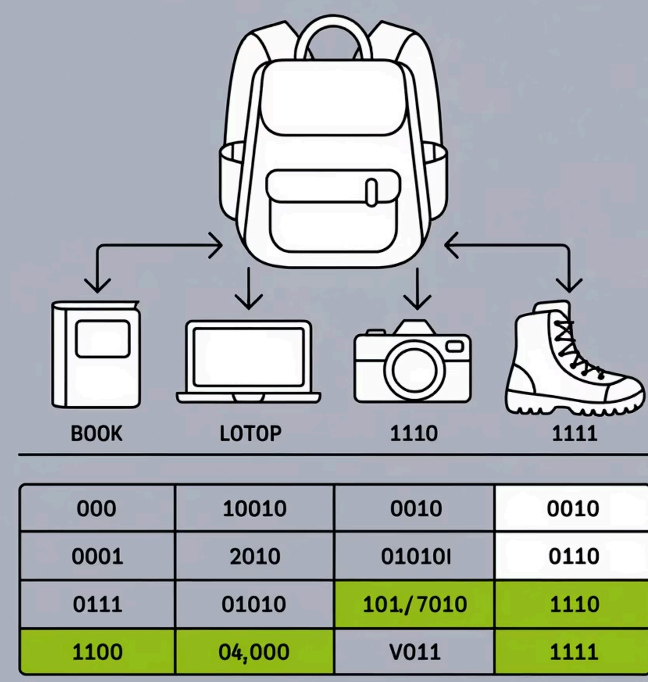
Discard combinations where total weight exceeds capacity W

Select Optimal Solution

Choose the valid combination with the highest total value

Time Complexity: $O(2^n)$ where n is the number of items

Space Complexity: $O(n)$ to store the current combination



Key Functions in Brute Force Knapsack Implementation

1

totalCombinations

Uses bit manipulation to enumerate all 2^n possible subsets of items

- Each bit position represents including (1) or excluding (0) an item
- Iterates from 0 to 2^n-1, creating a unique combination for each number

1

evaluate_mask

Tracks the best valid combination found so far

- Checks if current combination is valid (weight ≤ capacity)
- Updates maximum value and selected items if better solution found

2

update_best_if_better

Computes the total weight and value for a given subset

- Iterates through the bit representation of a combination
- Adds weight and value of included items to running totals

2

printSolution

Outputs the optimal solution details

- Displays selected items, their weights and values
- Shows total weight used and maximum value achieved

These functions work together in an iterative approach that avoids recursion while still exploring the complete solution space.

C++ Code: Brute Force Knapsack Implementation

```
#include <iostream>
using namespace std;

struct Item {
    int weight;
    int value;
};

// Devuelve el máximo de dos enteros (evitamos <algorithm>)
inline int imax(int a, int b) { return (a > b) ? a : b; }

// Fuerza bruta con poda temprana y mejores tipos para máscaras y acumuladores
void knapsackBruteForce(const Item items[], int n, int capacity) {
    // Validaciones básicas
    if (n <= 0 || capacity <= 0) {
        cout << "No hay solucion valida (n<=0 o capacity<=0).\n";
        return;
    }

    // Importante: usamos unsigned para la mascara (bits) y 1u en los shifts
    // para evitar comportamientos indefinidos de desplazamiento con signo.
    // Limite: n debe ser menor al número de bits de 'unsigned' (usualmente 32).
    const int BITS = (int)(8 * sizeof(unsigned));
    if (n >= BITS) {
        cout << "Advertencia: n demasiado grande para iterar todas las combinaciones con 'unsigned'.\n";
        cout << "Reducir n o usar un tipo de mascara mas grande.\n";
        return;
    }

    int bestValue = 0;
    unsigned bestMask = 0u;
    int bestWeight = 0;

    const unsigned totalComb = (1u << n);  // 2^n combinaciones

    for (unsigned mask = 0u; mask < totalComb; ++mask) {
        // Usamos 'int' para acumuladores; si esperas numeros muy grandes, usa long long
        int currWeight = 0;
        int currValue  = 0;

        // Recorremos los n bits; poda si nos pasamos de capacidad
        for (int j = 0; j < n; ++j) {
            if (mask & (1u << j)) {
                currWeight += items[j].weight;
                // Poda temprana: si ya nos pasamos, no seguimos sumando
                if (currWeight > capacity) {
                    // Salimos del bucle interno temprano
                    currValue = -1; // bandera para indicar invalido
                    break;
                }
                currValue  += items[j].value;
            }
        }

        if (currValue < 0) continue; // combinación inválida por peso

        // Actualizamos el mejor (si empata en valor, preferimos menor peso)
        if (currValue > bestValue || (currValue == bestValue && currWeight < bestWeight)) {
            bestValue = currValue;
            bestMask  = mask;
            bestWeight = currWeight;
        }
    }

    // Reporte
    cout << "Items a incluir en la mochila:\n";
    for (int i = 0; i < n; ++i) {
        if (bestMask & (1u << i)) {
            cout << "- Item " << (i + 1)
                 << ": Weight=" << items[i].weight
                 << ", Value="  << items[i].value << "\n";
        }
    }
    cout << "Total Weight: " << bestWeight << "/" << capacity << "\n";
    cout << "Maximum Value: " << bestValue << "\n";
}

int main() {
    Item items[] = {
            {10, 60},  // Item 1
            {20, 100}, // Item 2
            {30, 120}, // Item 3
            {15, 80}   // Item 4
    };
    int n = (int)(sizeof(items) / sizeof(items[0]));
    int capacity = 50;

    knapsackBruteForce(items, n, capacity);
    return 0;
}
```