BANGLADESH UNIVERSITY
Department of Computer Science & Engineering
CSE2202 – Algorithm Lab

LAB REPORT

SUBMITTED BY:

Student's Name	: Sheoti
Batch	: 63
Student ID	: 202221063026
Course Code	: CSE2202
Course Title	: Algorithm Lab

SUBMITTED TO:

Muhammad Minoar Hossain

Lecturer

Department of Computer Science & Engineering

Bangladesh University

Date of Submission : 25/11/2023

Marks:

Knapsack Algorithm

Knapsack Algorithm: The Knapsack algorithm is a common optimization problem that deals with selecting a subset of items with maximum value or minimum cost, subject to a constraint on the total weight or size of the selected items. This problem is often referred to as the "knapsack problem" because it can be visualized as filling a knapsack with a set of items, each having a weight and a value.

There are different variations of the knapsack problem, but the most well-known ones are the 0/1 Knapsack Problem and the Fractional Knapsack Problem.

0/1 Knapsack Problem: In this version, each item can either be selected or rejected; there's no possibility of taking a fraction of an item. The goal is to maximize the total value of the selected items without exceeding the weight capacity of the knapsack.

Fractional Knapsack Problem: In this version, the items can be divided into fractions, meaning you can take a part of an item. The objective is still to maximize the total value of the selected items, but now you can take fractions of items to achieve this.

Use of Knapsack Algorithm: The Knapsack algorithm is a combinatorial optimization problem that involves selecting a subset of items with given weights and values to maximize the total value while not exceeding a given weight constraint. The Knapsack algorithm and its variations have several practical applications in different domains. Here are some common uses:

- ❖ **Resource Allocation:** In resource allocation problems, such as project management or task scheduling, the Knapsack algorithm can be used to optimize the assignment of resources to tasks based on their respective values and resource requirements.
- ❖ **Finance and Investment:** The Knapsack algorithm can be applied to portfolio optimization, where the goal is to maximize the return on investment given a set of financial assets with different expected returns and risk levels, subject to a constraint on the total investment amount.

- **Supply Chain Management:** In supply chain management, the Knapsack algorithm can help in selecting the most valuable items to include in a shipment based on their weights and values, considering constraints such as the maximum weight capacity of the shipment.
- **Data Compression:** The Knapsack problem can be used in data compression, where the goal is to select a subset of data (items) to maximize the compression ratio while staying within a certain storage limit.
- **Broadcast Scheduling:** In wireless communication networks, the Knapsack algorithm can be applied to optimize the scheduling of data transmissions to maximize the overall data throughput or minimize the transmission delay, considering the limited capacity of the communication channels.
- **DNA Sequencing:** In bioinformatics, the Knapsack algorithm can be used for DNA sequence assembly, where the goal is to assemble a sequence with the highest coverage based on overlapping fragments while considering constraints on the computational resources available.
- **Resource Management in Cloud Computing:** In cloud computing environments, where resources are allocated to different tasks or virtual machines, the Knapsack algorithm can help in optimizing resource utilization and minimizing costs by selecting the most valuable combination of resources.
- **Multi-objective Optimization:** The Knapsack problem can be extended to handle multiple objectives, such as maximizing profit while minimizing cost or maximizing value while staying within multiple resource constraints.

In general, the Knapsack algorithm provides a flexible and powerful tool for solving optimization problems involving the selection of items subject to constraints, making it applicable to a wide range of real-world scenarios.

Methods of Knapsack Algorithm: There are two methods of Knapsack algorithm.

i. Greedy method
ii. Dynamic method

Greedy method

Greedy method: The greedy method is a popular strategy for resolving the classic optimization problem known as the "knapsack problem," in which a subset of items is chosen from a set of available items in order to maximize the total profit or value of the chosen items while staying within a predetermined weight restriction.

The greedy strategy in the context of the backpack problem entails sorting the objects based on their value-to-weight ratio, or the ratio of an object's value to its weight. After the items are sorted, the item with the highest value-to-weight ratio that will fit in the backpack's remaining capacity is chosen iteratively via the Greedy algorithm. Until the backpack is completely filled or no more goods can be loaded, this process is repeated.

This is a detailed breakdown of the Greedy approach to the fractional knapsack problem-

i. **Sort items:** Puts things in descending order of value to weight.
ii. **Set up the variables:** Set the overall gain to 0 and the current knapsack weight to 0.
iii. **Iterate over items:** Go over the elements that have been sorted again. Verify that each item fits inside the knapsack without weighing more than the allowed amount. If the item is addable, do so and update the total gain and current knapsack weight to reflect the addition.
iv. **Termination state:** When the knapsack is full or there are no more items to add, the iteration should end.
v. **Output:** The ultimate total profit figure indicates the highest profit that the greedy approach is capable of producing.

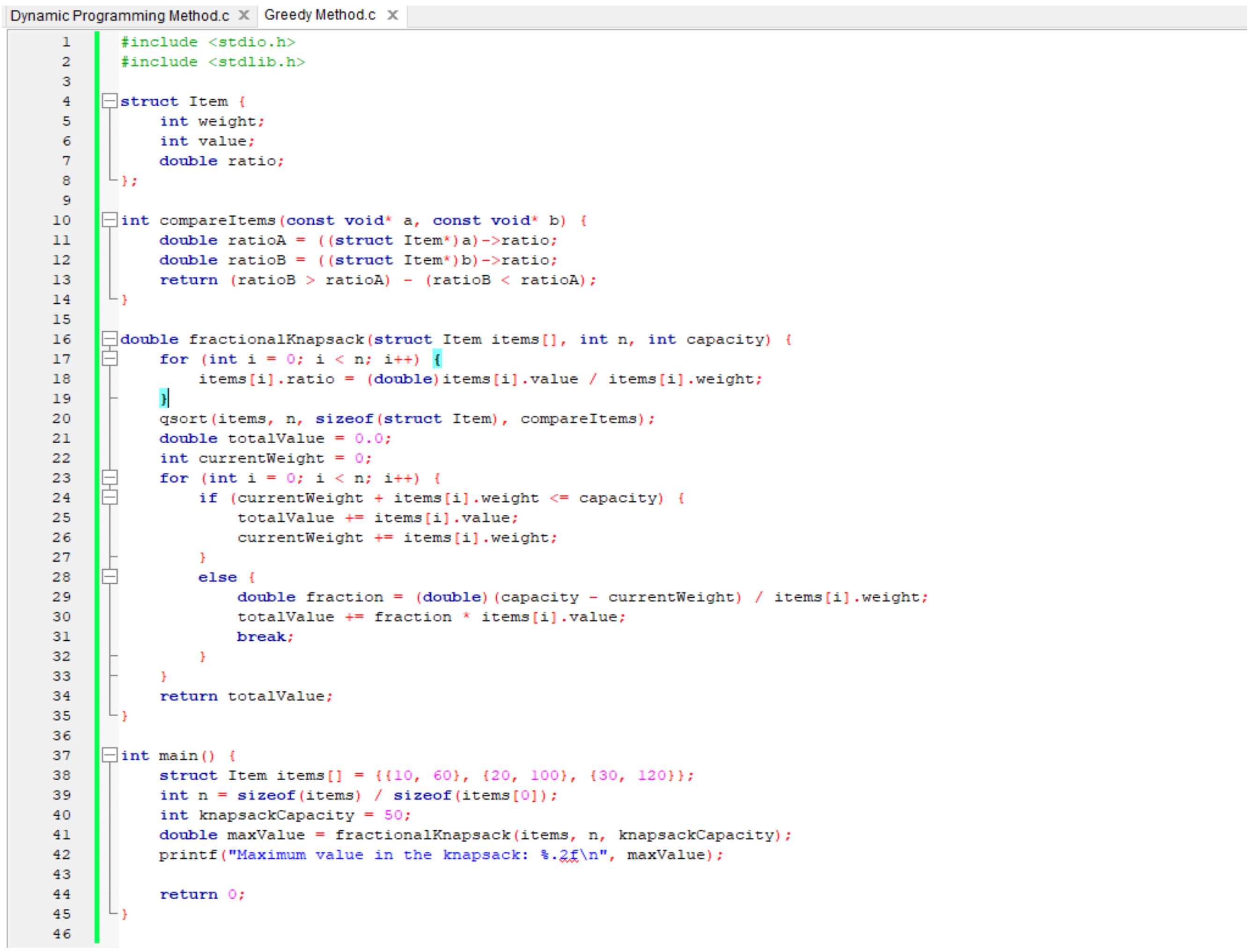
Benefits:

- **Effectiveness for Fractional Knapsack:** Since items in the Fractional Knapsack problem can be separated, the Greedy approach is especially effective in solving it.
- **Easy to use:** The algorithm can be quickly solved because it is simple to comprehend and apply.

Drawbacks:

- **Unguaranteed Optimality:** Although the greedy approach solves the fractional knapsack problem optimally, it is not assured to solve the 0/1 knapsack problem optimally. It's possible that greedy algorithms don't always identify the globally best answer.

C Code Example for Greedy Method:



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

struct Item {
    int weight;
    int value;
    double ratio;
};

int compareItems(const void* a, const void* b) {
    double ratioA = ((struct Item*)a)->ratio;
    double ratioB = ((struct Item*)b)->ratio;
    return (ratioB > ratioA) - (ratioB < ratioA);
}

double fractionalKnapsack(struct Item items[], int n, int capacity) {
    for (int i = 0; i < n; i++) {
        items[i].ratio = (double)items[i].value / items[i].weight;
    }
    qsort(items, n, sizeof(struct Item), compareItems);
    double totalValue = 0.0;
    int currentWeight = 0;
    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= capacity) {
            totalValue += items[i].value;
            currentWeight += items[i].weight;
        }
        else {
            double fraction = (double)(capacity - currentWeight) / items[i].weight;
            totalValue += fraction * items[i].value;
            break;
        }
    }
    return totalValue;
}

int main() {
    struct Item items[] = {{10, 60}, {20, 100}, {30, 120}};
    int n = sizeof(items) / sizeof(items[0]);
    int knapsackCapacity = 50;
    double maxValue = fractionalKnapsack(items, n, knapsackCapacity);
    printf("Maximum value in the knapsack: %.2f\n", maxValue);

    return 0;
}

```

Analysis: Because of the sorting step, where n is the number of elements, the Greedy Fractional Knapsack algorithm has a time complexity of O (n log n). When it comes to the 0/1 knapsack problem, where the elements cannot be divided, the greedy technique might not always provide the best answer, but it does perform optimally for the fractional knapsack problem.

Dynamic method

Dynamic method: The 0/1 knapsack issue, in which elements cannot be shared and must be included or eliminated, is frequently solved using the dynamic programming technique. The optimal solution to the knapsack problem is guaranteed to be found using the dynamic programming method, which is a more advanced approach.

Conversely, the greedy approach might not necessarily get the best result. The knapsack problem's sub-problems are solved in a table by the dynamic programming approach.

This is a detailed breakdown of the Dynamic Programming method to the 0/1 knapsack problem-

i. **Make a table:** Make a table with capacity+1 columns and n+1 rows, where n is the number of elements.
ii. **Set up the table:** Enter 0 in the table's first row and first column.
iii. **Finish the table:** Beginning with the second row and second column, loop through the table. Verify whether the item can fit in the knapsack in each cell in the table without going over the weight restriction. Determine the maximum profit for the current knapsack capacity and a subset of items, accounting for both the current item and the best answers to the earlier sub-problems, if the item can be added. In the cell of the current table, record the maximum win.
iv. **Find the best solution:** The best possible outcome for the entire knapsack and all items is represented by the final cell of the table, which contains the ideal solution.
v. **Track down the solution:** Trace from the last cell of the table to identify the components of the optimal solution, then pick the components that were part of the best answers to the earlier sub-problems.

Benefits:

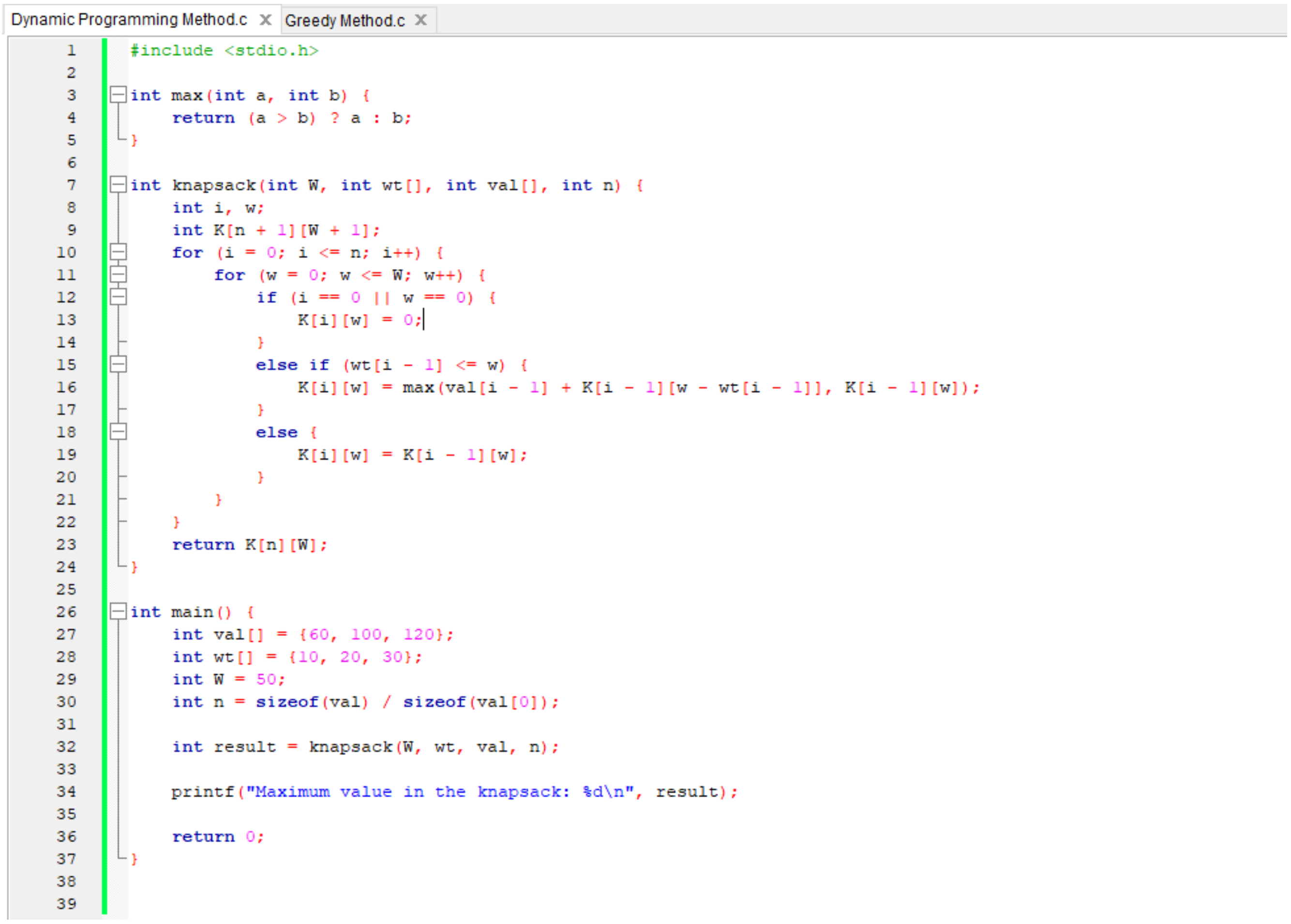
- **Guaranteed Optimality:** The dynamic programming approach ensures that the 0/1 knapsack problem has an optimal solution.

- ❖ **Versatility:** Dynamic programming is a versatile technique that may be applied to a wide range of optimization problems.

Drawbacks:

- ❖ **Computational complexity:** When dealing with larger issue instances, the dynamic programming solution typically has a higher temporal complexity than the greedy method.
- ❖ **Memory Requirements:** Higher memory needs arise from the requirement to store interim results in a table under the dynamic programming approach.

C Code Example for Dynamic Programming Method:



```c
#include <stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int knapsack(int W, int wt[], int val[], int n) {
    int i, w;
    int K[n + 1][W + 1];
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                K[i][w] = 0;
            }
            else if (wt[i - 1] <= w) {
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            }
            else {
                K[i][w] = K[i - 1][w];
            }
        }
    }
    return K[n][W];
}

int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);

    int result = knapsack(W, wt, val, n);

    printf("Maximum value in the knapsack: %d\n", result);

    return 0;
}
```

Analysis: O (n * c) is the time complexity of the dynamic programming solution to the 0/1 knapsack problem, where n is the number of elements and c is the capacity. Through memorizing, the dynamic programming approach effectively minimizes superfluous calculations.

Conclusion: In summary, The Knapsack problem is considered an NP-hard problem, meaning there is no known polynomial time algorithm to solve it optimally in all cases. However, dynamic programming can be used to solve the 0/1 Knapsack Problem efficiently, and a greedy algorithm is often used for the Fractional Knapsack Problem.

The particular needs of the problem determine whether to use the greedy or dynamic programming approach. The greedy technique is an appropriate choice if a fast and approximate solution is sufficient. However, the dynamic programming approach is the best choice if an exact optimal solution is needed.