Objective: To implement and analyze GREEDY AND DYNAMIC approaches for solving Knapsack problem and compare their time and space complexities (through graph)

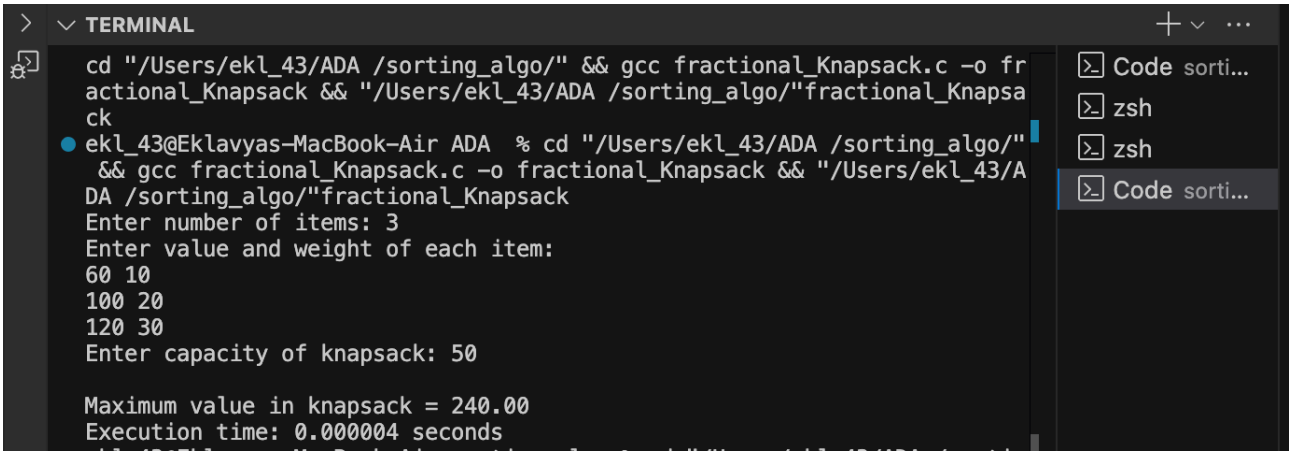
4(a) Fractional Knapsack

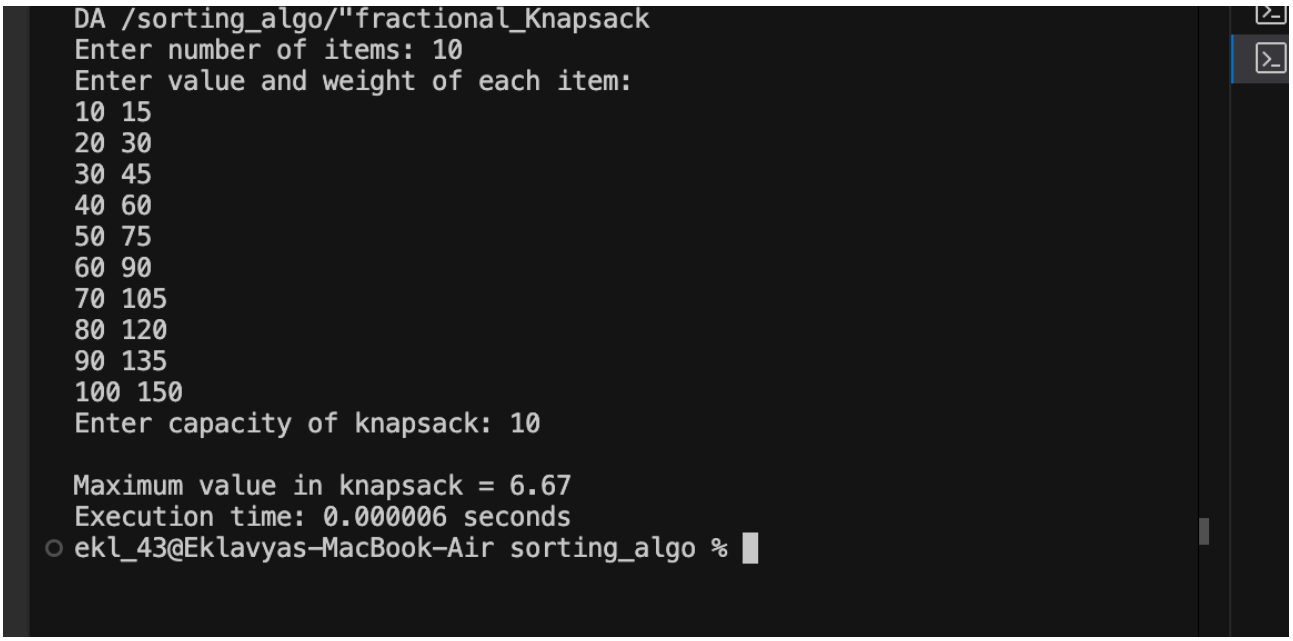
Code:

```
#include <stdio.h>
#include <time.h>

struct Item {
    int value, weight;
    float ratio;
};

// Function to sort items in decreasing order of ratio
void sortItems(struct Item items[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (items[j].ratio < items[j + 1].ratio) {
                struct Item temp = items[j];
                items[j] = items[j + 1];
                items[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n;
    float capacity;
    printf("Enter number of items: ");
    scanf("%d", &n);

    struct Item items[n];
    printf("Enter value and weight of each item:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &items[i].value, &items[i].weight);
        items[i].ratio = (float)items[i].value / items[i].weight;
    }

    printf("Enter capacity of knapsack: ");
    scanf("%f", &capacity);

    clock_t start, end;
    start = clock(); // start timer

    sortItems(items, n);

    float totalValue = 0.0;
    for (int i = 0; i < n; i++) {
        if (items[i].weight <= capacity) {
            totalValue += items[i].value;
            capacity -= items[i].weight;
```

```
        } else {
            totalValue += items[i].value * (capacity / items[i].weight);
            break;
        }
    }

    end = clock(); // end timer double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("\nMaximum value in knapsack = %.2f", totalValue);
    printf("\nExecution time: %.6f seconds\n", time_taken);

    return 0;
}
```

Output:





Algorithm:

Step 1: Input the number of items n, knapsack capacity W,
weights w[i] and values v[i] for all items.

Step 2: Calculate value/weight ratio for each item.

Step 3: Sort all items in decreasing order of value/weight ratio.

Step 4: Initialize total_value = 0 and remaining_capacity = W.

Step 5: For each item i in sorted order:
    if (w[i] ≤ remaining_capacity)
        take the whole item
        total_value += v[i]
        remaining_capacity -= w[i]
    else
        take fractional part:
        total_value += v[i] * (remaining_capacity / w[i])
        remaining_capacity = 0
        break

Step 6: Print total_value as maximum profit.

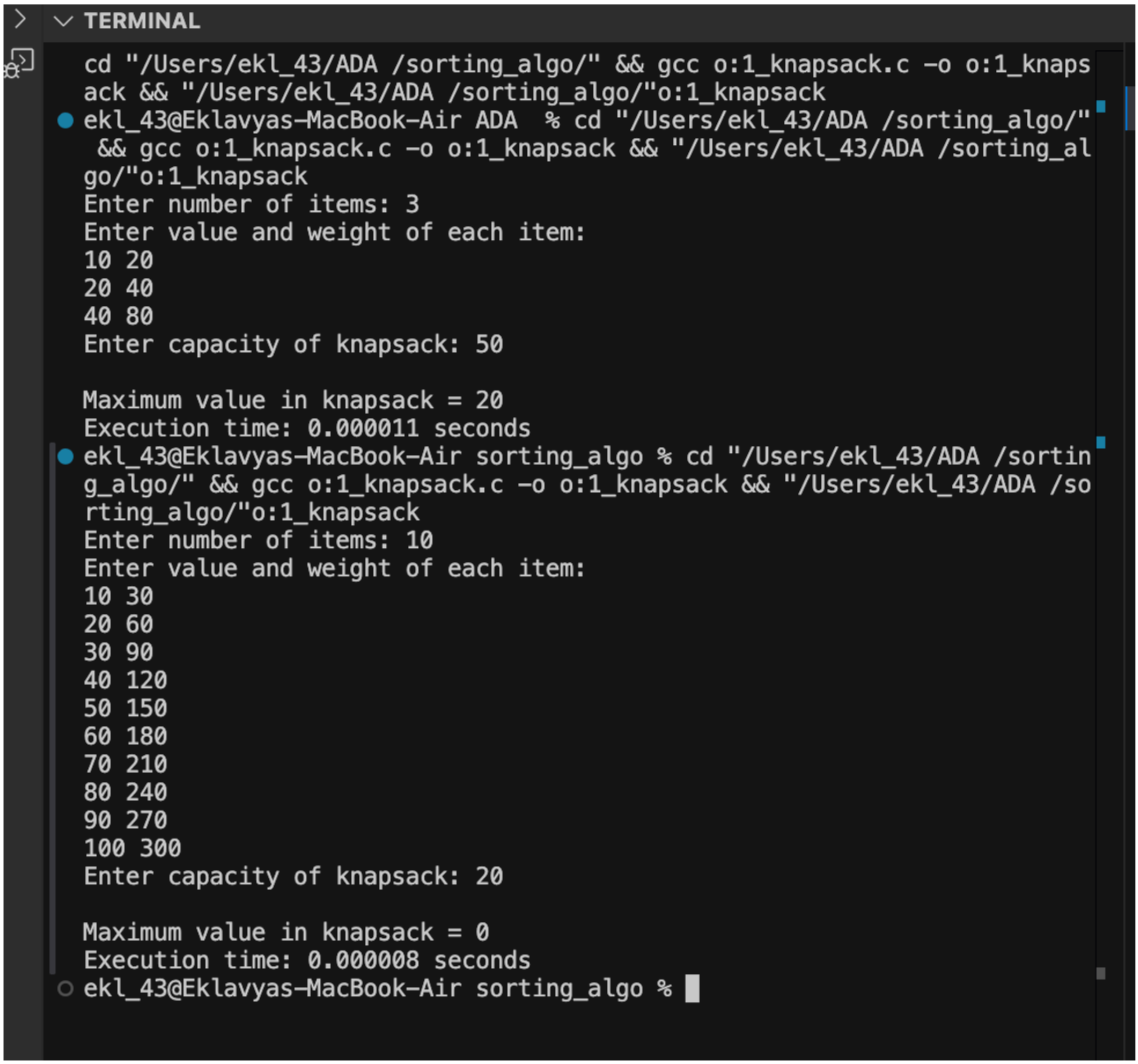
4(b) 0/1 Knapsack

Code:

```
#include <stdio.h>
#include <time.h>

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

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

int main() {
    int n, W;
    printf("Enter number of items: ");
    scanf("%d", &n);

    int val[n], wt[n];
    printf("Enter value and weight of each item:\n");
    for (int i = 0; i < n; i++)
```

```
        scanf("%d %d", &val[i], &wt[i]);

    printf("Enter capacity of knapsack: ");
    scanf("%d", &W);

    clock_t start, end;
    start = clock(); // start timer

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

    end = clock(); // end timer
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("\nMaximum value in knapsack = %d", result);
    printf("\nExecution time: %.6f seconds\n", time_taken);

    return 0;
}
```

Output:



Algorithm:

Step 1: Input number of items n, knapsack capacity W,
weights w[i] and values v[i].

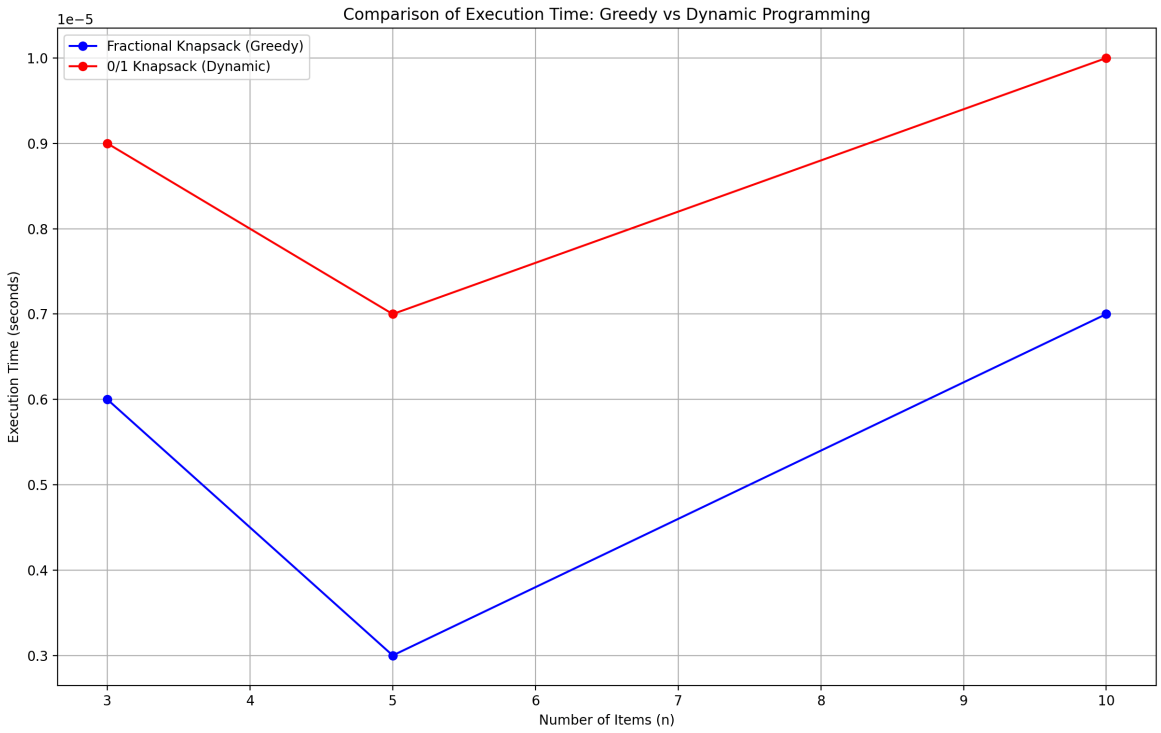
Step 2: Create a 2D array dp[n+1][W+1] where
dp[i][j] represents max value with i items and capacity j.

Step 3: Initialize dp[0][j] = 0 and dp[i][0] = 0.

Step 4: For i = 1 to n:
For j = 1 to W:
if (w[i-1] <= j)
dp[i][j] = max(v[i-1] + dp[i-1][j - w[i-1]], dp[i-1][j])
else
dp[i][j] = dp[i-1][j]

Step 5: Result = dp[n][W]

Step 6: Print dp[n][W] as maximum profit.



Conclusion:

Both Fractional Knapsack (Greedy Approach) and 0/1 Knapsack (Dynamic Programming Approach) were successfully implemented and executed.
The Fractional Knapsack algorithm selects items based on the highest value-to-weight ratio, allowing fractional inclusion of items. It provides an optimal solution for problems where items can be divided. Its time complexity is O(n log n) due to sorting, and it performs efficiently even for large input sizes.

The 0/1 Knapsack algorithm, implemented using Dynamic Programming, provides an exact optimal solution for cases where items cannot be divided. However, it requires more computation and memory, with time and space complexity O(nW), where W is the knapsack capacity.
The experimental comparison and graph show that the Greedy approach executes faster than the Dynamic Programming approach, but the Dynamic method guarantees exact optimality for 0/1 cases.

Hence, both algorithms were implemented successfully, tested for various inputs, and their time and space complexities were analyzed and compared.