**Practical-5 Knapsack Problem**

Part A - Fractional Knapsack

Pseudo Code:

Structure Item:

profit: integer

weight: integer

ratio: float

Function compare(itemA, itemB):

If itemA.ratio < itemB.ratio: return 1 If itemA.ratio > itemB.ratio: return -1 Else: return 0

Function Knapsack(capacity, n, profits[], weights[]): Create array items[0..n-1] of type Item

For i from 0 to n - 1:

items[i].profit = profits[i] items[i].weight = weights[i] items[i].ratio = profits[i] / weights[i] // Compute profit-to-weight ratio

Sort items in descending order by ratio using compare() currentWeight = 0

finalProfit = 0.0

For i from 0 to n - 1:

If currentWeight + items[i].weight ≤ capacity: currentWeight += items[i].weight

finalProfit += items[i].profit Else:

remainingCapacity = capacity - currentWeight fraction = remainingCapacity / items[i].weight finalProfit += items[i].profit \* fraction

Break // Can't fit more items Return finalProfit

**C code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct {

    int value;

    int weight;

    double ratio;

} Item;

void merge(Item arr[], int l, int m, int r) {

    int i, j, k;

    int n1 = m - l + 1;

    int n2 = r - m;

    Item \*L = (Item \*)malloc(n1 \* sizeof(Item));

    Item \*R = (Item \*)malloc(n2 \* sizeof(Item));

    if (L == NULL || R == NULL) {

        printf("Memory allocation failed!\n");

        return;

    }

    for (i = 0; i < n1; i++){

        L[i] = arr[l + i];

    }

    for (j = 0; j < n2; j++){

        R[j] = arr[m + 1 + j];

    }

    i = 0;

    j = 0;

    k = l;

    while (i < n1 && j < n2) {

        if (L[i].ratio >= R[j].ratio) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

    free(L);

    free(R);

}

void mergeSort(Item arr[], int l, int r) {

    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);

        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);

    }

}

double fractionalKnapsack(int capacity, Item items[], int n) {

    for (int i = 0; i < n; i++) {

        items[i].ratio = (double)items[i].value / items[i].weight;

    }

    mergeSort(items, 0, n - 1);

    double totalValue = 0.0;

    int currentWeight = 0;

    for (int i = 0; i < n; i++) {

        if (currentWeight + items[i].weight <= capacity) {

            currentWeight += items[i].weight;

            totalValue += items[i].value;

        } else {

            int remainingCapacity = capacity - currentWeight;

            totalValue += items[i].ratio \* remainingCapacity;

            break;

        }

    }

    return totalValue;

}

int main() {

    int n, capacity;

    printf("Enter the number of items: ");

    scanf("%d", &n);

    Item \*items = (Item \*)malloc(n \* sizeof(Item));

    if (items == NULL) {

        printf("Memory allocation failed!\n");

        return 1;

    }

    for (int i = 0; i < n; i++) {

         items[i].weight = rand() % 20 + 1;

          items[i].value = rand() % 20 + 1;

         if(n < 20){

 printf("Item %d: Weight = %d, Value = %d\n", i + 1, items[i].weight, items[i].value);

      }

    }

    printf("Enter the capacity of the knapsack: ");

    scanf("%d", &capacity);

    clock\_t start = clock();

    double maxValue = fractionalKnapsack(capacity, items, n);

    clock\_t end = clock();

    double time\_spent = (double)(end - start) / CLOCKS\_PER\_SEC;

    printf("Maximum value in fractional knapsack = %.2f\n", maxValue);

    printf("\nTime taken: %.6f seconds\n", time\_spent);

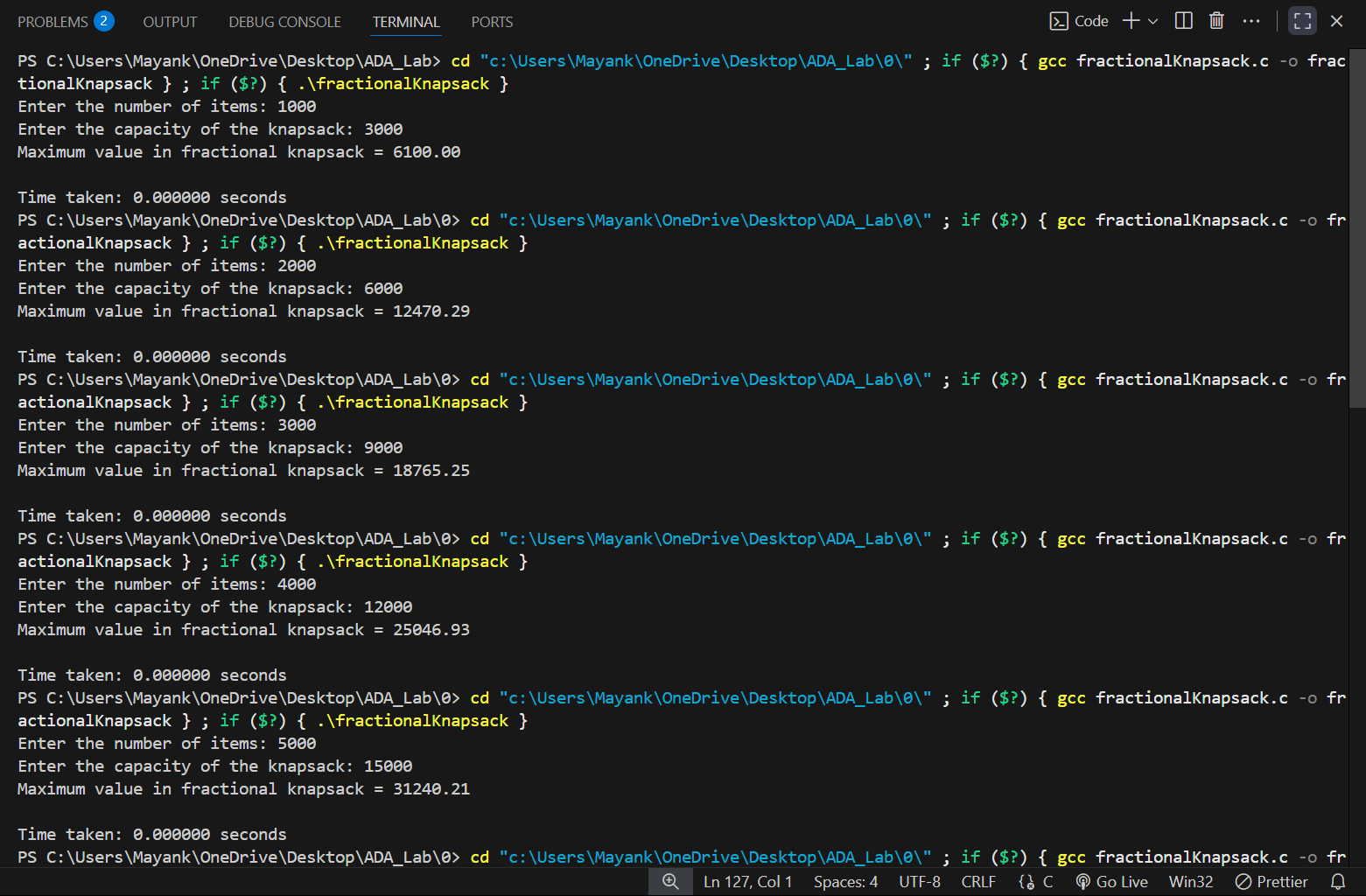
    free(items);

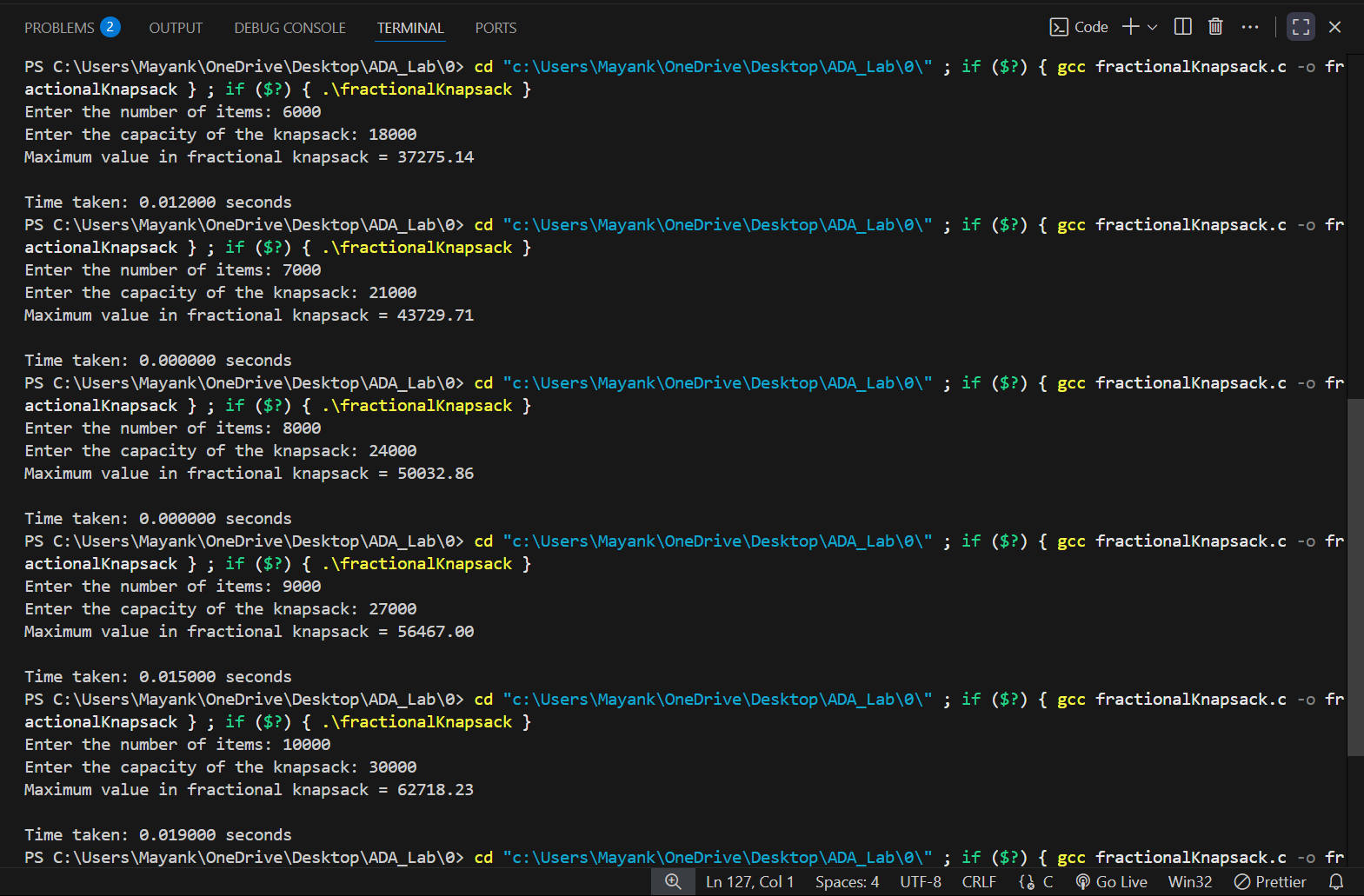
    items = NULL;

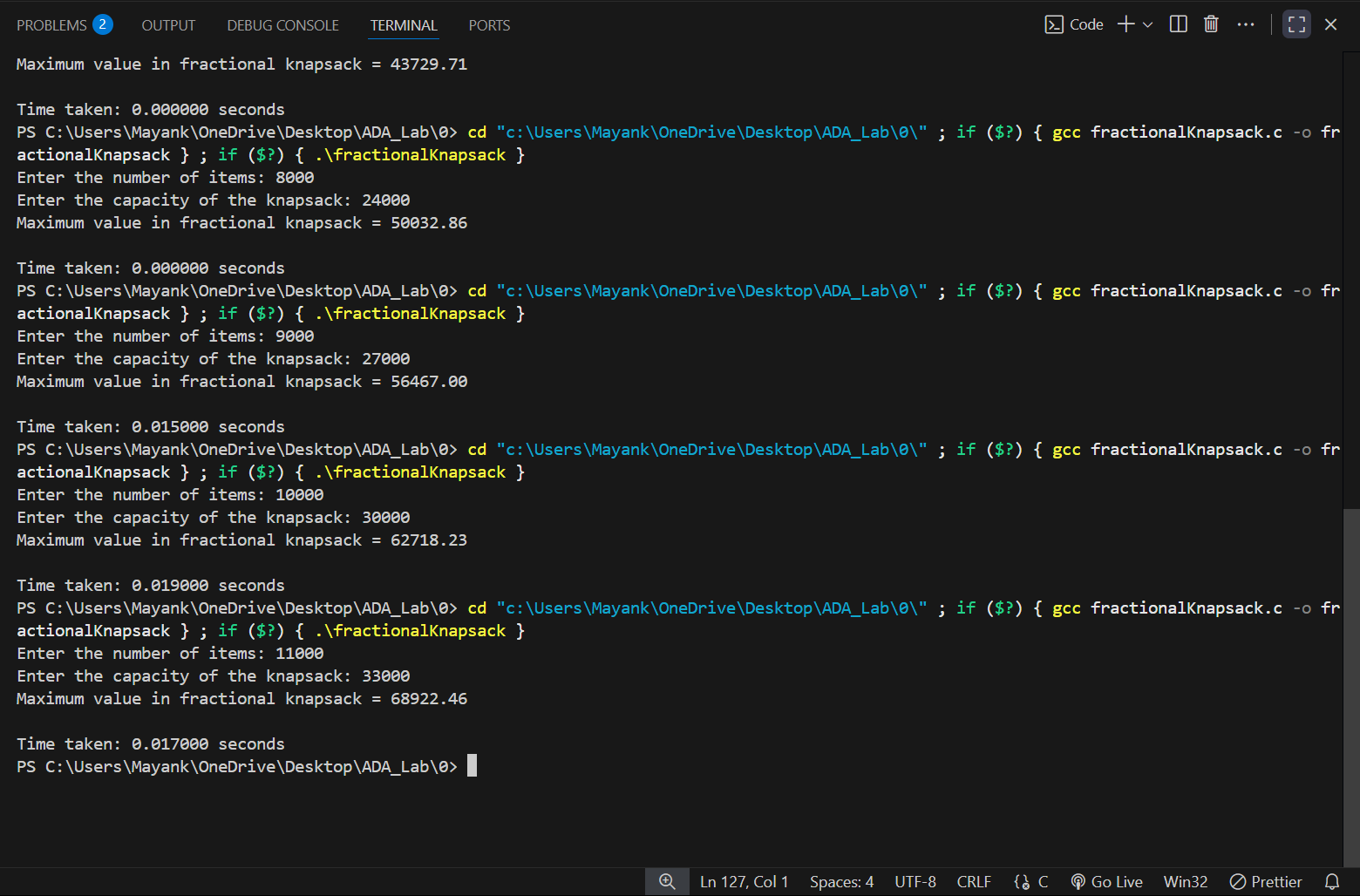
    return 0;

}

**OUTPUT:**

****

****

****

**PYTHON CODE:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [1000,2000,3000,4000,5000,6000,7000,8000,9000,10000, 11000]

time\_taken = [0.000000 ,0.000000, 0.000000 , 0.000000 , 0.000000 , 0.012000 , 0.000000 , 0.000000 , 0.015000 , 0.019000, 0.017000 ]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='-')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

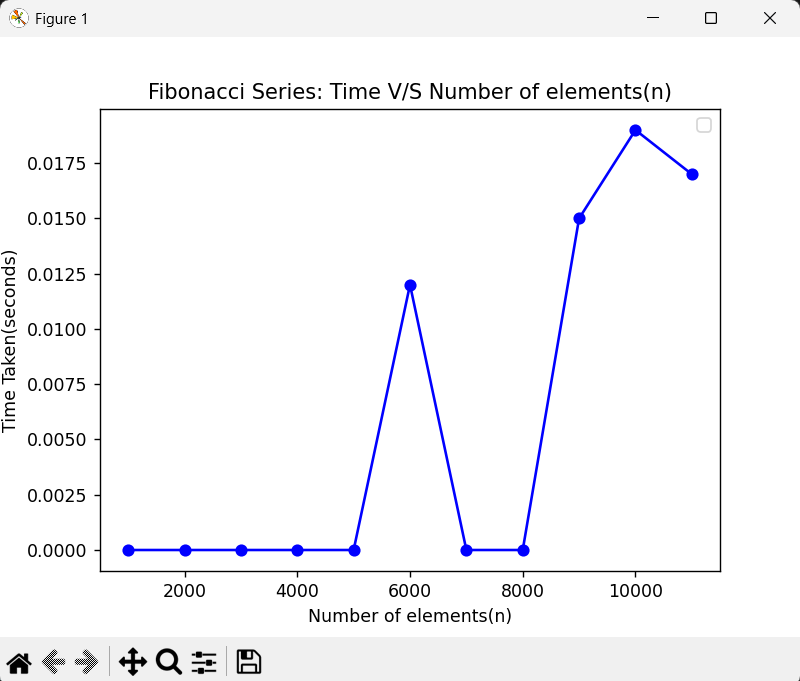
plt.title("Fibonacci Series: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**

****

Part B - 0/1 Knapsack

Pseudo Code :

Function Knapsack(W, n, profits[], weights[]): Create a 2D array table of size (n + 1) x (W + 1)

For i from 0 to n:

For w from 0 to W: If i == 0 or w == 0: table[i][w] = 0 // Base case: no item or zero capacity Else if weights[i - 1] <= w:

// Option 1: include item i-1 // Option 2: exclude item i-1 include = profits[i - 1] + table[i - 1][w - weights[i - 1]] exclude = table[i - 1][w] table[i][w] = max(include, exclude) Else:

// Cannot include item i-1 table[i][w] = table[i - 1][w]

result = table[n][W]

**C code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#define MAX\_ITEMS 1500

#define MAX\_CAPACITY 1500

int dp[MAX\_ITEMS][MAX\_CAPACITY];

int max(int a, int b){

    return (a > b) ? a : b;

}

int knapsackMemo(int W, int weight[], int value[], int n) {

    if (n == 0 || W == 0) {

        return 0;

    }

    if (dp[n][W] != -1) {

        return dp[n][W];

    }

    if (weight[n - 1] > W) {

        return dp[n][W] = knapsackMemo(W, weight, value, n - 1);

    }

    else {

        return dp[n][W] = max(value[n - 1] + knapsackMemo(W - weight[n - 1], weight, value, n - 1),

                             knapsackMemo(W, weight, value, n - 1));

    }

}

int main() {

    int n, W;

    printf("Enter the number of items: ");

    scanf("%d", &n);

    if (n <= 0 || n > MAX\_ITEMS - 1) {

        printf("Invalid number of items.\n");

        return 1;

    }

    int\* weight = (int\*)malloc(n \* sizeof(int));

    int\* value = (int\*)malloc(n \* sizeof(int));

    if (weight == NULL || value == NULL) {

        printf("Memory allocation failed!\n");

        free(weight);

        free(value);

        return 1;

    }

    srand(time(NULL));

    for (int i = 0; i < n; i++) {

        weight[i] = rand() % 20 + 1;

        value[i] = rand() % 20 + 1;

        if (n < 20) {

            printf("Item %d: Weight = %d, Value = %d\n", i + 1, weight[i], value[i]);

        }

    }

    printf("Enter the knapsack capacity: ");

    scanf("%d", &W);

    if (W <= 0 || W > MAX\_CAPACITY - 1) {

        printf("Invalid knapsack capacity.\n");

        free(weight);

        free(value);

        return 1;

    }

    memset(dp, -1, sizeof(dp));

    clock\_t start = clock();

    int result = knapsackMemo(W, weight, value, n);

    clock\_t end = clock();

    printf("\nMaximum value (Memoization) = %d\n", result);

    printf("Time taken: %.6f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

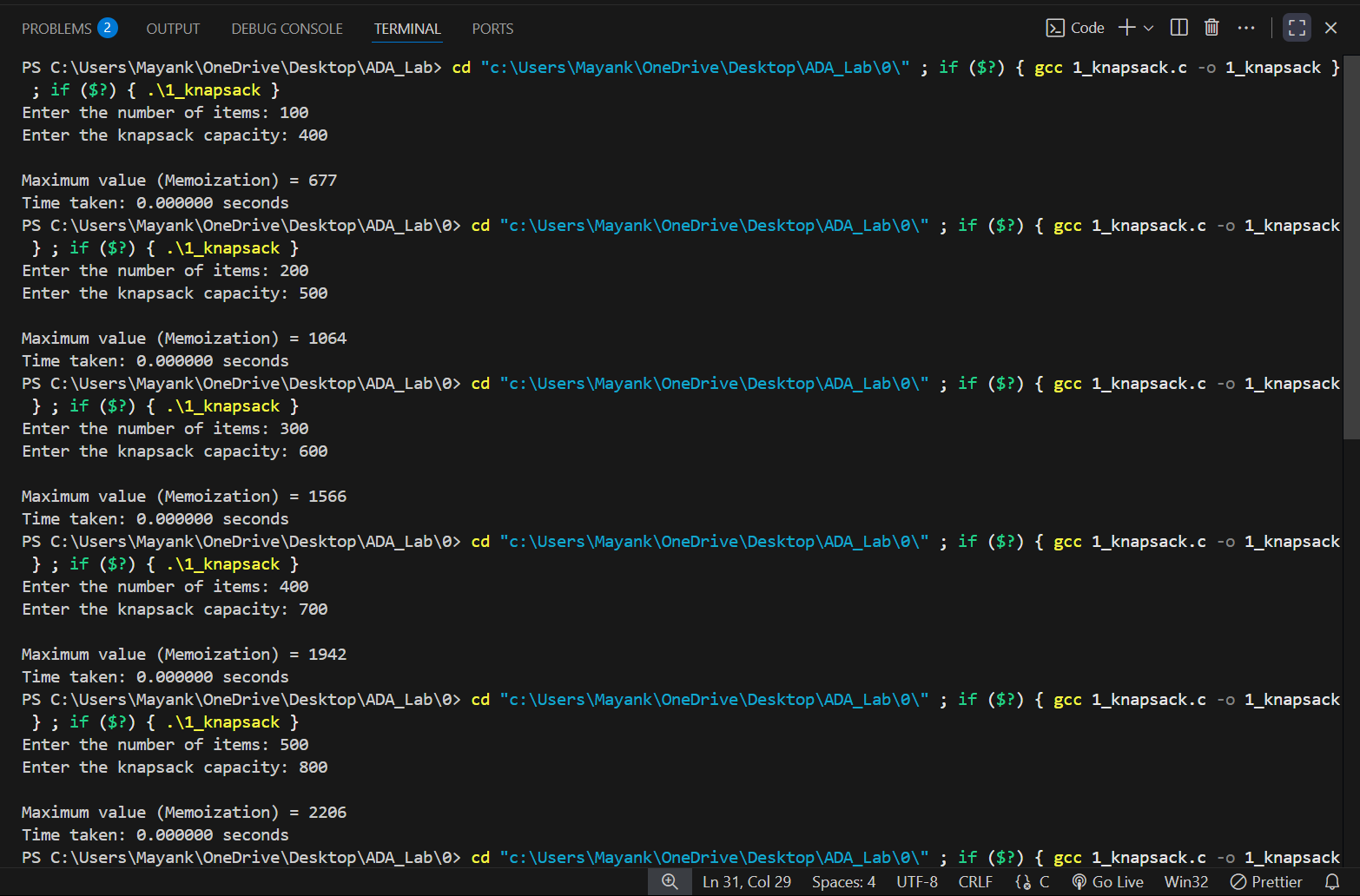
    free(weight);

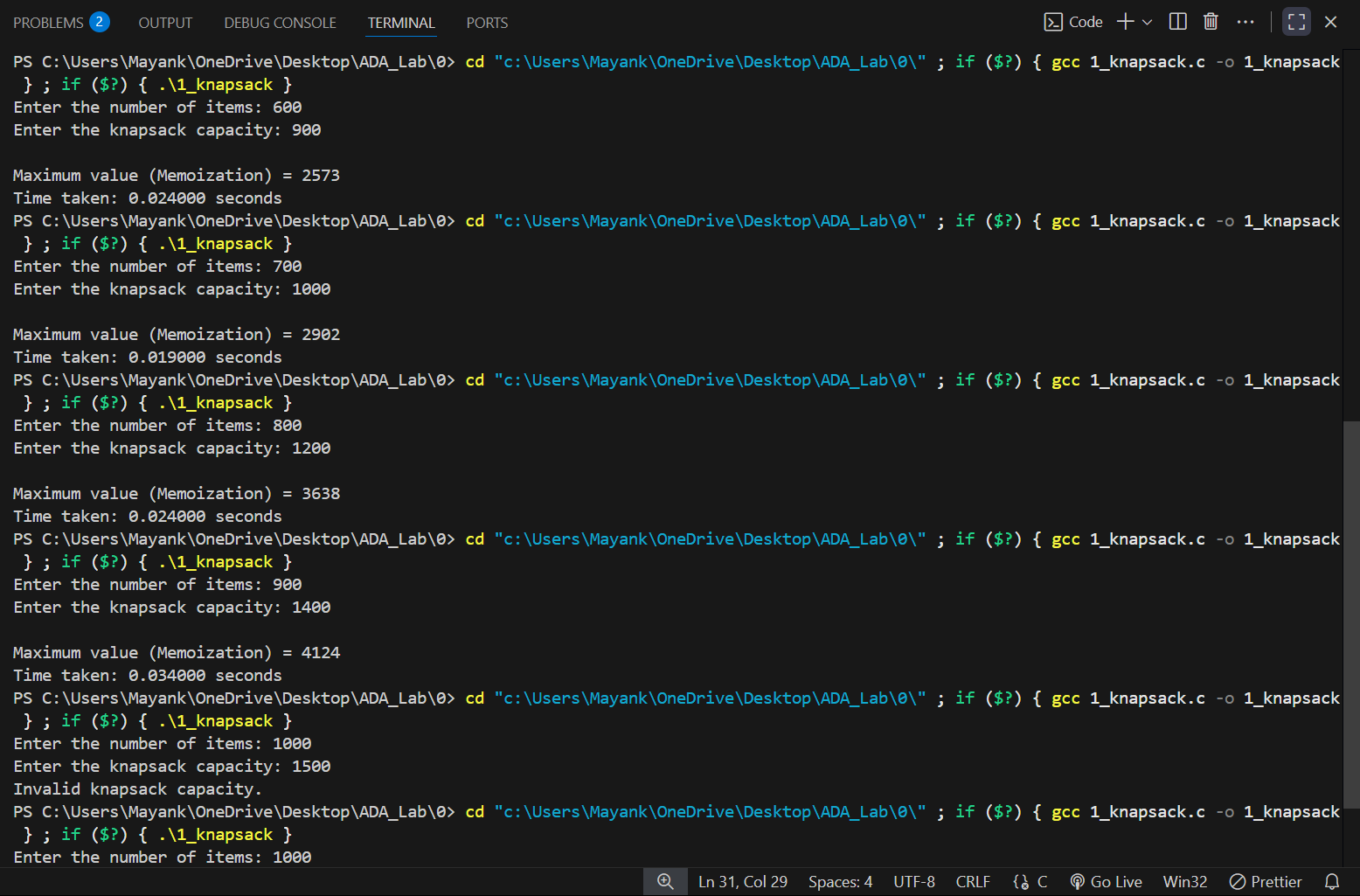
    free(value);

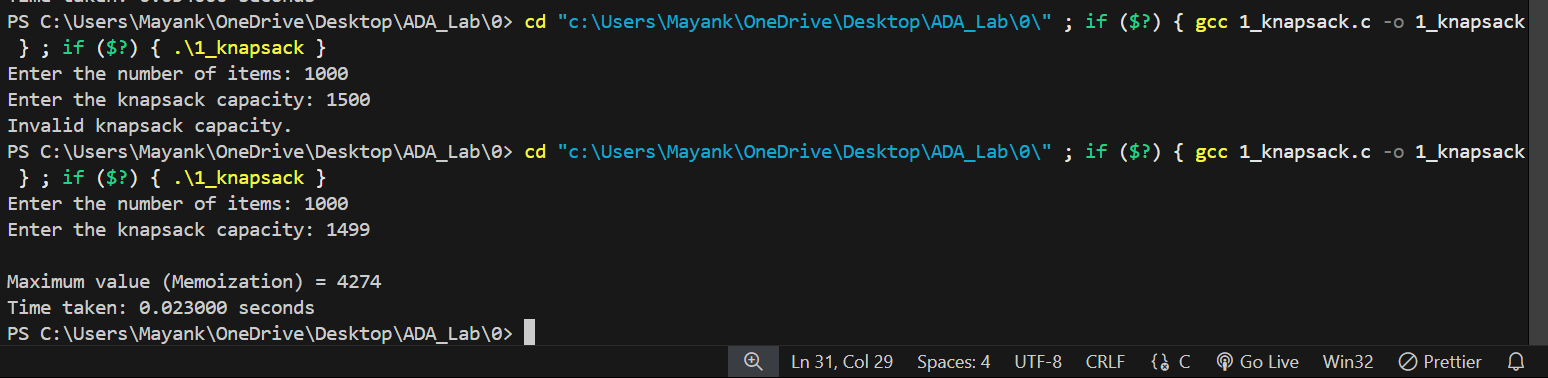
    return 0;

}

**OUTPUT:**







**PYTHON CODE:-**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [100,200,300,400,500,600,700,800,900,1000]

time\_taken = [0.000000 ,0.000000, 0.000000 , 0.000000 , 0.000000 , 0.024000 , 0.019000 , 0.024000 , 0.034000 , 0.023000 ]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='-')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

plt.title("Fibonacci Series: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**

