OPEN ENDED EXPERIMENT

AIM : Help a student select courses for a semester while considering course credits and grades. Apply the 0/1 knapsack algorithm to maximize the GPA while staying within the credit limit.

ALGORITHMS USED :

KnapsackGreedy(int capacity, int weights[], int values[], int n)

```
Step 1 : Sort the items by the value-to-weight ratio in non-increasing order.
Step 2 : Initialize an empty knapsack.
Step 3 : For each item from the highest ratio to the lowest:
Step 4 : If adding the entire item doesn't exceed the knapsack's capacity, add it to the knapsack.
Step 5 : Otherwise, add a fraction of the item to fill the knapsack to its capacity optimally.
Step 6 : The items in the knapsack are the selected items.
```

KnapsackDP(int capacity, int weights[], int values[], int n)

```
Step 1: Create a 2D array dp of size (n+1) x (capacity+1).
Step 2: Initialize the first row and the first column of dp with 0s.
Step 3: for i from 1 to n, do steps 4-8
Step 4:     for w from 0 to capacity, do steps 5-8
Step 5:         if weights[i-1] <= w, then
Step 6:             Set dp[i][w] to the maximum of (values[i-1] + dp[i-1][w - weights[i-1]]) and dp[i-1][w].
Step 7:         else
Step 8:             Set dp[i][w] to dp[i-1][w].
Step 9: Initialize variables maxProfit to 0 and remainingCapacity to capacity.
Step 10: Starting from dp[n][capacity], traverse the dp array to find the selected items.
        a. If dp[i][remainingCapacity] is not equal to dp[i-1][remainingCapacity], include item i and subtract its weight from remainingCapacity.
        b. Move to dp[i-1][remainingCapacity] if dp[i][remainingCapacity] is equal to dp[i-1][remainingCapacity].
Step 11: Return maxProfit and the list of selected items.
```

KnapsackBacktrack(int capacity, int weights[], int values[], int n, int currentIndex)

```
Step 1: If currentIndex is out of bounds (i.e., greater than or equal to n) or the knapsack is full, return 0.
Step 2: If the weight of the item at currentIndex exceeds the remaining capacity, skip to the next item.
Step 3: Calculate the maximum profit by either including or excluding the item at currentIndex:
    a. Include the item:
        - Calculate the profit if the item is included: values[currentIndex] + KnapsackBacktrack(capacity - weights[currentIndex], weights, values, n, currentIndex + 1)
    b. Exclude the item:
        - Calculate the profit if the item is excluded: KnapsackBacktrack(capacity, weights, values, n, currentIndex + 1)
Step 4: Return the maximum profit of the two choices.
```

CONSTRAINTS :

The 0/1 Knapsack Problem has the following constraints:

- Each course has a certain number of credits and a certain GPA.
- The student has a credit limit for the semester.
- The student can only take each course once.

INPUTS :

The input for the 0/1 Knapsack Problem includes:

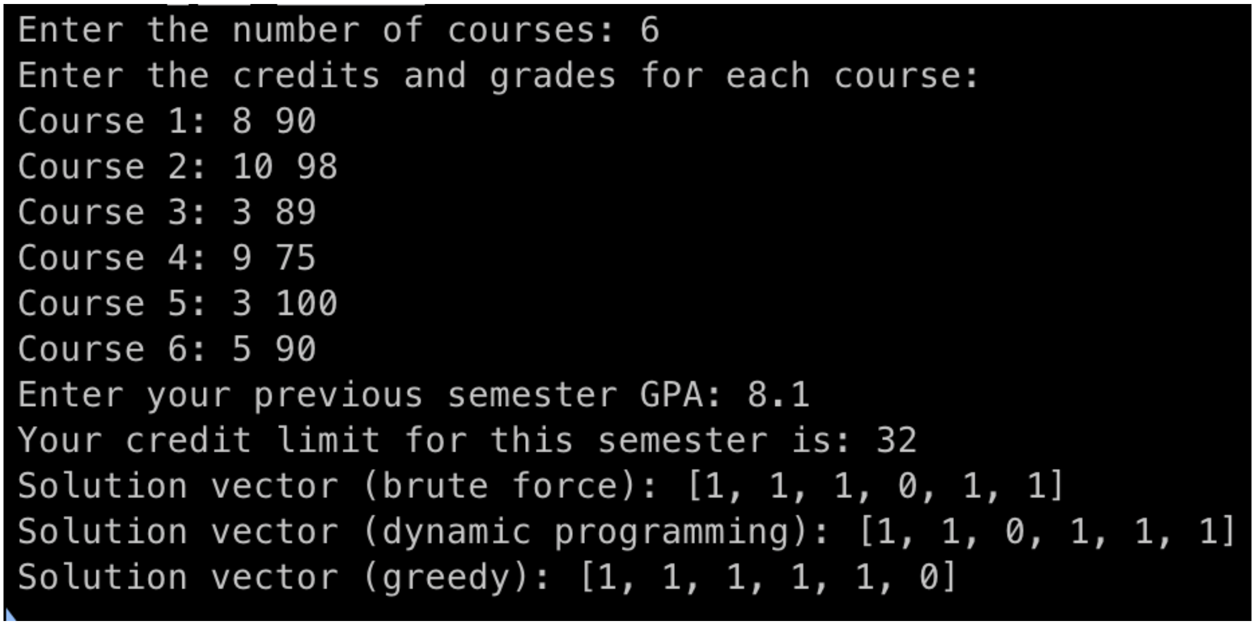
- An array of credits for each course.
- An array of GPAs for each course.
- The number of courses.
- The credit limit for the semester.

CODE :

```
#include <stdio.h>

#define MAX_COURSES 100
#define MAX_CREDITS 1000

void knapsack_brute_force(int credits[], int grades[], int n, int credit_limit) {
   int i, j;
   int max_gpa = 0;
   int max_combination = 0;
```

```
    for (i = 0; i < (1 << n); i++) {
        int total_credits = 0;
        int total_gpa = 0;
        for (j = 0; j < n; j++) {
            if (i & (1 << j)) {
                total_credits += credits[j];
                total_gpa += grades[j];
            }
        }
        if (total_credits <= credit_limit && total_gpa > max_gpa) {
            max_gpa = total_gpa;
            max_combination = i;
        }
    }

    printf("Solution vector (brute force): [");
    for (i = 0; i < n; i++) {
        if (max_combination & (1 << i)) {
            printf("1");
        } else {
            printf("0");
        }
        if (i < n - 1) {
            printf(", ");
        }
    }
    printf("]\n");
}

void knapsack_dynamic_programming(int credits[], int grades[], int n, int credit_limit) {
    int i, j;
    int table[MAX_COURSES + 1][MAX_CREDITS + 1];
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= credit_limit; j++) {
            if (i == 0 || j == 0) {
                table[i][j] = 0;
            } else if (credits[i - 1] <= j) {
                table[i][j] = max(grades[i - 1] + table[i - 1][j - credits[i - 1]], table[i - 1][j]);
            } else {
                table[i][j] = table[i - 1][j];
            }
        }
    }

    printf("Solution vector (dynamic programming): [");
    i = n;
    j = credit_limit;
    while (i > 0 && j > 0) {
        if (table[i][j] != table[i - 1][j]) {
            printf("1");
            j -= credits[i - 1];
        } else {
            printf("0");
        }
        i--;
        if (i > 0 && j > 0) {
            printf(", ");
        }
    }
}

void knapsack_greedy(int credits[], int grades[], int n, int credit_limit) {
    int i, j;
    int max_gpa = 0;
    int solution[MAX_COURSES] = {0};
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (grades[i] < grades[j]) {
                int temp = grades[i];
                grades[i] = grades[j];
                grades[j] = temp;
                temp = credits[i];
                credits[i] = credits[j];
                credits[j] = temp;
            }
        }
    }

    for (i = 0; i < n; i++) {
        if (credits[i] <= credit_limit) {
            solution[i] = 1;
            max_gpa += grades[i];
            credit_limit -= credits[i];
        }
    }

    printf("Solution vector (greedy): [");
    for (i = 0; i < n; i++) {
        printf("%d", solution[i]);
        if (i < n - 1) {
            printf(", ");
        }
    }
    printf("]\n");
}

int main() {
    int credits[MAX_COURSES];
    int grades[MAX_COURSES];
    int n, i;
    printf("Enter the number of courses: ");
    scanf("%d", &n);
    printf("Enter the credits and grades for each course:\n");
    for (i = 0; i < n; i++) {
        printf("Course %d: ", i + 1);
        scanf("%d %d", &credits[i], &grades[i]);
    }
    float prev_gpa;
    int credit_limit;
    printf("Enter your previous semester GPA: ");
    scanf("%f", &prev_gpa);
    credit_limit = (int) (prev_gpa * 4);

    printf("Your credit limit for this semester is: %d\n", credit_limit);
    knapsack_brute_force(credits, grades, n, credit_limit);
    knapsack_dynamic_programming(credits, grades, n, credit_limit);
    knapsack_greedy(credits, grades, n, credit_limit);

    return 0;
}
```

OUTPUT :



```
Enter the number of courses: 6
Enter the credits and grades for each course:
Course 1: 8 90
Course 2: 10 98
Course 3: 3 89
Course 4: 9 75
Course 5: 3 100
Course 6: 5 90
Enter your previous semester GPA: 8.1
Your credit limit for this semester is: 32
Solution vector (brute force): [1, 1, 1, 0, 1, 1]
Solution vector (dynamic programming): [1, 1, 0, 1, 1, 1]
Solution vector (greedy): [1, 1, 1, 1, 1, 0]
```

COMPLEXITY :

1. Brute Force Approach:

   ○ Time Complexity: O(2^n)
   ○ Space Complexity: 0(1)

2. Dynamic Programming Approach:

   ○ Time Complexity: O(n)
   ○ Space Complexity: O(n)

3. Greedy Approach:

   ○ Time Complexity: O(n log n)
   ○ Space Complexity: O(n)