**LAB 4: Implementation of fractional Knapsack**

**NAME :** Vivian Vijay Ludrick

**BRANCH :** SE Comps A

**BATCH : C**

**ROLL NO :** 9914

Knapsack Problem:

CODE:

#include <iostream>

using namespace std;

struct item {

int item\_no; // stores the item number

double ratio; // stores the profit/weight ratio

int weight; // stores the weight of item

int profit; // stores the profit of item

} arr[50];

// Sorting the ratios by modified bubble sort

void sort\_items(item \*Item, int n) // Item of datatype item is created

{

bool exchanged = true;

for (int pass = 1; pass <= n - 1 && exchanged == true; pass++) {

exchanged = false;

for (int i = 1; i <= n - pass; i++) {

if (Item[i].ratio <

Item[i + 1].ratio) // sorting the ratios in descending order

{

swap(Item[i], Item[i + 1]);

exchanged = true;

}

}

}

}

// knapsack function

pair<double, item \*>

fractional\_Knapsack(item \*Items, int max\_capacity, int no\_of\_items,

double \*soln\_vector) // returns the value in pair

{

sort\_items(Items, no\_of\_items); // sorts the Items

double total\_value = 0.0; // total profit

int remaining\_capacity =

max\_capacity; // initially max capacity is the remaining capacity

int i = 1;

while (remaining\_capacity != 0 &&

i <= no\_of\_items) // till capacity gets exhausted and no. of items gets

// exhausted

{

if (Items[i].weight <= remaining\_capacity) // if the weight of item is less

// or equal to remaining capacity

{

soln\_vector[Items[i].item\_no] =

1; // mark the solution vector of that item as 1 as it will be

// completely filled

remaining\_capacity -= Items[i].weight; // subtract the weight of item from

// remaining capacity and update it

} else // if the weight of item is greater than remaining capacity

{

soln\_vector[Items[i].item\_no] =

static\_cast<double>(remaining\_capacity) /

Items[i].weight; // mark the solution vector of that item as

// remaining\_capacity/weight\_of\_item

remaining\_capacity = 0; // make the capacity as 0

}

i++; // increment the i counter

}

cout << "The solution vector is : " << endl;

cout << "[ ";

for (int i = 1; i <= no\_of\_items; i++) // prints the solution vector

{

cout << soln\_vector[i] << " ";

}

cout << "]";

cout << endl << endl;

for (int i = 1; i <= no\_of\_items; i++) // calaculates the total value

{

total\_value += soln\_vector[Items[i].item\_no] \* Items[i].profit;

}

return make\_pair(

total\_value,

Items); // return the total profit and the sorted Items in a pair

}

int main() {

int M, n; // M is used for max capacity and n for no\_of\_items

double x[50] = {0}; // x[] is used to store the solution vector

cout << "Enter the maximum capacity of the knapsack : ";

cin >> M;

cout << "Enter the number of items : ";

cin >> n;

cout << endl;

cout << "Enter the profit and weight of the items respectively : " << endl;

for (int i = 1; i <= n; i++) {

arr[i].item\_no = i;

cout << "Enter for the item " << i << " : ";

cin >> arr[i].profit >> arr[i].weight;

arr[i].ratio = static\_cast<double>(arr[i].profit) /

arr[i].weight; // calculates the profit/weight ratio

}

cout << endl;

auto result = fractional\_Knapsack(arr, M, n, x);

cout << "The sorted ratios are : " << endl;

for (int i = 1; i <= n; i++) // prints the sorted ratios along with items

{

cout << "Item " << arr[i].item\_no << " : " << arr[i].ratio << "\t";

}

cout << endl << endl;

cout << "Total Profit obtained : " << result.first << endl

<< endl; // prints the total profit

cout << "Item Status : " << endl;

for (int i = 1; i <= n; i++) // prints the status of the items

{

if (x[i] == 1)

cout << "Item " << i << " added fully." << endl;

else if (x[i] < 1 && x[i] > 0)

cout << "Item " << i << " is added partially." << endl;

else

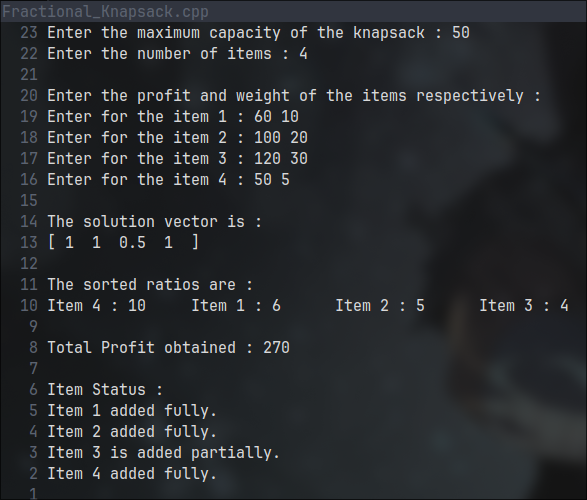
cout << "Item " << i << " is not added." << endl;

}

return 0;

}

OUTPUT:



**POSTLAB:**

Activity Selection:

CODE:

#include <iostream>

#include <vector>

// structure of time taken by activity

typedef struct {

int startTime; // starting time of the activity

int endTime; // ending time of the activity

} Activity;

// inserting the acitivties

void insertAct(Activity arr[], int n) {

for (int i = 0; i < n; i++) {

std::cout << "Enter the start time of activity " << i + 1 << ":\t";

std::cin >> arr[i].startTime;

std::cout << "Enter the end time of activity " << i + 1 << ":\t";

std::cin >> arr[i].endTime;

// valid time check error

while (arr[i].startTime > arr[i].endTime) {

std::cerr << "Error: The start time can't be lesser than the end time.";

std::cout << "Enter the end time of activity " << i + 1 << ":\t";

std::cin >> arr[i].endTime;

}

std::cout << std::endl;

}

}

// printing the activities array

void printAct(Activity arr[], int n) {

for (int i = 0; i < n; i++) {

std::cout << "Start time: " << arr[i].startTime

<< "\t End time: " << arr[i].endTime << std::endl;

}

std::cout << std::endl;

}

void printResult(std::vector<Activity> result) {

for (Activity a : result) {

std::cout << "Start time: " << a.startTime << "\t End time: " << a.endTime

<< std::endl;

}

}

// sorting the element

int \_partition(Activity arr[], int start, int end) {

int randomIndex = rand() % (end - start + 1) + start;

// std::cout << randomIndex << std::endl;

std::swap(arr[start], arr[randomIndex]);

int pivot = arr[start].endTime; // lets save some space here

int pivotPos = start;

for (int i = start + 1; i <= end; i++) {

if (arr[i].endTime <= pivot) {

std::swap(arr[++pivotPos], arr[i]);

}

}

std::swap(arr[start], arr[pivotPos]);

// arr[start] = arr[pivotPos];

// arr[pivotPos] = pivot;

return pivotPos;

}

// sorting the activities based on their end times

void quickSort(Activity arr[], int start, int end) {

if (start >= end) {

return;

}

int p = \_partition(arr, start, end);

quickSort(arr, start, p - 1);

quickSort(arr, p + 1, end);

}

// stores the

void selectActivity(Activity arr[], int n, int maxTime,

std::vector<Activity> result[]) {

result->clear();

if (n == 0) {

return;

}

// since the arr is not empty

result->push\_back(arr[0]);

int lastActivity = 0;

// loop over each array element

for (int i = 1; i < n; i++) {

// check if the startime and endtime can be matched and the endtime is not

// over the limit

if (arr[i].startTime >= arr[lastActivity].endTime &&

arr[i].endTime <= maxTime) {

// add the current activity to your results vector

result->push\_back(arr[i]);

lastActivity = i; // make the current activity as your last activity

}

}

}

// main function

int main(int argc, char \*argv[]) {

int maxTime, n;

std::cout << "Enter the maximum time alloted for the activites:\t";

std::cin >> maxTime;

std::cout << "Enter the number of activities :\t";

std::cin >> n;

Activity arr[n];

std::cout << std::endl;

insertAct(arr, n);

std::cout << "Activities array" << std::endl;

printAct(arr, n);

quickSort(arr, 0, n - 1);

std::cout << "Sorted Activities array" << std::endl;

printAct(arr, n);

std::vector<Activity>

result; // let me get some of the space i saved during pivot back

selectActivity(arr, n, maxTime, &result);

std::cout << "Maximum activities can be executed in the following order"

<< std::endl;

printResult(result);

return 0;

}

OUTPUT: