Assignment-1

Name: Abdullah Al Masud Bhuiyan

ID: 011 221 074

Course: Data Structure and Algorithms II Laboratory

Section: A

Date: 27/12/2023

Divide-and-Conquer Algorithm:

Optimizing Team Performance for UIU Vice Chancellor's Cup T-20 Tournament.

Problem Statement:

UIU is hosting the Vice Chancellor's Cup T-20 Cricket Tournament with 8 participating teams. Syed Abid Hussain Sami, a renowned cricket analyst, has been hired to coach the teams. To assist in his training strategy, you need to develop a program. This program should analyze players' performance data during training sessions to identify each team's most effective continuous training period. This period is pinpointed by finding the subarray with the maximum sum of performance scores, reflecting the phase of highest improvement.

Input:

- N: number of days.
- An array of integers for each team, representing daily performance scores. Positive values indicate performance improvement, while negative values signify less effective training or rest days.

Output:

For each team, the program should provide the starting and ending days of their most effective continuous training period, along with the total improvement score for that period.

Example:

Input	Output
9 -2 1 -3 4 -1 2 1 -5 4	The most effective training period is day 4 to day 7. Total improvement score: 6
7 2 -4 5 4 -1 5 7	The most effective training period is day 3 to day 7. Total improvement score: 20
8 -5 4 8 -3 2 7 3 -6	The most effective training period is day 2 to day 7. Total improvement score: 21

Explanation:

This approach allows Coach Syed Abid Hussain Sami to objectively assess each team's performance over time, identifying specific periods where the training was most effective. This information can be crucial for fine-tuning training strategies and ensuring peak performance during the tournament.

Let's consider the first input,

{-2 1 -3 4 -1 2 1 -5 4}

There's an array of 9 integers, from these 9 we have the subarray {4,-1,2,1} that contains the maximum sum of performance which is 6 and the index starts from 3 to 6 hence the answer is 4 to 7.

Justification:

The divide-and-conquer algorithm is suitable for this problem because it efficiently handles large data sets by breaking them down into smaller subproblems. In the context of analyzing performance scores, this method can quickly and effectively find the continuous subarray (training period) with the maximum sum (optimal performance improvement), even with fluctuating daily scores.

Solution:

```
#include <bits/stdc++.h>
using namespace std;
struct Result
{
    int sum;
    int leftIndex;
    int rightIndex;
};

Result crossingSum(int arr[], int left, int mid, int right)
{

    int sum = arr[mid];
    int LS = arr[mid];
    int indexLS = mid;
    for (int i = mid - 1; i >= left; i--)
    {
        sum = sum + arr[i];
        if (sum > LS)
        {
            LS = sum;
            indexLS = i;
        }
    }
```

```
    sum = arr[mid + 1];
    int RS = arr[mid + 1];
    int indexRS = mid + 1;
    for (int i = mid + 2; i <= right; i++)
    {
        sum = sum + arr[i];
        if (sum > RS)
        {
            RS = sum;
            indexRS = i;
        }
    }

    Result crossing;
    crossing.sum = LS + RS;
    crossing.leftIndex = indexLS;
    crossing.rightIndex = indexRS;
    return crossing;
}
Result maximumSumSubarray(int arr[], int left, int right)
{
    if (left == right)
    {
        Result result;
        result.sum = arr[left];
        result.leftIndex = left;
        result.rightIndex = right;
        return result;
    }
    else
    {
        int mid = (left + right) / 2;

        Result leftSideMaximum = maximumSumSubarray(arr, left, mid);
        Result rightSideMaximum = maximumSumSubarray(arr, mid + 1,
right);
        Result crossingMaximum = crossingSum(arr, left, mid, right);
        if (leftSideMaximum.sum >= rightSideMaximum.sum &&
            leftSideMaximum.sum >= crossingMaximum.sum)
            return leftSideMaximum;
        else if (rightSideMaximum.sum >= leftSideMaximum.sum &&
                 rightSideMaximum.sum >= crossingMaximum.sum)
```

```
            return rightSideMaximum;
        else
            return crossingMaximum;
    }
}
int main()
{
    int N;
    cin >> N;
    int arr[N];
    for (int i = 0; i < N; i++)
    {
        cin >> arr[i];
    }
    Result result = maximumSumSubarray(arr, 0, N - 1);

    cout << "The most effective training period is day " <<
result.leftIndex + 1
         << " to day " << result.rightIndex + 1 << "." << endl;
    cout << "Total improvement score: " << result.sum << endl;

    return 0;
}
```

Greedy Algorithm:

Communication Devices Selection for the Royal Mint Heist.

Problem Statement:

The Royal Mint heist involves a team of skilled individuals, each assigned a crucial role. The success of the operation hinges on effective communication, requiring careful selection of communication devices. The Professor needs to make decisions that balance the need for extensive communication coverage with the limitation of a restricted budget.

In the intricate planning of the Royal Mint heist, The Professor faces the challenge of strategically choosing communication devices for the team. Each device has a unique weight, communication range, and cost associated with it. The objective is to maximize the overall communication range to ensure seamless coordination during the operation, all while staying within a specified budget constraint.

Input:

Device List: N size of the array that contains available communication devices, each characterized by:

- Device name.
- Weight (in kilograms)
- Communication Range (in meters)
- Cost (in monetary units)

Budget Constraint: The maximum amount of money The Professor can allocate for acquiring communication devices.

Output:

Develop a program to determine the combination of communication devices that maximizes the total communication range while adhering to the budget constraint. The output should include the selected devices and the total communication range achieved.

Example:

Input	Output
4 A 2 500 200 B 1 800 150 C 3 300 300 D 0.5 1000 100 500	Selected Devices: D B A Total Communication Range: 2300 Total Cost: 450

5 A 1.5 600 180 B 2 700 250 C 1 500 120 D 0.8 800 200 E 2.5 900 300 700	Selected Devices: C D A Total Communication Range: 1900 Total Cost: 500

Explanation:

Utilize the Knapsack algorithm to iteratively select communication devices with the highest efficiency ratio (communication range per unit cost). The chosen devices ensure the maximum communication range possible within the budget constraints. The success of the Royal Mint heist depends on the seamless coordination facilitated by the selected communication devices.

Justification:

The application of the greedy approach in the Fractional Knapsack algorithm is justified as it enables The Professor to make optimal choices at each step. This approach maximizes communication efficiency within the allocated budget, a critical aspect of the heist's success.

Solution:

```
#include <bits/stdc++.h>
using namespace std;

struct CommunicationDevice
{
    string id;
    double weight;
    double range;
    double cost;
};
bool compare(CommunicationDevice a, CommunicationDevice b)
{
    double result1 = (double)a.range / (double)a.cost;
    double result2 = (double)b.range / (double)b.cost;
    if (result1 > result2)
        return true;
    else
        return false;
}
```

```
void Knapsack(vector<CommunicationDevice> &devices, int budget)
{
    sort(devices.begin(), devices.end(), compare);

    double totalCommunicationRange = 0;
    double totalCost = 0;
    cout << "\nSelected Devices:" << endl;
    for (CommunicationDevice &device : devices)
    {

        if (device.cost <= budget)
        {
            totalCommunicationRange += device.range;
            totalCost += device.cost;

            cout << device.id << endl;

            budget -= device.cost;
        }
    }
    cout << "\nTotal Communication Range: " <<
    totalCommunicationRange << endl;
    cout << "Total Cost: " << totalCost << endl;
}

int main()
{
    int N;
    cin >> N;

    vector<CommunicationDevice> devices(N);
    for (int i = 0; i < N; i++)
    {

        cin >> devices[i].id;
        cin >> devices[i].weight;
        cin >> devices[i].range;
        cin >> devices[i].cost;
    }

    int budget;

    cin >> budget;

    Knapsack(devices, budget);

    return 0;
}
```

Dynamic Programming Algorithm:

Optimal Team Assembly for Project.

Problem Statement:

Mr. Tarek Hasan, a lecturer at UIU, is embarking on a significant and challenging project. To ensure the project's success, he needs to assemble a team of highly efficient individuals. Each potential team member offers a different number of working hours per day, reflecting their availability and commitment to the project. Additionally, each individual requires a certain amount of contractual compensation.

He has a limited budget for hiring his team, making it crucial to select team members strategically. The goal is to maximize the total working hours per day of the team to ensure maximum productivity while staying within the financial constraints of the project's budget.

Input:

- Budget: An integer representing the total budget available for hiring.
- n: An integer representing the number of applicants.
- Nth number of Applicant data (Name, Working hours, Contractual cost).

Output:

- Maximum Working Hours: An integer representing the maximum total working hours per day that can be achieved within the given budget.
- Selected Applicants: A list of names of applicants that should be hired to achieve this maximum.

Example:

Input	Output
10000 5 Masud 8 3000 Limon 6 2000 Musfiq 7 2500 Onik 9 4000 Imran 5 1500	Maximum Working Hours: 27 Selected Applicants: Imran Onik Musfiq Limon
15000 6 A 8 5000 B 2 1500 C 9 7000 D 4 2500 E 3 3000 F 7 5500	Maximum Working Hours: 21 Selected Applicants: D C A

Explanation:

Let's Consider the first example,

Here the maximum budget is 5000. We need to hire developers within the budget. The dynamic programming approach efficiently explores various player combinations and ensures that the optimal solution is acquired.

Here we get the team members-

Imran, Onik, Musfiq, Limon

Total working Hours:(5+9+7+6)=27

Total Cost for Hiring them: (1500+4000+2500+2000)=10000.

Justification:

Optimal Solution: DP ensures an optimal solution by systematically exploring all combinations, making it ideal for this combinatorial problem. It avoids the inefficiency of a brute-force approach.

Efficiency: DP significantly reduces computational complexity compared to considering all possible combinations of applicants, making the solution feasible for larger datasets.

Structured Data Management: Using a struct for applicant data enhances code readability and manageability. It allows for easy modifications and better data organization.

Maximization of Working Hours: The approach guarantees the maximum possible working hours within the budget, a key requirement of the problem.

Practical Application: The solution not only provides the maximum working hours but also identifies which applicants to choose, aligning well with real-world project management scenarios.

Solution:

```
#include <bits/stdc++.h>
using namespace std;

struct Applicant
{
    string name;
    int workingHours;
    int cost;
};

int maximizeWorkingHours(int budget, vector<Applicant> &applicants,
vector<string> &selectedNames, int n)
```

```
{

    vector<vector<int>> dp(n + 1, vector<int>(budget + 1, 0));

    for (int i = 1; i <= n; ++i)
    {
        int hours = applicants[i - 1].workingHours;
        int cost = applicants[i - 1].cost;

        for (int j = 0; j <= budget; ++j)
        {
            if (j >= cost)
            {
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cost] +
hours);
            }
            else
            {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    int remainingBudget = budget;
    for (int i = n; i > 0 && remainingBudget >= 0; --i)
    {
        if (dp[i][remainingBudget] != dp[i - 1][remainingBudget])
        {
            selectedNames.push_back(applicants[i - 1].name);
            remainingBudget -= applicants[i - 1].cost;
        }
    }

    return dp[n][budget];
}

int main()
{
    int budget;
    cin >> budget;
    int n;
    cin >> n;

    vector<Applicant> applicants(n);
    for (int i = 0; i < n; i++)
```

```
 {
        cin >> applicants[i].name;
        cin >> applicants[i].workingHours;
        cin >> applicants[i].cost;
    }

    vector<string> selectedApplicants;

    cout << "Maximum Working Hours: " <<
maximizeWorkingHours(budget, applicants, selectedApplicants, n) <<
endl;
    cout << "Selected Applicants: ";
    for (const auto &name : selectedApplicants)
    {
        cout << name << " ";
    }
    cout << endl;

    return 0;
}
```