Assignment No. 02

- **TITLE –** Design and implement a greedy algorithm to solve a real-world optimization problem, such as minimizing currency notes in a digital wallet (coin change) or maximizing non-overlapping events in a scheduler (activity selection). Represent the logic using pseudo-code and a flowchart. Analyse the algorithm's time complexity (Best, Average, Worst Case) using Big-O, Θ, and Ω notations.

- **THEORY –**

A **greedy algorithm** is an intuitive and straightforward algorithmic paradigm used for optimization problems. The core idea is to build a solution piece by piece, always making the choice that seems best at the current moment without considering future consequences. This "**locally optimal**" choice is made in the hope that it will lead to a "**globally optimal**" solution for the entire problem.

For a greedy algorithm to guarantee an optimal solution, the problem must exhibit two key properties:

1. **Greedy Choice Property:** This means a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, the choice made at each step should be the best possible at that moment and doesn't need to be reconsidered later.
2. **Optimal Substructure:** A problem has optimal substructure if an optimal solution to the entire problem contains optimal solutions to its subproblems. This allows the greedy strategy to work, as each locally optimal choice progressively builds upon the optimal solution of a smaller subproblem.

---

1. Coin Change Problem

- **Greedy Choice:** The greedy choice is to always take the largest coin denomination that is less than or equal to the remaining amount.
- **How it Works:** By repeatedly selecting the largest coin, you reduce the amount as quickly as possible, which seems like the best local decision.
- **Important Caveat:** The greedy approach for coin change is **not always optimal**. It works perfectly for standard, canonical coin systems (like {1, 2, 5, 10}), but can fail for others. For example, with denominations {1, 3, 4} and an amount of 6, the greedy choice would be {4, 1, 1} (3 coins), while the optimal solution is {3, 3} (2 coins).

---

2. Activity Selection Problem

- **Greedy Choice:** The greedy choice is to select the activity that **finishes earliest**.
- **How it Works:** By picking the activity that finishes first, you free up your time resource as early as possible. This maximizes the remaining time available for other potential activities, giving you the best opportunity to fit in more activities overall.
- **Guaranteed Optimality:** Unlike the coin change problem, this specific greedy strategy (sorting by finish time) is **proven** to always yield the globally optimal solution for the Activity Selection Problem.

➢ ALGORITHM –

1. Greedy Coin Change Algorithm

Step 1: Sort the coin denominations in descending order. sort(coins.begin(), coins.end(), greater<int>())
Step 2: Initialize variables to track the remaining amount and the coins used.
Step 3: Iterate through the sorted coin denominations. for i = 0 to coins.length - 1:
Step 4: For each coin, use it as many times as possible. while amount >= coins[i]:
Step 5: Subtract the coin's value from the amount and record that the coin was used. amount = amount - coins[i] count[i]++
Step 6: Repeat until the amount is zero, then return the result.

2. Greedy Activity Selection Algorithm

Step 1: Sort the activities in ascending order based on their finish times. sort(activities, by_finish_time)
Step 2: Select the first activity. select activities[0] last_finish_time = activities[0].finish
Step 3: Iterate through the rest of the activities. for i = 1 to activities.length - 1:
Step 4: Check if the current activity is compatible with the last selected one. if activities[i].start >= last_finish_time:
Step 5: If it is compatible, select the current activity and update the last_finish_time. select activities[i] last_finish_time = activities[i].finish
Step 6: Return the final list of selected activities.

➢ EXAMPLE –

1. Making Change for 93 using Greedy Approach

- **Step 1: Setup**
  - Amount to make: 93
  - Available denominations: {1, 2, 5, 10, 20, 50, 100}
  - Sorted denominations (descending): {100, 50, 20, 10, 5, 2, 1}

- **Step 2: Process Largest Coins**
  - Select 50: 93 >= 50. Use one 50.
    - Remaining amount: 93 - 50 = 43
    - Coins used: 1 x 50
  - Select 20: 43 >= 20. Use two 20s.
    - Remaining amount: 43 - (2 * 20) = 3
    - Coins used: 1 x 50, 2 x 20

- **Step 3: Process Smaller Coins**
  - Select 10: 3 < 10. Cannot use.
  - Select 5: 3 < 5. Cannot use.
  - Select 2: 3 >= 2. Use one 2.
    - Remaining amount: 3 - 2 = 1
    - Coins used: 1 x 50, 2 x 20, 1 x 2
  - Select 1: 1 >= 1. Use one 1.
    - Remaining amount: 1 - 1 = 0
    - Coins used: 1 x 50, 2 x 20, 1 x 2, 1 x 1

- **Step 4: Final Result**
  - The amount is now 0.
  - **Final coin combination: 50 x 1, 20 x 2, 2 x 1, 1 x 1**

2. Maximizing Activities using Greedy Approach

- **Step 1: Setup**
  - Given activities with (start, finish) times:
    - A1(1, 4), A2(3, 5), A3(0, 6), A4(5, 7), A5(8, 9), A6(5, 9)
  - Activities sorted by finish time:
    - A1(1, 4), A2(3, 5), A3(0, 6), A4(5, 7), A5(8, 9), A6(5, 9)

- **Step 2: Initial Selection**
  - Select the first activity, **A1(1, 4)**, because it finishes earliest.
  - last_finish_time = 4
  - Selected = {A1}

- **Step 3: Iterate and Select Compatible Activities**
  - **Check A2(3, 5):** start_time (3) < last_finish_time (4). **Reject.**
  - **Check A3(0, 6):** start_time (0) < last_finish_time (4). **Reject.**
  - **Check A4(5, 7):** start_time (5) >= last_finish_time (4). **Select A4.**
    - last_finish_time = 7
    - Selected = {A1, A4}

- **Step 4: Continue Iteration**
  - **Check A5(8, 9):** start_time (8) >= last_finish_time (7). **Select A5.**
    - last_finish_time = 9
    - Selected = {A1, A4, A5}
  - **Check A6(5, 9):** start_time (5) < last_finish_time (7). **Reject.**

- **Step 5: Final Result**
  - All activities have been considered.
  - **Maximum activities selected: 3**
  - **Final activity set: {A1(1, 4), A4(5, 7), A5(8, 9)}**

➢ CODE –

1. Coin Change Problem

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void coinChange(vector<int> coins, int amount) {
    sort(coins.begin(), coins.end(), greater<int>());
    vector<int> count(coins.size(), 0);
```

```
    for (int i = 0; i < coins.size(); i++) {
        while (amount >= coins[i]) {
            amount -= coins[i];
            count[i]++;
        }
    }

    cout << "Coins used:\n";
    for (int i = 0; i < coins.size(); i++) {
        if (count[i] > 0) {
            cout << coins[i] << " x " << count[i] << endl;
        }
    }
}

int main() {
    int n, amount, result;
    cout << "Enter number of coin denominations: ";
    cin >> n;
    vector<int> coins(n);
    cout << "Enter coin denominations: ";
    for (int i = 0; i < n; i++) {
        cin >> coins[i];
        if (coins[i] <= 0) {
            cout << "Invalid coin!" << endl;
            return 0;
        }
    }
    cout << "Enter amount to make change: ";
    cin >> amount;
    if (amount <= 0) {
        cout << "Invalid amount!" << endl;
        return 0;
    }
    coinChange(coins, amount);
    return 0;
}
```

2. Activity Scheduler

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Activity {
    int start, finish, index;
};

bool activityCompare(Activity s1, Activity s2) {
    return (s1.finish < s2.finish);
}
```

```
vector<Activity> activitySelection(vector<Activity> activities) {
    sort(activities.begin(), activities.end(), activityCompare);
    vector<Activity> selected;
    selected.push_back(activities[0]);
    int lastFinish = activities[0].finish;
    for (int i = 1; i < activities.size(); i++) {
        if (activities[i].start >= lastFinish) {
            selected.push_back(activities[i]);
            lastFinish = activities[i].finish;
        }
    }
    return selected;
}

int main() {
    int n;
    cout << "Enter number of activities: ";
    cin >> n;
    vector<Activity> activities(n);
    cout << "Enter start and finish times of activities:\n";
    for (int i = 0; i < n; i++) {
        cin >> activities[i].start >> activities[i].finish;
        activities[i].index = i + 1;
    }
    vector<Activity> result = activitySelection(activities);
    cout << "\nMaximum number of non-conflicting activities: "<< result.size()
<< endl;
    cout << "Selected activities are:\n";
    for (int i = 0; i < result.size(); i++) {
        cout<<"Activity "<<result[i].index<<" ("<<result[i].start<<",
"<<result[i].finish<<")\n";
    }
    return 0;
}
```

- **OUTPUT –**

1. Coin Change Problem –

```
Enter number of coin denominations: 10
Enter coin denominations: 2000 500 200 100 50 20 10 5 2 1
Enter amount to make change: 1396
Coins used:
500 x 2
200 x 1
100 x 1
50 x 1
20 x 2
5 x 1
1 x 1
```

2. Activity Scheduling Problem –

```
Enter number of activities: 6
Enter start and finish times of activities:
1 3
2 5
4 6
6 7
5 9
8 10

Maximum number of non-conflicting activities: 4
Selected activities are:
Activity 1 (1, 3)
Activity 3 (4, 6)
Activity 4 (6, 7)
Activity 6 (8, 10)
```

- **CONCLUSION –**

  In this practical, we have observed that the greedy approach builds a solution by making the choice that appears best at each immediate step. This strategy is highly efficient and simple to implement, successfully finding the guaranteed optimal solution for well-structured problems like the Activity Selection problem.