Greedy: Practice Fractional Knapsack, Activity Selection problems and their variations. Also practice some basic greedy problems, follow the hint if given.

You may use practice codes or assignment codes given in the assignment as references. this is suggestion from my teache r. now generate me a problem soluton and the test case for these problems . 1st solve the main soution these problem and make variyation each 10 types for every problem .

Based on your teacher's assignment and the attached PDF, I'll provide comprehensive solutions for the main greedy problems and create 10 variations for each. Let me start with the core problems and then expand to variations.

1. Fractional Knapsack Problem

Main Solution

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

struct Item {
    int value, weight;
    double ratio;
    int index;

    Item(int v, int w, int i) : value(v), weight(w), index(i) {
        ratio = (double)value / weight;
    }
};

bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}

double fractionalKnapsack(vector<Item>& items, int capacity) {
    sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;
    int remainingCapacity = capacity;
```

```
    cout << "\nSelected items:\n";
    for (auto& item : items) {
        if (remainingCapacity == 0) break;

        if (item.weight <= remainingCapacity) {
            // Take whole item
            totalValue += item.value;
            remainingCapacity -= item.weight;
            cout << "Item " << item.index << ": " << item.weight
                 << " kg, " << item.value << " taka (100%)\n";
        } else {
            // Take fraction of item
            double fraction = (double)remainingCapacity / item.weight;
            totalValue += item.value * fraction;
            cout << "Item " << item.index << ": " << remainingCapacity
                 << " kg, " << item.value * fraction << " taka ("
                 << fraction * 100 << "%)\n";
            remainingCapacity = 0;
        }
    }

    return totalValue;
}

int main() {
    int n, capacity;
    cout << "Enter number of items: ";
    cin >> n;

    vector<Item> items;
    cout << "Enter weight and value for each item:\n";
    for (int i = 0; i < n; i++) {
        int weight, value;
        cin >> weight >> value;
        items.push_back(Item(value, weight, i + 1));
    }

    cout << "Enter knapsack capacity: ";
    cin >> capacity;

    double maxValue = fractionalKnapsack(items, capacity);
    cout << fixed << setprecision(2);
    cout << "\nMaximum profit: " << maxValue << " taka\n";

    return 0;
}
```

Test Case:

```
Input:
4
4 20
3 9
2 12
```

```
1 7
5

Output:
Selected items:
Item 4: 1 kg, 7 taka (100%)
Item 3: 2 kg, 12 taka (100%)
Item 1: 2 kg, 10 taka (50%)
Maximum profit: 29.00 taka
```

2. Activity Selection Problem

Main Solution

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

struct Activity {
    int start, finish, index;
    Activity(int s, int f, int i) : start(s), finish(f), index(i) {}
};

bool compareByFinish(Activity a, Activity b) {
    return a.finish < b.finish;
}

vector<Activity> activitySelection(vector<Activity>& activities) {
    sort(activities.begin(), activities.end(), compareByFinish);

    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;
    cout << "Enter start and finish time for each activity:\n";
    for (int i = 0; i < n; i++) {
        int start, finish;
```

```
        cin >> start >> finish;
        activities.push_back(Activity(start, finish, i + 1));
    }

    vector<Activity> selected = activitySelection(activities);

    cout << "\nSelected activities:\n";
    for (auto& activity : selected) {
        cout << "Activity " << activity.index << ": ["
             << activity.start << ", " << activity.finish << "]\n";
    }
    cout << "Maximum activities: " << selected.size() << endl;

    return 0;
}
```

Test Case:

```
Input:
6
1 4
3 5
0 6
5 7
3 9
5 9

Output:
Selected activities:
Activity 1: [1, 4]
Activity 2: [3, 5]
Activity 4: [5, 7]
Maximum activities: 3
```

10 Variations of Fractional Knapsack

Variation 1: Multiple Knapsacks

```
// Multiple thieves with different capacity knapsacks
struct Thief {
    int capacity;
    double profit;
    vector<pair<int, double>> takenItems; // item index, fraction
};

vector<Thief> multipleKnapsacks(vector<Item> items, vector<int> capacities) {
    sort(items.begin(), items.end(), compare);
    vector<Thief> thieves(capacities.size());

    for (int i = 0; i < capacities.size(); i++) {
        thieves[i].capacity = capacities[i];
        thieves[i].profit = 0;
```

```
    }

    vector<double> remaining(items.size(), 1.0); // remaining fraction of each item

    for (auto& thief : thieves) {
        int cap = thief.capacity;
        for (int i = 0; i < items.size() && cap > 0; i++) {
            if (remaining[i] > 0) {
                double availableWeight = items[i].weight * remaining[i];
                if (availableWeight <= cap) {
                    // Take all remaining
                    thief.profit += items[i].value * remaining[i];
                    thief.takenItems.push_back({i, remaining[i]});
                    cap -= availableWeight;
                    remaining[i] = 0;
                } else {
                    // Take fraction
                    double fraction = (double)cap / items[i].weight;
                    thief.profit += items[i].value * fraction;
                    thief.takenItems.push_back({i, fraction});
                    remaining[i] -= fraction;
                    cap = 0;
                }
            }
        }
    }

    return thieves;
}
```

Variation 2: Time-Limited Knapsack

```
// Items have both weight and time constraints
struct TimedItem {
    int value, weight, time;
    double ratio;
    int index;

    TimedItem(int v, int w, int t, int i) : value(v), weight(w), time(t), index(i) {
        ratio = (double)value / (weight + time); // Combined efficiency
    }
};

double timedKnapsack(vector<TimedItem>& items, int weightCapacity, int timeLimit) {
    sort(items.begin(), items.end(), [](TimedItem a, TimedItem b) {
        return a.ratio > b.ratio;
    });

    double totalValue = 0.0;
    int remainingWeight = weightCapacity;
    int remainingTime = timeLimit;

    for (auto& item : items) {
        if (remainingWeight == 0 || remainingTime == 0) break;
```

```
        if (item.weight <= remainingWeight && item.time <= remainingTime) {
            totalValue += item.value;
            remainingWeight -= item.weight;
            remainingTime -= item.time;
        } else {
            // Take fraction based on limiting constraint
            double weightFraction = (double)remainingWeight / item.weight;
            double timeFraction = (double)remainingTime / item.time;
            double fraction = min(weightFraction, timeFraction);

            totalValue += item.value * fraction;
            remainingWeight = 0;
            remainingTime = 0;
        }
    }

    return totalValue;
}
```

Variation 3: Penalty-Based Knapsack

```
// Items have penalty if not taken completely
struct PenaltyItem {
    int value, weight, penalty;
    double ratio;
    int index;

    PenaltyItem(int v, int w, int p, int i) : value(v), weight(w), penalty(p), index(i) {
        ratio = (double)(value - penalty) / weight;
    }
};

double penaltyKnapsack(vector<PenaltyItem>& items, int capacity) {
    sort(items.begin(), items.end(), [](PenaltyItem a, PenaltyItem b) {
        return a.ratio > b.ratio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

    for (auto& item : items) {
        if (remainingCapacity == 0) break;

        if (item.weight <= remainingCapacity) {
            totalValue += item.value;
            remainingCapacity -= item.weight;
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            totalValue += item.value * fraction - item.penalty * (1 - fraction);
            remainingCapacity = 0;
        }
    }
```

```
    return totalValue;
}
```

Variation 4: Bounded Knapsack (Limited Quantities)

```
struct BoundedItem {
    int value, weight, quantity;
    double ratio;
    int index;
};

double boundedKnapsack(vector<BoundedItem>& items, int capacity) {
    // Expand items based on quantities
    vector<Item> expandedItems;

    for (auto& item : items) {
        for (int i = 0; i < item.quantity; i++) {
            expandedItems.push_back(Item(item.value, item.weight, item.index));
        }
    }

    return fractionalKnapsack(expandedItems, capacity);
}
```

Variation 5: Category-Constrained Knapsack

```
// Items belong to categories with minimum/maximum limits
struct CategorizedItem {
    int value, weight, category;
    double ratio;
    int index;
};

struct CategoryConstraint {
    int minItems, maxItems;
    int currentCount;
};

double categoryConstrainedKnapsack(vector<CategorizedItem>& items, int capacity,
                                   map<int, CategoryConstraint>& constraints) {
    sort(items.begin(), items.end(), [](CategorizedItem a, CategorizedItem b) {
        return a.ratio > b.ratio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

    for (auto& item : items) {
        auto& constraint = constraints[item.category];

        if (remainingCapacity >= item.weight &&
            constraint.currentCount < constraint.maxItems) {
```

```
            totalValue += item.value;
            remainingCapacity -= item.weight;
            constraint.currentCount++;
        }
    }

    return totalValue;
}
```

Variation 6: Dynamic Weight Knapsack

```
// Item weights change based on what's already in knapsack
struct DynamicItem {
    int baseValue, baseWeight;
    double weightMultiplier; // Weight increases by this factor
    double ratio;
    int index;
};

double dynamicWeightKnapsack(vector<DynamicItem>& items, int capacity) {
    sort(items.begin(), items.end(), [](DynamicItem a, DynamicItem b) {
        return (double)a.baseValue / a.baseWeight > (double)b.baseValue / b.baseWeight;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;
    int itemsAdded = 0;

    for (auto& item : items) {
        int currentWeight = item.baseWeight * (1 + itemsAdded * item.weightMultiplier);

        if (currentWeight <= remainingCapacity) {
            totalValue += item.baseValue;
            remainingCapacity -= currentWeight;
            itemsAdded++;
        } else {
            double fraction = (double)remainingCapacity / currentWeight;
            totalValue += item.baseValue * fraction;
            break;
        }
    }

    return totalValue;
}
```

Variation 7: Priority-Based Knapsack

```
// Items have priority levels affecting selection order
enum Priority { LOW = 1, MEDIUM = 2, HIGH = 3, CRITICAL = 4 };

struct PriorityItem {
    int value, weight;
    Priority priority;
```

```
    double ratio;
    int index;

    double getAdjustedRatio() const {
        return ratio * priority;
    }
};

double priorityKnapsack(vector<PriorityItem>& items, int capacity) {
    sort(items.begin(), items.end(), [](PriorityItem a, PriorityItem b) {
        return a.getAdjustedRatio() > b.getAdjustedRatio();
    });

    return fractionalKnapsack(
        reinterpret_cast<vector<Item>&>(items), capacity);
}
```

Variation 8: Temperature-Sensitive Knapsack

```
// Items degrade over time/temperature
struct TemperatureItem {
    int baseValue, weight;
    double degradationRate; // Value loss per time unit
    double ratio;
    int timeToSpoil;
    int index;
};

double temperatureKnapsack(vector<TemperatureItem>& items, int capacity, int travelTime)
    sort(items.begin(), items.end(), [travelTime](TemperatureItem a, TemperatureItem b) {
        double aValue = max(0.0, a.baseValue - a.degradationRate * travelTime);
        double bValue = max(0.0, b.baseValue - b.degradationRate * travelTime);
        return aValue / a.weight > bValue / b.weight;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

    for (auto& item : items) {
        if (travelTime >= item.timeToSpoil) continue; // Item spoiled

        double currentValue = max(0.0, item.baseValue - item.degradationRate * travelTime

        if (item.weight <= remainingCapacity) {
            totalValue += currentValue;
            remainingCapacity -= item.weight;
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            totalValue += currentValue * fraction;
            break;
        }
    }
```

```
    return totalValue;
}
```

Variation 9: Dependency-Based Knapsack

```
// Some items require other items to be taken first
struct DependentItem {
    int value, weight;
    vector<int> dependencies; // Indices of required items
    double ratio;
    int index;
    bool taken;
};

double dependencyKnapsack(vector<DependentItem>& items, int capacity) {
    sort(items.begin(), items.end(), [](DependentItem a, DependentItem b) {
        return a.ratio > b.ratio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

    function<bool(int)> canTake = [&](int idx) -> bool {
        for (int dep : items[idx].dependencies) {
            if (!items[dep].taken) return false;
        }
        return true;
    };

    bool progress = true;
    while (progress && remainingCapacity > 0) {
        progress = false;
        for (auto& item : items) {
            if (!item.taken && canTake(item.index) && item.weight <= remainingCapacity) {
                totalValue += item.value;
                remainingCapacity -= item.weight;
                item.taken = true;
                progress = true;
            }
        }
    }

    return totalValue;
}
```

Variation 10: Profit-Sharing Knapsack

```
// Multiple people share profits based on contribution
struct SharedItem {
    int value, weight;
    vector<double> ownershipShares; // Sum should be 1.0
    double ratio;
    int index;
```

```
};

struct Person {
    double totalProfit;
    int contribution;
};

vector<Person> profitSharingKnapsack(vector<SharedItem>& items, int capacity, int numPeop
    sort(items.begin(), items.end(), [](SharedItem a, SharedItem b) {
        return a.ratio > b.ratio;
    });

    vector<Person> people(numPeople);
    int remainingCapacity = capacity;

    for (auto& item : items) {
        if (item.weight <= remainingCapacity) {
            for (int i = 0; i < numPeople; i++) {
                people[i].totalProfit += item.value * item.ownershipShares[i];
                people[i].contribution += item.weight * item.ownershipShares[i];
            }
            remainingCapacity -= item.weight;
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            for (int i = 0; i < numPeople; i++) {
                people[i].totalProfit += item.value * item.ownershipShares[i] * fraction;
                people[i].contribution += item.weight * item.ownershipShares[i] * fractio
            }
            break;
        }
    }

    return people;
}
```

10 Variations of Activity Selection

Variation 1: Weighted Activity Selection

```
struct WeightedActivity {
    int start, finish, weight, index;
    WeightedActivity(int s, int f, int w, int i) : start(s), finish(f), weight(w), index(
};

int weightedActivitySelection(vector<WeightedActivity>& activities) {
    sort(activities.begin(), activities.end(), [](WeightedActivity a, WeightedActivity b)
        return a.finish < b.finish;
    });

    int n = activities.size();
    vector<int> dp(n);
    dp[^0] = activities[^0].weight;

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

```
        int latestNonConflicting = -1;
        for (int j = i - 1; j >= 0; j--) {
            if (activities[j].finish <= activities[i].start) {
                latestNonConflicting = j;
                break;
            }
        }

        int including = activities[i].weight;
        if (latestNonConflicting != -1) {
            including += dp[latestNonConflicting];
        }

        dp[i] = max(dp[i-1], including);
    }

    return dp[n-1];
}
```

Variation 2: Activity Selection with Preparation Time

```
struct PrepActivity {
    int start, finish, prepTime, index;
    PrepActivity(int s, int f, int p, int i) : start(s), finish(f), prepTime(p), index(i)
};

vector<PrepActivity> activitySelectionWithPrep(vector<PrepActivity>& activities) {
    sort(activities.begin(), activities.end(), [](PrepActivity a, PrepActivity b) {
        return a.finish < b.finish;
    });

    vector<PrepActivity> selected;
    selected.push_back(activities[^0]);
    int lastFinish = activities[^0].finish;

    for (int i = 1; i < activities.size(); i++) {
        if (activities[i].start - activities[i].prepTime >= lastFinish) {
            selected.push_back(activities[i]);
            lastFinish = activities[i].finish;
        }
    }

    return selected;
}
```

Variation 3: Multi-Resource Activity Selection

```
struct MultiResourceActivity {
    int start, finish;
    vector<int> resourceNeeds; // Resources required
    int index;
};
```

```
vector<MultiResourceActivity> multiResourceSelection(vector<MultiResourceActivity>& activ
                                                     vector<int>& resourceCapacity) {
    sort(activities.begin(), activities.end(), [](MultiResourceActivity a, MultiResourceA
        return a.finish < b.finish;
    });

    vector<MultiResourceActivity> selected;
    vector<vector<pair<int, int>>> resourceSchedule(resourceCapacity.size()); // resource

    for (auto& activity : activities) {
        bool canSchedule = true;

        // Check if resources are available
        for (int r = 0; r < resourceCapacity.size(); r++) {
            if (activity.resourceNeeds[r] > resourceCapacity[r]) {
                canSchedule = false;
                break;
            }

            int usedResource = 0;
            for (auto& slot : resourceSchedule[r]) {
                if (!(activity.finish <= slot.first || activity.start >= slot.second)) {
                    usedResource += activity.resourceNeeds[r];
                }
            }

            if (usedResource + activity.resourceNeeds[r] > resourceCapacity[r]) {
                canSchedule = false;
                break;
            }
        }

        if (canSchedule) {
            selected.push_back(activity);
            for (int r = 0; r < resourceCapacity.size(); r++) {
                if (activity.resourceNeeds[r] > 0) {
                    resourceSchedule[r].push_back({activity.start, activity.finish});
                }
            }
        }
    }

    return selected;
}
```

Variation 4: Priority-Based Activity Selection

```
struct PriorityActivity {
    int start, finish, priority, index;
    PriorityActivity(int s, int f, int p, int i) : start(s), finish(f), priority(p), inde
};

vector<PriorityActivity> priorityActivitySelection(vector<PriorityActivity>& activities)
    // Sort by priority first, then by finish time
    sort(activities.begin(), activities.end(), [](PriorityActivity a, PriorityActivity b)
```

```
        if (a.priority != b.priority) return a.priority > b.priority;
        return a.finish < b.finish;
    });

    vector<PriorityActivity> selected;

    for (auto& activity : activities) {
        bool conflicts = false;
        for (auto& sel : selected) {
            if (!(activity.start >= sel.finish || activity.finish <= sel.start)) {
                conflicts = true;
                break;
            }
        }

        if (!conflicts) {
            selected.push_back(activity);
        }
    }

    return selected;
}
```

Variation 5: Grouped Activity Selection

```
struct GroupedActivity {
    int start, finish, groupId, index;
    GroupedActivity(int s, int f, int g, int i) : start(s), finish(f), groupId(g), index(
};

vector<GroupedActivity> groupedActivitySelection(vector<GroupedActivity>& activities,
                                                 map<int, int>& maxPerGroup) {
    sort(activities.begin(), activities.end(), [](GroupedActivity a, GroupedActivity b) {
        return a.finish < b.finish;
    });

    vector<GroupedActivity> selected;
    map<int, int> groupCount;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish &&
            groupCount[activity.groupId] < maxPerGroup[activity.groupId]) {

            selected.push_back(activity);
            groupCount[activity.groupId]++;
            lastFinish = activity.finish;
        }
    }

    return selected;
}
```

Variation 6: Interval Activity Selection with Gaps

```
struct GapActivity {
    int start, finish, minGapAfter, index;
    GapActivity(int s, int f, int g, int i) : start(s), finish(f), minGapAfter(g), index(
};

vector<GapActivity> gapActivitySelection(vector<GapActivity>& activities) {
    sort(activities.begin(), activities.end(), [](GapActivity a, GapActivity b) {
        return a.finish < b.finish;
    });

    vector<GapActivity> selected;
    selected.push_back(activities[^0]);
    int lastRequiredGap = activities[^0].finish + activities[^0].minGapAfter;

    for (int i = 1; i < activities.size(); i++) {
        if (activities[i].start >= lastRequiredGap) {
            selected.push_back(activities[i]);
            lastRequiredGap = activities[i].finish + activities[i].minGapAfter;
        }
    }

    return selected;
}
```

Variation 7: Conditional Activity Selection

```
struct ConditionalActivity {
    int start, finish, index;
    vector<int> prerequisites; // Must complete these activities first
    vector<int> unlocks;      // Unlocks these activities
    bool completed;

    ConditionalActivity(int s, int f, int i) : start(s), finish(f), index(i), completed(f
};

vector<ConditionalActivity> conditionalActivitySelection(vector<ConditionalActivity>& act
    vector<ConditionalActivity> selected;
    set<int> completedActivities;

    bool progress = true;
    while (progress) {
        progress = false;

        // Sort available activities by finish time
        vector<ConditionalActivity*> available;
        for (auto& activity : activities) {
            if (!activity.completed) {
                bool canDo = true;
                for (int prereq : activity.prerequisites) {
                    if (completedActivities.find(prereq) == completedActivities.end()) {
                        canDo = false;
                        break;
```

```
                    }
                }
                if (canDo) available.push_back(&activity);
            }
        }

        sort(available.begin(), available.end(), [](ConditionalActivity* a, ConditionalAc
            return a->finish < b->finish;
        });

        // Select non-conflicting activities
        int lastFinish = -1;
        for (auto* activity : available) {
            if (activity->start >= lastFinish) {
                selected.push_back(*activity);
                activity->completed = true;
                completedActivities.insert(activity->index);
                lastFinish = activity->finish;
                progress = true;
            }
        }
    }

    return selected;
}
```

Variation 8: Deadline-Constrained Activity Selection

```
struct DeadlineActivity {
    int start, finish, deadline, penalty, index;
    DeadlineActivity(int s, int f, int d, int p, int i) :
        start(s), finish(f), deadline(d), penalty(p), index(i) {}
};

pair<vector<DeadlineActivity>, int> deadlineActivitySelection(vector<DeadlineActivity>& a
    sort(activities.begin(), activities.end(), [](DeadlineActivity a, DeadlineActivity b)
        return a.penalty > b.penalty; // Sort by penalty (descending)
    });

    vector<DeadlineActivity> selected;
    int totalPenalty = 0;

    for (auto& activity : activities) {
        bool conflicts = false;
        for (auto& sel : selected) {
            if (!(activity.start >= sel.finish || activity.finish <= sel.start)) {
                conflicts = true;
                break;
            }
        }

        if (!conflicts && activity.finish <= activity.deadline) {
            selected.push_back(activity);
        } else {
            totalPenalty += activity.penalty;
```

```
        }
    }

    return {selected, totalPenalty};
}
```

Variation 9: Flexible Duration Activity Selection

```
struct FlexibleActivity {
    int start, minDuration, maxDuration, index;
    double valuePerUnit; // Value per time unit

    FlexibleActivity(int s, int minD, int maxD, double v, int i) :
        start(s), minDuration(minD), maxDuration(maxD), valuePerUnit(v), index(i) {}
};

vector<pair<FlexibleActivity, int>> flexibleActivitySelection(vector<FlexibleActivity>& a
                                                              int timeLimit) {
    sort(activities.begin(), activities.end(), [](FlexibleActivity a, FlexibleActivity b)
        return a.valuePerUnit > b.valuePerUnit;
    });

    vector<pair<FlexibleActivity, int>> selected; // activity, actual duration
    int currentTime = 0;

    for (auto& activity : activities) {
        if (currentTime < activity.start) currentTime = activity.start;

        int availableTime = timeLimit - currentTime;
        if (availableTime >= activity.minDuration) {
            int duration = min(activity.maxDuration, availableTime);
            selected.push_back({activity, duration});
            currentTime += duration;
        }

        if (currentTime >= timeLimit) break;
    }

    return selected;
}
```

Variation 10: Collaborative Activity Selection

```
struct CollaborativeActivity {
    int start, finish, requiredPeople, index;
    double efficiency; // Output per person

    CollaborativeActivity(int s, int f, int rp, double e, int i) :
        start(s), finish(f), requiredPeople(rp), efficiency(e), index(i) {}
};

vector<CollaborativeActivity> collaborativeActivitySelection(vector<CollaborativeActivity
                                                             int totalPeople) {
```

```
    sort(activities.begin(), activities.end(), [](CollaborativeActivity a, CollaborativeA
        return a.efficiency > b.efficiency;
    });

    vector<CollaborativeActivity> selected;
    vector<pair<int, int>> peopleSchedule; // (start, finish) when people are busy

    for (auto& activity : activities) {
        int availablePeople = totalPeople;

        // Count people busy during this activity
        for (auto& schedule : peopleSchedule) {
            if (!(activity.start >= schedule.second || activity.finish <= schedule.first)
                availablePeople -= activity.requiredPeople;
            }
        }

        if (availablePeople >= activity.requiredPeople) {
            selected.push_back(activity);
            peopleSchedule.push_back({activity.start, activity.finish});
        }
    }

    return selected;
}
```

Additional Basic Greedy Problems

Coin Change Problem

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

void coinChange(int amount, vector<int>& coins) {
    sort(coins.rbegin(), coins.rend()); // Sort in descending order

    vector<int> result;
    int totalCoins = 0;

    for (int coin : coins) {
        while (amount >= coin) {
            result.push_back(coin);
            amount -= coin;
            totalCoins++;
        }
    }

    if (amount > 0) {
        cout << "Cannot make exact change\n";
        return;
    }
```

```
    cout << "Coins used:\n";
    map<int, int> coinCount;
    for (int coin : result) {
        coinCount[coin]++;
    }

    for (auto& p : coinCount) {
        cout << p.first << " cents --- " << p.second << "\n";
    }
    cout << "Total " << totalCoins << " coins\n";
}

int main() {
    int amount;
    cout << "Enter amount: ";
    cin >> amount;

    vector<int> coins = {25, 10, 5, 1}; // quarters, dimes, nickels, pennies
    coinChange(amount, coins);

    return 0;
}
```

Test Case:

```
Input: 173
Output:
25 cents --- 6
10 cents --- 2
1 cents --- 3
Total 11 coins
```

Minimum Stops Problem

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

vector<int> minimumStops(int distance, int tankCapacity, vector<int>& stations) {
    vector<int> stops;
    int currentPos = 0;
    int fuel = tankCapacity;
    int stationIndex = 0;

    while (currentPos + fuel < distance) {
        int farthestReach = currentPos + fuel;
        int nextStop = -1;

        // Find the farthest station within reach
        while (stationIndex < stations.size() && stations[stationIndex] <= farthestReach)
            nextStop = stationIndex;
            stationIndex++;
        }
```

```
        if (nextStop == -1) {
            cout << "Cannot reach destination\n";
            return {};
        }

        stops.push_back(nextStop);
        fuel = tankCapacity - (stations[nextStop] - currentPos) + fuel;
        currentPos = stations[nextStop];
    }

    return stops;
}

int main() {
    int distance, capacity, n;
    cout << "Enter distance, tank capacity, number of stations: ";
    cin >> distance >> capacity >> n;

    vector<int> stations(n);
    cout << "Enter station positions: ";
    for (int i = 0; i < n; i++) {
        cin >> stations[i];
    }

    vector<int> stops = minimumStops(distance, capacity, stations);

    if (!stops.empty()) {
        cout << "Stops needed:\n";
        for (int stop : stops) {
            cout << "Stop at station " << stop + 1 << " at position " << stations[stop] <
        }
    }

    return 0;
}
```

These solutions provide comprehensive coverage of greedy algorithms with practical implementations. Each variation demonstrates different aspects of greedy problem-solving, from basic optimization to complex constraint handling. The test cases help verify correctness and understand the expected input/output format.

⁂

1. 04.-Greedy-algorithm.pdf