make moree 30 variation

I'll provide **30 additional variations** for greedy algorithm problems, expanding beyond the 20 I already covered. Here are 15 more **Fractional Knapsack variations** and 15 more **Activity Selection variations**:

Additional Fractional Knapsack Variations (11-25)

Variation 11: Environmental Impact Knapsack

```
struct EcoItem {
    int value, weight, carbonFootprint, recycleValue;
    double ecoRatio; // (value + recycleValue) / (weight + carbonFootprint)
    int index;
};

double ecoKnapsack(vector<EcoItem>& items, int capacity, int maxCarbon) {
    sort(items.begin(), items.end(), [](EcoItem a, EcoItem b) {
        return a.ecoRatio > b.ecoRatio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;
    int remainingCarbon = maxCarbon;

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

        double weightFraction = (double)remainingCapacity / item.weight;
        double carbonFraction = (double)remainingCarbon / item.carbonFootprint;
        double fraction = min({1.0, weightFraction, carbonFraction});

        totalValue += (item.value + item.recycleValue) * fraction;
        remainingCapacity -= item.weight * fraction;
        remainingCarbon -= item.carbonFootprint * fraction;
    }

    return totalValue;
}
```

Variation 12: Seasonal Knapsack

```
struct SeasonalItem {
    int baseValue, weight;
    map<string, double> seasonMultiplier; // season -> multiplier
    double ratio;
    int index;
};

double seasonalKnapsack(vector<SeasonalItem>& items, int capacity, string currentSeason)
    for (auto& item : items) {
        int adjustedValue = item.baseValue * item.seasonMultiplier[currentSeason];
        item.ratio = (double)adjustedValue / item.weight;
    }

    sort(items.begin(), items.end(), [](SeasonalItem a, SeasonalItem b) {
        return a.ratio > b.ratio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

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

        int seasonalValue = item.baseValue * item.seasonMultiplier[currentSeason];

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

    return totalValue;
}
```

Variation 13: Risk-Adjusted Knapsack

```
struct RiskItem {
    int expectedValue, weight;
    double riskFactor; // 0.0 to 1.0 (0 = no risk, 1 = total loss possible)
    double confidenceInterval;
    double riskAdjustedRatio;
    int index;

    void calculateRiskRatio() {
        double adjustedValue = expectedValue * (1.0 - riskFactor * 0.5); // Risk penalty
        riskAdjustedRatio = adjustedValue / weight;
    }
};
```

```
double riskAdjustedKnapsack(vector<RiskItem>& items, int capacity, double riskTolerance)
    for (auto& item : items) {
        if (item.riskFactor > riskTolerance) continue; // Skip high-risk items
        item.calculateRiskRatio();
    }

    // Remove items exceeding risk tolerance
    items.erase(remove_if(items.begin(), items.end(),
        [riskTolerance](RiskItem& item) { return item.riskFactor > riskTolerance; }),
        items.end());

    sort(items.begin(), items.end(), [](RiskItem a, RiskItem b) {
        return a.riskAdjustedRatio > b.riskAdjustedRatio;
    });

    double totalExpectedValue = 0.0;
    int remainingCapacity = capacity;

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

        if (item.weight <= remainingCapacity) {
            totalExpectedValue += item.expectedValue * (1.0 - item.riskFactor * 0.3);
            remainingCapacity -= item.weight;
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            totalExpectedValue += item.expectedValue * fraction * (1.0 - item.riskFactor
            break;
        }
    }

    return totalExpectedValue;
}
```

Variation 14: Multi-Currency Knapsack

```
struct CurrencyItem {
    map<string, int> values; // currency -> value
    int weight;
    map<string, double> ratios;
    int index;

    void calculateRatios() {
        for (auto& [currency, value] : values) {
            ratios[currency] = (double)value / weight;
        }
    }
};

map<string, double> multiCurrencyKnapsack(vector<CurrencyItem>& items, int capacity,
                                          map<string, double>& exchangeRates,
                                          string baseCurrency = "USD") {
    for (auto& item : items) {
        item.calculateRatios();
    }

    // Convert all values to base currency for comparison
    sort(items.begin(), items.end(), [&](CurrencyItem a, CurrencyItem b) {
        double aBaseValue = 0, bBaseValue = 0;
        for (auto& [currency, value] : a.values) {
            aBaseValue += value * exchangeRates[currency];
        }
        for (auto& [currency, value] : b.values) {
            bBaseValue += value * exchangeRates[currency];
        }
        return aBaseValue / a.weight > bBaseValue / b.weight;
    });

    map<string, double> totalValues;
    int remainingCapacity = capacity;

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

        if (item.weight <= remainingCapacity) {
            for (auto& [currency, value] : item.values) {
                totalValues[currency] += value;
            }
            remainingCapacity -= item.weight;
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            for (auto& [currency, value] : item.values) {
                totalValues[currency] += value * fraction;
            }
            break;
        }
    }

    return totalValues;
}
```

Variation 15: Energy-Efficiency Knapsack

```
struct EnergyItem {
    int value, weight, energyConsumption, energyProduction;
    double netEnergyRatio;
    int index;

    void calculateNetRatio() {
        int netEnergy = energyProduction - energyConsumption;
        netEnergyRatio = (double)(value + netEnergy * 10) / weight; // Energy worth 10 ur
    }
};

double energyEfficientKnapsack(vector<EnergyItem>& items, int capacity, int energyBudget)
    for (auto& item : items) {
        item.calculateNetRatio();
    }
    sort(items.begin(), items.end(), [](EnergyItem a, EnergyItem b) {
        return a.netEnergyRatio > b.netEnergyRatio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;
    int remainingEnergy = energyBudget;

    for (auto& item : items) {
        if (remainingCapacity == 0 || remainingEnergy < item.energyConsumption) continue;

        if (item.weight <= remainingCapacity) {
            totalValue += item.value;
            remainingCapacity -= item.weight;
            remainingEnergy += (item.energyProduction - item.energyConsumption);
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            totalValue += item.value * fraction;
            remainingEnergy += (item.energyProduction - item.energyConsumption) * fractio
            break;
        }
    }

    return totalValue;
}
```

Variation 16: Location-Based Knapsack

```
struct LocationItem {
    int value, weight;
    pair<double, double> location; // (latitude, longitude)
    double transportCost;
    double distanceAdjustedRatio;
    int index;

    void calculateDistanceRatio(pair<double, double> destination) {
        double distance = sqrt(pow(location.first - destination.first, 2) +
                              pow(location.second - destination.second, 2));
        transportCost = distance * 0.1; // Cost per unit distance
        double adjustedValue = value - transportCost;
        distanceAdjustedRatio = max(0.0, adjustedValue) / weight;
    }
};

double locationBasedKnapsack(vector<LocationItem>& items, int capacity,
                           pair<double, double> destination) {
    for (auto& item : items) {
        item.calculateDistanceRatio(destination);
    }

    sort(items.begin(), items.end(), [](LocationItem a, LocationItem b) {
        return a.distanceAdjustedRatio > b.distanceAdjustedRatio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

    for (auto& item : items) {
        if (remainingCapacity == 0 || item.distanceAdjustedRatio <= 0) break;

        double netValue = item.value - item.transportCost;
        if (netValue <= 0) continue;

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

    return totalValue;
}
```

Variation 17: Quality-Decay Knapsack

```
struct DecayItem {
    int initialValue, weight;
    double decayRate; // Value loss per time unit
    int shelfLife;
    double ratio;
    int index;

    int getCurrentValue(int currentTime) {
        if (currentTime >= shelfLife) return 0;
        return max(0, (int)(initialValue * exp(-decayRate * currentTime)));
    }
};

double qualityDecayKnapsack(vector<DecayItem>& items, int capacity, int currentTime, int
    for (auto& item : items) {
        int futureValue = item.getCurrentValue(futureTime);
        if (futureValue > 0) {
            item.ratio = (double)futureValue / item.weight;
        } else {
            item.ratio = 0;
        }
    }

    sort(items.begin(), items.end(), [](DecayItem a, DecayItem b) {
        return a.ratio > b.ratio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

    for (auto& item : items) {
        if (remainingCapacity == 0 || item.ratio <= 0) break;
        int futureValue = item.getCurrentValue(futureTime);
        if (futureValue <= 0) continue;

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

    return totalValue;
}
```

Variation 18: Social Impact Knapsack

```
struct SocialItem {
    int economicValue, weight;
    int socialImpactScore; // People helped
    int environmentalScore; // Positive environmental impact
    double holisticRatio;
    int index;

    void calculateHolisticRatio() {
        double totalValue = economicValue + socialImpactScore * 5 + environmentalScore *
        holisticRatio = totalValue / weight;
    }
};

double socialImpactKnapsack(vector<SocialItem>& items, int capacity,
                          double socialWeight = 0.4, double envWeight = 0.3, double econW
    for (auto& item : items) {
        double totalValue = item.economicValue * econWeight +
                           item.socialImpactScore * socialWeight * 10 +
                           item.environmentalScore * envWeight * 8;
        item.holisticRatio = totalValue / item.weight;
    }

    sort(items.begin(), items.end(), [](SocialItem a, SocialItem b) {
        return a.holisticRatio > b.holisticRatio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;

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

        if (item.weight <= remainingCapacity) {
            totalValue += item.economicValue + item.socialImpactScore * 5 + item.environm
            remainingCapacity -= item.weight;
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            totalValue += (item.economicValue + item.socialImpactScore * 5 + item.enviror
            break;
        }
    }

    return totalValue;
}
```

Variation 19: Subscription-Based Knapsack

```
struct SubscriptionItem {
    int monthlyValue, setupWeight, monthlyWeight;
    int subscriptionLength; // months
    double totalRatio;
    int index;
    bool isSubscription;

    void calculateSubscriptionRatio() {
        int totalValue = monthlyValue * subscriptionLength;
        int totalWeight = setupWeight + monthlyWeight * subscriptionLength;
        totalRatio = (double)totalValue / totalWeight;
    }
};

double subscriptionKnapsack(vector<SubscriptionItem>& items, int initialCapacity,
                          int monthlyCapacity, int planningHorizon) {
    for (auto& item : items) {
        if (item.isSubscription) {
            item.calculateSubscriptionRatio();
        } else {
            item.totalRatio = (double)item.monthlyValue / item.setupWeight;
        }
    }

    sort(items.begin(), items.end(), [](SubscriptionItem a, SubscriptionItem b) {
        return a.totalRatio > b.totalRatio;
    });

    double totalValue = 0.0;
    int remainingInitialCapacity = initialCapacity;
    int remainingMonthlyCapacity = monthlyCapacity;

    for (auto& item : items) {
        if (item.isSubscription) {
            if (item.setupWeight <= remainingInitialCapacity &&
                item.monthlyWeight <= remainingMonthlyCapacity) {
                totalValue += item.monthlyValue * min(item.subscriptionLength, planningHo
                remainingInitialCapacity -= item.setupWeight;
                remainingMonthlyCapacity -= item.monthlyWeight;
            }
        } else {
            if (item.setupWeight <= remainingInitialCapacity) {
                totalValue += item.monthlyValue;
                remainingInitialCapacity -= item.setupWeight;
            }
        }
    }

    return totalValue;
}
```

Variation 20: Skill-Requirement Knapsack

```
struct SkillItem {
    int value, weight;
    map<string, int> requiredSkills; // skill -> minimum level
    map<string, int> gainedSkills;   // skill -> gained level
    double skillAdjustedRatio;
    int index;
    bool canUse;
};

double skillBasedKnapsack(vector<SkillItem>& items, int capacity,
                         map<string, int>& currentSkills) {
    for (auto& item : items) {
        item.canUse = true;
        for (auto& [skill, required] : item.requiredSkills) {
            if (currentSkills[skill] < required) {
                item.canUse = false;
                break;
            }
        }

        if (item.canUse) {
            int skillBonus = 0;
            for (auto& [skill, gained] : item.gainedSkills) {
                skillBonus += gained * 2; // Each skill level worth 2 points
            }
            item.skillAdjustedRatio = (double)(item.value + skillBonus) / item.weight;
        } else {
            item.skillAdjustedRatio = 0;
        }
    }

    sort(items.begin(), items.end(), [](SkillItem a, SkillItem b) {
        return a.skillAdjustedRatio > b.skillAdjustedRatio;
    });

    double totalValue = 0.0;
    int remainingCapacity = capacity;
    map<string, int> skillsGained = currentSkills;

    for (auto& item : items) {
        if (remainingCapacity == 0 || !item.canUse) continue;

        // Recheck if item can be used with gained skills
        bool canUseNow = true;
        for (auto& [skill, required] : item.requiredSkills) {
            if (skillsGained[skill] < required) {
                canUseNow = false;
                break;
            }
        }

        if (!canUseNow) continue;

        if (item.weight <= remainingCapacity) {
            totalValue += item.value;
            remainingCapacity -= item.weight;

            // Update gained skills
            for (auto& [skill, gained] : item.gainedSkills) {
                skillsGained[skill] += gained;
            }
        }
    }

    return totalValue;
}
```

Variation 21: Network-Effect Knapsack

```
struct NetworkItem {
    int baseValue, weight;
    double networkMultiplier; // Value increases with more items
    vector<int> synergisticItems; // Items that boost each other
    double currentRatio;
    int index;
    bool selected;
};

double networkEffectKnapsack(vector<NetworkItem>& items, int capacity) {
    int iteration = 0;
    vector<NetworkItem> selected;
    int remainingCapacity = capacity;

    while (remainingCapacity > 0 && iteration < items.size()) {
        // Recalculate ratios based on current network
        for (auto& item : items) {
            if (item.selected) continue;

            int networkBonus = 0;
            int synergisticCount = 0;

            for (auto& sel : selected) {
                networkBonus += item.baseValue * item.networkMultiplier;

                if (find(item.synergisticItems.begin(), item.synergisticItems.end(),
                        sel.index) != item.synergisticItems.end()) {
                    synergisticCount++;
                }
            }

            int totalValue = item.baseValue + networkBonus + synergisticCount * 5;
            item.currentRatio = (double)totalValue / item.weight;
        }

        // Find best item
        auto bestItem = max_element(items.begin(), items.end(),
            [](NetworkItem a, NetworkItem b) {
                if (a.selected && !b.selected) return true;
                if (!a.selected && b.selected) return false;
                return a.currentRatio < b.currentRatio;
            });

        if (bestItem != items.end() && !bestItem->selected &&
            bestItem->weight <= remainingCapacity) {
            bestItem->selected = true;
            selected.push_back(*bestItem);
            remainingCapacity -= bestItem->weight;
        } else {
            break;
        }

        iteration++;
    }

    // Calculate final value with all network effects
    double totalValue = 0;
    for (auto& item : selected) {
        int networkBonus = 0;
        int synergisticCount = 0;

        for (auto& other : selected) {
            if (other.index != item.index) {
                networkBonus += item.baseValue * item.networkMultiplier;

                if (find(item.synergisticItems.begin(), item.synergisticItems.end(),
                        other.index) != item.synergisticItems.end()) {
                    synergisticCount++;
                }
            }
        }

        totalValue += item.baseValue + networkBonus + synergisticCount * 5;
    }

    return totalValue;
}
```

Variation 22: Insurance-Protected Knapsack

```
struct InsuredItem {
    int value, weight, insuranceCost;
    double lossRisk; // Probability of losing item
    bool hasInsurance;
    double expectedValue;
    double ratio;
    int index;
    void calculateExpectedValue() {
        if (hasInsurance) {
            expectedValue = value - insuranceCost; // Guaranteed value minus cost
        } else {
            expectedValue = value * (1.0 - lossRisk); // Risk-adjusted value
        }
        ratio = expectedValue / weight;
    }
};

double insuranceKnapsack(vector<InsuredItem>& items, int capacity, double riskTolerance)
    for (auto& item : items) {
        // Decide whether to buy insurance
        double valuWithInsurance = item.value - item.insuranceCost;
        double expectedWithoutInsurance = item.value * (1.0 - item.lossRisk);

        if (item.lossRisk > riskTolerance || valuWithInsurance > expectedWithoutInsurance
            item.hasInsurance = true;
        } else {
            item.hasInsurance = false;
        }

        item.calculateExpectedValue();
    }

    sort(items.begin(), items.end(), [](InsuredItem a, InsuredItem b) {
        return a.ratio > b.ratio;
    });

    double totalExpectedValue = 0.0;
    int remainingCapacity = capacity;

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

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

    return totalExpectedValue;
}
```

Variation 23: Load-Balancing Knapsack

```
struct LoadItem {
    int value, weight;
    string category;
    int processingTime;
    double efficiency;
    int index;
};

struct LoadBalancer {
    map<string, int> categoryLimits;
    map<string, int> currentLoad;
    int totalCapacity;
    int remainingCapacity;
};

double loadBalancedKnapsack(vector<LoadItem>& items, LoadBalancer& balancer) {
    for (auto& item : items) {
        item.efficiency = (double)item.value / (item.weight + item.processingTime * 0.1);
    }

    sort(items.begin(), items.end(), [](LoadItem a, LoadItem b) {
        return a.efficiency > b.efficiency;
    });

    double totalValue = 0.0;

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

        // Check category limit
        if (balancer.currentLoad[item.category] >= balancer.categoryLimits[item.category]
            continue;
        }

        if (item.weight <= balancer.remainingCapacity) {
            totalValue += item.value;
            balancer.remainingCapacity -= item.weight;
            balancer.currentLoad[item.category]++;
        } else {
            // Can take fraction if it doesn't violate load balancing
            double fraction = (double)balancer.remainingCapacity / item.weight;
            totalValue += item.value * fraction;
            balancer.remainingCapacity = 0;
        }
    }

    return totalValue;
}
```

Variation 24: Maintenance-Cost Knapsack

```
struct MaintenanceItem {
    int initialValue, weight;
    int maintenanceCostPerPeriod;
    int maintenanceFrequency; // Every N periods
    int lifespan; // Total periods before replacement
    double netPresentValue;
    double ratio;
    int index;

    void calculateNPV(double discountRate, int planningHorizon) {
        netPresentValue = initialValue;

        for (int period = 1; period <= min(lifespan, planningHorizon); period++) {
            if (period % maintenanceFrequency == 0) {
                double discountedCost = maintenanceCostPerPeriod / pow(1 + discountRate,
                netPresentValue -= discountedCost;
            }
        }

        ratio = netPresentValue / weight;
    }
};

double maintenanceKnapsack(vector<MaintenanceItem>& items, int capacity,
                         double discountRate, int planningHorizon) {
    for (auto& item : items) {
        item.calculateNPV(discountRate, planningHorizon);
    }

    sort(items.begin(), items.end(), [](MaintenanceItem a, MaintenanceItem b) {
        return a.ratio > b.ratio;
    });

    double totalNPV = 0.0;
    int remainingCapacity = capacity;

    for (auto& item : items) {
        if (remainingCapacity == 0 || item.netPresentValue <= 0) break;

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

    return totalNPV;
}
```

Variation 25: Collaborative Filtering Knapsack

```
struct CollaborativeItem {
    int value, weight;
    vector<double> userRatings;
    map<int, double> similarUserWeights; // userID -> similarity weight
    double predictedValue;
    double ratio;
    int index;

    void calculatePredictedValue(int targetUser, map<int, vector<double>>& userProfiles)
        double weightedSum = 0, totalWeight = 0;

        for (auto& [userID, similarity] : similarUserWeights) {
            if (userID < userRatings.size()) {
                weightedSum += userRatings[userID] * similarity;
                totalWeight += similarity;
            }
        }

        if (totalWeight > 0) {
            predictedValue = value * (weightedSum / totalWeight);
        } else {
            predictedValue = value * 0.5; // Default prediction
        }

        ratio = predictedValue / weight;
    }
};

double collaborativeKnapsack(vector<CollaborativeItem>& items, int capacity,
                           int targetUser, map<int, vector<double>>& userProfiles) {
    for (auto& item : items) {
        item.calculatePredictedValue(targetUser, userProfiles);
    }

    sort(items.begin(), items.end(), [](CollaborativeItem a, CollaborativeItem 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.predictedValue;
            remainingCapacity -= item.weight;
        } else {
            double fraction = (double)remainingCapacity / item.weight;
            totalValue += item.predictedValue * fraction;
            break;
        }
    }
    return totalValue;
}
```

Additional Activity Selection Variations (11-25)

Variation 11: Temperature-Dependent Activity Selection

```
struct TemperatureActivity {
    int start, finish, index;
    int minTemp, maxTemp; // Activity can only be done in this temperature range
    int comfortTemp;      // Optimal temperature for activity
    double efficiency;    // Performance multiplier based on temperature

    double calculateEfficiency(int currentTemp) {
        if (currentTemp < minTemp || currentTemp > maxTemp) return 0;

        int tempDiff = abs(currentTemp - comfortTemp);
        efficiency = max(0.1, 1.0 - tempDiff * 0.02); // 2% efficiency loss per degree
        return efficiency;
    }
};

vector<TemperatureActivity> temperatureActivitySelection(vector<TemperatureActivity>& act
                                                        vector<int>& temperatureForecast)
    // Calculate efficiency for each activity based on temperature during its time
    for (auto& activity : activities) {
        double totalEfficiency = 0;
        int count = 0;

        for (int time = activity.start; time < activity.finish && time < temperatureForec
            totalEfficiency += activity.calculateEfficiency(temperatureForecast[time]);
            count++;
        }

        if (count > 0) {
            activity.efficiency = totalEfficiency / count;
        } else {
            activity.efficiency = 0;
        }
    }

    // Sort by efficiency-adjusted finish time
    sort(activities.begin(), activities.end(), [](TemperatureActivity a, TemperatureActiv
        return a.finish * (2.0 - a.efficiency) < b.finish * (2.0 - b.efficiency);
    });

    vector<TemperatureActivity> selected;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.efficiency > 0.3 && activity.start >= lastFinish) { // Minimum effic
            selected.push_back(activity);
            lastFinish = activity.finish;
        }
    }

    return selected;
}
```

Variation 12: Energy-Level Activity Selection

```
struct EnergyActivity {
    int start, finish, index;
    int energyRequired, energyRestored;
    int energyLevel; // 1-5 scale
    bool isResting;

    EnergyActivity(int s, int f, int req, int rest, int level, int i) :
        start(s), finish(f), energyRequired(req), energyRestored(rest),
        energyLevel(level), index(i), isResting(rest > req) {}
};

vector<EnergyActivity> energyAwareActivitySelection(vector<EnergyActivity>& activities,
                                                     int initialEnergy, int maxEnergy) {
    sort(activities.begin(), activities.end(), [](EnergyActivity a, EnergyActivity b) {
        return a.finish < b.finish;
    });

    vector<EnergyActivity> selected;
    int currentEnergy = initialEnergy;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish) {
            if (activity.isResting || currentEnergy >= activity.energyRequired) {
                selected.push_back(activity);
                currentEnergy = min(maxEnergy,
                    max(0, currentEnergy - activity.energyRequired + activity.energyResto
                lastFinish = activity.finish;
            }
        }
    }

    return selected;
}
```

Variation 13: Social-Distance Activity Selection

```
struct SocialActivity {
    int start, finish, index;
    int maxParticipants;
    int currentParticipants;
    double socialDistanceRequired; // meters
    pair<double, double> location;

    bool canAccommodate(int additionalPeople, double availableSpace) {
        int totalPeople = currentParticipants + additionalPeople;
        double requiredSpace = totalPeople * (socialDistanceRequired * socialDistanceRequ
        return requiredSpace <= availableSpace && totalPeople <= maxParticipants;
    }
};

vector<SocialActivity> socialDistanceActivitySelection(vector<SocialActivity>& activities
                                                     double venueSpace, int groupSize) {
    sort(activities.begin(), activities.end(), [](SocialActivity a, SocialActivity b) {
        return a.finish < b.finish;
    });

    vector<SocialActivity> selected;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish && activity.canAccommodate(groupSize, venueSpace
            selected.push_back(activity);
            lastFinish = activity.finish;
        }
    }

    return selected;
}
```

Variation 14: Weather-Dependent Activity Selection

```
struct WeatherActivity {
    int start, finish, index;
    set<string> acceptableWeather; // {"sunny", "cloudy", "rainy", "windy"}
    map<string, double> performanceModifier;
    string activityType; // "indoor", "outdoor", "flexible"
};

vector<WeatherActivity> weatherActivitySelection(vector<WeatherActivity>& activities,
                                                 map<int, string>& weatherForecast) {
    // Filter activities based on weather compatibility
    vector<WeatherActivity> feasible;

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

        for (int time = activity.start; time < activity.finish; time++) {
            if (weatherForecast.count(time) &&
                activity.acceptableWeather.find(weatherForecast[time]) == activity.accept
                weatherCompatible = false;
                break;
            }
        }

        if (weatherCompatible) {
            feasible.push_back(activity);
        }
    }

    sort(feasible.begin(), feasible.end(), [](WeatherActivity a, WeatherActivity b) {
        return a.finish < b.finish;
    });

    vector<WeatherActivity> selected;
    int lastFinish = -1;

    for (auto& activity : feasible) {
        if (activity.start >= lastFinish) {
            selected.push_back(activity);
            lastFinish = activity.finish;
        }
    }

    return selected;
}
```

Variation 15: Learning-Curve Activity Selection

```
struct LearningActivity {
    int start, finish, index;
    string skillCategory;
    int skillLevel; // 1-10
    int learningGain;
    map<string, int> prerequisites; // skill -> required level
    bool completed;
};

vector<LearningActivity> learningCurveActivitySelection(vector<LearningActivity>& activit
                                                       map<string, int>& currentSkills) {
    vector<LearningActivity> selected;
    map<string, int> skillLevels = currentSkills;

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

        // Find available activities that can be done with current skills
        vector<LearningActivity*> available;
        for (auto& activity : activities) {
            if (activity.completed) continue;

            bool canDo = true;
            for (auto& [skill, required] : activity.prerequisites) {
                if (skillLevels[skill] < required) {
                    canDo = false;
                    break;
                }
            }

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

        // Sort by finish time
        sort(available.begin(), available.end(), [](LearningActivity* a, LearningActivity
            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;
                skillLevels[activity->skillCategory] += activity->learningGain;
                lastFinish = activity->finish;
                progress = true;
            }
        }
    }

    return selected;
}
```

Variation 16: Multi-Location Activity Selection

```
struct LocationActivity {
    int start, finish, index;
    pair<double, double> location; // (x, y) coordinates
    int travelTime; // Time needed to reach from origin
    string venue;

    int calculateTravelTime(pair<double, double> from) {
        double distance = sqrt(pow(location.first - from.first, 2) +
                              pow(location.second - from.second, 2));
        return (int)(distance * 0.5); // 0.5 time units per distance unit
    }
};

vector<LocationActivity> multiLocationActivitySelection(vector<LocationActivity>& activit
                                                         pair<double, double> startLocatior
    sort(activities.begin(), activities.end(), [](LocationActivity a, LocationActivity b)
        return a.finish < b.finish;
    });

    vector<LocationActivity> selected;
    pair<double, double> currentLocation = startLocation;
    int lastFinish = -1;

    for (auto& activity : activities) {
        int travelTime = activity.calculateTravelTime(currentLocation);

        // Check if we can reach the activity in time
        if (activity.start >= lastFinish + travelTime) {
            selected.push_back(activity);
            currentLocation = activity.location;
            lastFinish = activity.finish;
        }
    }
    return selected;
}
```

Variation 17: Team-Size Activity Selection

```
struct TeamActivity {
    int start, finish, index;
    int minTeamSize, maxTeamSize, optimalTeamSize;
    double efficiency; // Based on actual team size vs optimal
    int currentTeamSize;

    double calculateEfficiency(int teamSize) {
        if (teamSize < minTeamSize || teamSize > maxTeamSize) return 0;

        if (teamSize == optimalTeamSize) return 1.0;

        int deviation = abs(teamSize - optimalTeamSize);
        return max(0.1, 1.0 - deviation * 0.1); // 10% efficiency loss per person deviati
    }
};

vector<TeamActivity> teamSizeActivitySelection(vector<TeamActivity>& activities,
                                             int availableTeamMembers) {
    // Calculate efficiency for each activity
    for (auto& activity : activities) {
        activity.efficiency = activity.calculateEfficiency(availableTeamMembers);
    }

    // Sort by efficiency-weighted finish time (prefer high efficiency, early finish)
    sort(activities.begin(), activities.end(), [](TeamActivity a, TeamActivity b) {
        double aScore = a.finish / max(0.1, a.efficiency);
        double bScore = b.finish / max(0.1, b.efficiency);
        return aScore < bScore;
    });

    vector<TeamActivity> selected;
    int remainingMembers = availableTeamMembers;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.efficiency > 0.5 && // Minimum efficiency threshold
            activity.start >= lastFinish &&
            activity.minTeamSize <= remainingMembers) {

            int assignedMembers = min(activity.maxTeamSize, remainingMembers);
            activity.currentTeamSize = assignedMembers;
            activity.efficiency = activity.calculateEfficiency(assignedMembers);

            selected.push_back(activity);
            remainingMembers -= assignedMembers;
            lastFinish = activity.finish;
        }
    }
    return selected;
}
```

Variation 18: Stress-Level Activity Selection

```
struct StressActivity {
    int start, finish, index;
    int stressLevel; // 1-10 (1 = relaxing, 10 = very stressful)
    int stressRelief; // Negative stress added (recovery activities)
    bool isRecovery;

    StressActivity(int s, int f, int stress, int relief, int i) :
        start(s), finish(f), stressLevel(stress), stressRelief(relief), index(i) {
        isRecovery = (relief > stress);
    }
};

vector<StressActivity> stressAwareActivitySelection(vector<StressActivity>& activities,
                                                    int maxStressLevel, int initialStress =
    sort(activities.begin(), activities.end(), [](StressActivity a, StressActivity b) {
        return a.finish < b.finish;
    });

    vector<StressActivity> selected;
    int currentStress = initialStress;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish) {
            int newStressLevel = currentStress + activity.stressLevel - activity.stressRe

            if (newStressLevel <= maxStressLevel || activity.isRecovery) {
                selected.push_back(activity);
                currentStress = max(0, newStressLevel);
                lastFinish = activity.finish;
            }
        }
    }

    return selected;
}
```

Variation 19: Budget-Constrained Activity Selection

```
struct BudgetActivity {
    int start, finish, index;
    double cost, revenue;
    double netValue; // revenue - cost
    double roi; // return on investment
    string category;

    void calculateMetrics() {
        netValue = revenue - cost;
        roi = cost > 0 ? revenue / cost : (revenue > 0 ? INFINITY : 0);
    }
};

vector<BudgetActivity> budgetActivitySelection(vector<BudgetActivity>& activities,
                                              double totalBudget,
                                              map<string, double>& categoryBudgets) {
    for (auto& activity : activities) {
        activity.calculateMetrics();
    }

    // Sort by ROI descending, then by finish time ascending
    sort(activities.begin(), activities.end(), [](BudgetActivity a, BudgetActivity b) {
        if (abs(a.roi - b.roi) > 1e-9) return a.roi > b.roi;
        return a.finish < b.finish;
    });

    vector<BudgetActivity> selected;
    double remainingBudget = totalBudget;
    map<string, double> remainingCategoryBudgets = categoryBudgets;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish &&
            activity.cost <= remainingBudget &&
            activity.cost <= remainingCategoryBudgets[activity.category]) {

            selected.push_back(activity);
            remainingBudget -= activity.cost;
            remainingCategoryBudgets[activity.category] -= activity.cost;
            lastFinish = activity.finish;
        }
    }

    return selected;
}
```

Variation 20: Skill-Building Activity Selection

```
struct SkillBuildingActivity {
    int start, finish, index;
    map<string, int> skillsRequired; // skill -> minimum level needed
    map<string, int> skillsGained;   // skill -> levels gained
    int difficulty;
    double learningEfficiency;

    double calculateLearningValue(map<string, int>& currentSkills,
                                 map<string, double>& skillValues) {
        double totalValue = 0;

        // Check if activity is feasible
        for (auto& [skill, required] : skillsRequired) {
            if (currentSkills[skill] < required) return -1; // Not feasible
        }
        // Calculate value of skills gained
        for (auto& [skill, gained] : skillsGained) {
            totalValue += gained * skillValues[skill];
        }

        return totalValue / (finish - start); // Value per time unit
    }
};

vector<SkillBuildingActivity> skillBuildingActivitySelection(vector<SkillBuildingActivity
                                                             map<string, int>& initialSkil
                                                             map<string, double>& skillValu
    vector<SkillBuildingActivity> selected;
    map<string, int> currentSkills = initialSkills;

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

        // Calculate learning value for each activity
        vector<pair<double, int>> activityValues;
        for (int i = 0; i < activities.size(); i++) {
            double value = activities[i].calculateLearningValue(currentSkills, skillValue
            if (value >= 0) {
                activityValues.push_back({value, i});
            }
        }

        // Sort by learning value per time unit
        sort(activityValues.begin(), activityValues.end(), greater<pair<double, int>>());

        // Select activities by finish time among high-value ones
        vector<int> candidates;
        for (auto& [value, idx] : activityValues) {
            candidates.push_back(idx);
            if (candidates.size() >= min(10, (int)activityValues.size())) break; // Top 1
        }

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

        // Standard activity selection on candidates
        int lastFinish = -1;
        if (!selected.empty()) {
            lastFinish = selected.back().finish;
        }

        for (int idx : candidates) {
            auto& activity = activities[idx];
            if (activity.start >= lastFinish) {
                selected.push_back(activity);

                // Update skills
                for (auto& [skill, gained] : activity.skillsGained) {
                    currentSkills[skill] += gained;
                }

                lastFinish = activity.finish;
                progress = true;
                break; // Take one activity per iteration
            }
        }
    }

    return selected;
}
```

Variation 21: Seasonal Activity Selection

```
struct SeasonalActivity {
    int start, finish, index;
    string season; // "spring", "summer", "autumn", "winter"
    map<string, double> seasonalMultiplier; // season -> efficiency multiplier
    double baseValue;
    bool weatherDependent;

    double getSeasonalValue(string currentSeason) {
        if (seasonalMultiplier.count(currentSeason)) {
            return baseValue * seasonalMultiplier[currentSeason];
        }
        return weatherDependent ? baseValue * 0.5 : baseValue; // Reduced value if weathe
    }
};

vector<SeasonalActivity> seasonalActivitySelection(vector<SeasonalActivity>& activities,
                                                  string currentSeason,
                                                  map<int, string>& seasonCalendar) {
    // Calculate seasonal values
    for (auto& activity : activities) {
        string activitySeason = seasonCalendar.count(activity.start) ?
                               seasonCalendar[activity.start] : currentSeason;
        activity.baseValue = activity.getSeasonalValue(activitySeason);
    }

    // Sort by seasonal value (descending) then by finish time (ascending)
    sort(activities.begin(), activities.end(), [](SeasonalActivity a, SeasonalActivity b)
        if (abs(a.baseValue - b.baseValue) > 1e-9) return a.baseValue > b.baseValue;
        return a.finish < b.finish;
    });

    vector<SeasonalActivity> selected;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish && activity.baseValue > 0) {
            selected.push_back(activity);
            lastFinish = activity.finish;
        }
    }
    return selected;
}
```

Variation 22: Crowd-Size Activity Selection

```
struct CrowdActivity {
    int start, finish, index;
    int expectedCrowd;
    int optimalCrowdSize;
    double crowdTolerance; // How much deviation from optimal is acceptable
    double satisfactionLevel;

    double calculateSatisfaction(int actualCrowd) {
        if (actualCrowd == 0) return 0;

        double deviation = abs(actualCrowd - optimalCrowdSize) / (double)optimalCrowdSize
        if (deviation <= crowdTolerance) {
            return 1.0;
        }

        return max(0.0, 1.0 - (deviation - crowdTolerance));
    }
};

vector<CrowdActivity> crowdAwareActivitySelection(vector<CrowdActivity>& activities,
                                                  map<int, int>& crowdForecast) {
    // Calculate satisfaction for each activity
    for (auto& activity : activities) {
        int avgCrowd = 0, count = 0;

        for (int time = activity.start; time < activity.finish; time++) {
            if (crowdForecast.count(time)) {
                avgCrowd += crowdForecast[time];
                count++;
            }
        }

        if (count > 0) {
            avgCrowd /= count;
            activity.satisfactionLevel = activity.calculateSatisfaction(avgCrowd);
        } else {
            activity.satisfactionLevel = activity.calculateSatisfaction(activity.expectec
        }
    }

    // Sort by satisfaction level (descending) then finish time (ascending)
    sort(activities.begin(), activities.end(), [](CrowdActivity a, CrowdActivity b) {
        if (abs(a.satisfactionLevel - b.satisfactionLevel) > 1e-9) {
            return a.satisfactionLevel > b.satisfactionLevel;
        }
        return a.finish < b.finish;
    });

    vector<CrowdActivity> selected;
    int lastFinish = -1;
    for (auto& activity : activities) {
        if (activity.start >= lastFinish && activity.satisfactionLevel > 0.3) { // Minimu
            selected.push_back(activity);
            lastFinish = activity.finish;
        }
    }

    return selected;
}
```

Variation 23: Health-Impact Activity Selection

```
struct HealthActivity {
    int start, finish, index;
    int physicalImpact;    // Positive = improves health, Negative = strains health
    int mentalImpact;      // Positive = reduces stress, Negative = increases stress
    int requiredFitness;   // Minimum fitness level required
    bool isRecovery;       // Recovery activities restore health

    double calculateHealthScore() {
        return physicalImpact * 0.6 + mentalImpact * 0.4;
    }
};

vector<HealthActivity> healthActivitySelection(vector<HealthActivity>& activities,
                                             int currentFitness, int currentStress,
                                             int maxStress) {
    sort(activities.begin(), activities.end(), [](HealthActivity a, HealthActivity b) {
        return a.finish < b.finish;
    });

    vector<HealthActivity> selected;
    int stress = currentStress;
    int fitness = currentFitness;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish && fitness >= activity.requiredFitness) {
            int newStress = stress - activity.mentalImpact;

            if (newStress <= maxStress || activity.isRecovery) {
                selected.push_back(activity);
                stress = max(0, newStress);
                fitness += activity.physicalImpact;
                lastFinish = activity.finish;
            }
        }
    }

    return selected;
}
```

Variation 24: Transportation-Mode Activity Selection

```
struct TransportActivity {
    int start, finish, index;
    pair<double, double> location;
    map<string, int> transportTime; // mode -> time needed
    map<string, double> transportCost; // mode -> cost
    string preferredTransport;
    bool accessibleByPublicTransport;
};

vector<TransportActivity> transportModeActivitySelection(vector<TransportActivity>& activ
                                                        pair<double, double> startLocatior
                                                        map<string, bool>& availableTransp
                                                        double budgetLimit) {
    sort(activities.begin(), activities.end(), [](TransportActivity a, TransportActivity
        return a.finish < b.finish;
    });

    vector<TransportActivity> selected;
    pair<double, double> currentLocation = startLocation;
    double remainingBudget = budgetLimit;
    int lastFinish = -1;

    for (auto& activity : activities) {
        // Find best available transport mode
        string bestMode = "";
        int minTime = INT_MAX;
        double cost = 0;

        for (auto& [mode, time] : activity.transportTime) {
            if (availableTransport[mode] && time < minTime &&
                activity.transportCost[mode] <= remainingBudget) {
                bestMode = mode;
                minTime = time;
                cost = activity.transportCost[mode];
            }
        }

        if (!bestMode.empty() && activity.start >= lastFinish + minTime) {
            selected.push_back(activity);
            currentLocation = activity.location;
            remainingBudget -= cost;
            lastFinish = activity.finish;
        }
    }

    return selected;
}
```

Variation 25: Social-Network Activity Selection

```
struct NetworkActivity {
    int start, finish, index;
    set<int> requiredFriends;     // Friend IDs that must attend
    set<int> optionalFriends;     // Friend IDs that could attend
    map<int, double> friendInfluence; // Friend ID -> influence weight
    double networkValue;
    int minAttendees, maxAttendees;

    double calculateNetworkValue(set<int>& availableFriends) {
        networkValue = 0;
        int attendees = 1; // Self

        // Count required friends
        for (int friendId : requiredFriends) {
            if (availableFriends.count(friendId)) {
                attendees++;
                networkValue += friendInfluence[friendId];
            } else {
                return -1; // Cannot do activity without required friends
            }
        }

        // Count optional friends (up to max capacity)
        for (int friendId : optionalFriends) {
            if (attendees >= maxAttendees) break;
            if (availableFriends.count(friendId)) {
                attendees++;
                networkValue += friendInfluence[friendId] * 0.8; // Less value for optior
            }
        }

        if (attendees < minAttendees) return -1; // Not enough people

        // Network effect: value increases with more connections
        networkValue *= sqrt(attendees);
        return networkValue;
    }
};

vector<NetworkActivity> socialNetworkActivitySelection(vector<NetworkActivity>& activitie
                                                      map<int, set<int>>& friendAvailabil
    sort(activities.begin(), activities.end(), [](NetworkActivity a, NetworkActivity b) {
        return a.finish < b.finish;
    });

    vector<NetworkActivity> selected;
    int lastFinish = -1;

    for (auto& activity : activities) {
        if (activity.start >= lastFinish) {
            // Find available friends during this time slot
            set<int> availableFriends;
            for (auto& [friendId, availability] : friendAvailability) {
                bool available = true;
                for (int time = activity.start; time < activity.finish; time++) {
                    if (availability.count(time) == 0) { // Friend is busy
                        available = false;
                        break;
                    }
                }
                if (available) {
                    availableFriends.insert(friendId);
                }
            }

            double value = activity.calculateNetworkValue(availableFriends);
            if (value > 0) {
                selected.push_back(activity);
                lastFinish = activity.finish;

                // Update friend availability (mark them as busy)
                for (int friendId : activity.requiredFriends) {
                    if (friendAvailability.count(friendId)) {
                        for (int time = activity.start; time < activity.finish; time++)
                            friendAvailability[friendId].erase(time);
                        }
                    }
                }

                int attendees = 1;
                for (int friendId : activity.optionalFriends) {
                    if (attendees >= activity.maxAttendees) break;
                    if (availableFriends.count(friendId)) {
                        attendees++;
                        for (int time = activity.start; time < activity.finish; time++)
                            friendAvailability[friendId].erase(time);
                        }
                    }
                }
            }
        }
    }

    return selected;
}
```

These 30 additional variations cover advanced scenarios including **environmental constraints**, **risk management**, **multi-currency optimization**, **network effects**, **seasonal dependencies**, **health impacts**, **social dynamics**, **transportation logistics**, and **collaborative filtering**. Each variation demonstrates different aspects of greedy algorithm design and real-world problem-solving complexity, providing comprehensive practice for competitive programming and algorithm design skills.

⁂

1. https://www.geeksforgeeks.org/dsa/fractional-knapsack-problem/
2. https://i.cs.hku.hk/~hkual/Notes/Greedy/Fractional knapsack.html

3. https://cp-algorithms.com/dynamic_programming/knapsack.html
4. https://www.educative.io/answers/what-is-the-fractional-knapsack-problem
5. https://fiveable.me/introduction-algorithms/unit-11/knapsack-problem-variations/study-guide/ijVYVStOF5PtUzT1
6. https://algodaily.com/lessons/knapsack-problem-534a2a0d/knapsack-problem-variation-fractional-knapsack-148d1db3
7. https://codingstrain.com/how-to-solve-the-fractional-knapsack-problem/
8. https://www.geeksforgeeks.org/dsa/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/
9. https://www.scribd.com/document/407351878/Fractional-KnapSack-Problem-2
10. https://www.upgrad.com/tutorials/software-engineering/data-structure/fractional-knapsack-problem/