1. Mathematical Formulation for Knapsack Problems

i) Fractional Knapsack Problem

The Fractional Knapsack problem is a solvable problem using the **Greedy Method**. In this variation, items are divisible, meaning we can take a fraction of an item to maximize the total value in the knapsack.

Problem Statement: Given n items, each with a weight w_i and a value (profit) v_i, and a knapsack with a maximum weight capacity W. The objective is to maximize the total value in the knapsack.

Mathematical Formulation:

- **Maximize:** $\sum_{i=1}^{n} v_i x_i$
- **Subject to constraint:** $\sum_{i=1}^{n} w_i x_i \leq W$
- **Variable constraint:** $0 \leq x_i \leq 1$ for all $i = 1, \ldots, n$

Here, x_i represents the fraction of item i taken. Since $0 \leq x_i \leq 1$, we can take any portion of the item.

ii) 0/1 Knapsack Problem

The 0/1 Knapsack problem is solved using **Dynamic Programming**. In this variation, items are indivisible; you must either take the item entirely or leave it behind.

Mathematical Formulation:

- **Maximize:** $\sum_{i=1}^{n} v_i x_i$
- **Subject to constraint:** $\sum_{i=1}^{n} w_i x_i \leq W$
- **Variable constraint:** $x_i \in \{0, 1\}$ for all $i = 1, \ldots, n$

Here, $x_i = 1$ implies the item is included, and $x_i = 0$ implies the item is excluded.

Comparison: Fractional vs. 0/1 Knapsack

Point of Comparison	Fractional Knapsack	0/1 Knapsack
1. Divisibility	Items can be broken into smaller fractions.	Items are indivisible (atomic).
2. Algorithmic Strategy	Solved using the **Greedy Method**.	Solved using **Dynamic Programming** (or Branch & Bound).
3. Complexity	Time complexity is $O(n \log n)$ (due to sorting).	Time complexity is $O(nW)$ (Pseudo-polynomial).
4. Optimality Criteria	Uses Value-to-Weight ratio (v_i/w_i) to select items.	Uses recursive comparison of including vs. excluding items.
5. Selection Variable	x_i can be any real number between 0 and 1.	x_i is strictly binary (0 or 1).
6. Full Capacity Usage	The knapsack is always filled to maximum capacity W (if total item weight > W).	The knapsack might not be filled completely to capacity W.
7. Principle of Optimality	Does not necessarily require the full Principle of Optimality in the same recursive sense.	Relies heavily on the Principle of Optimality and overlapping subproblems.

8. Practical Example	Filling a sack with grains, sugar, or liquids.	Packing a suitcase with electronic devices or solid objects.

Export to Sheets

---

2. Activity Selection Problem (Greedy Strategy)

Problem Analysis: We are given a set of activities with start times (S) and finish times (F). We must select the maximum number of non-conflicting activities. The Greedy strategy dictates that we should always select the activity that **finishes earliest** to leave the maximum remaining time for other activities.

Given Data:

Activity	Start	Finish
A	1	3
B	3	4
C	2	5
D	0	7
E	5	9
F	8	10
G	11	12

Export to Sheets

Step-by-Step Execution:

1. **Sort:** The activities must be sorted in ascending order of their **Finish Times**.
   - Sorted Order: A(1-3), B(3-4), C(2-5), D(0-7), E(5-9), F(8-10), G(11-12).
   - *Note: The table provided in the question was already mostly sorted by finish time.*
2. **Selection Logic:**
   - Select the first activity in the sorted list.
   - Move to the next activity; if its Start Time $\geq$ the Finish Time of the previously selected activity, select it.
3. **Trace:**
   - **Select A** (Start 1, Finish 3). Current Finish Time = 3.
   - Check B: Start 3 $\geq$ 3? **Yes**. **Select B**. Current Finish Time = 4.
   - Check C: Start 2 $\geq$ 4? No. Skip.
   - Check D: Start 0 $\geq$ 4? No. Skip.
   - Check E: Start 5 $\geq$ 4? **Yes**. **Select E**. Current Finish Time = 9.
   - Check F: Start 8 $\geq$ 9? No. Skip.
   - Check G: Start 11 $\geq$ 9? **Yes**. **Select G**. Current Finish Time = 12.

Final Schedule: The execution schedule with the maximum number of non-conflicting activities is: **{ Activity A, Activity B, Activity E, Activity G }** Total Activities: 4.

---

3. Principle of Optimality in Dynamic Programming

Definition: The **Principle of Optimality** states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. In simpler terms, an optimal solution to a problem instance must contain optimal solutions to its sub-problems.

If a problem can be broken down into sub-problems, and we can prove that combining the optimal solutions of these sub-problems yields the global optimal solution, the problem holds the Principle of Optimality.

Mathematical Representation (Bellman Equation): For a problem usually formulated as finding the minimum cost or maximum profit, the recurrence relation is often written as:

$$Cost(i, j) = \min_{k}\{Cost(i, k) + Cost(k, j)\} + C_{constant}$$

Or specifically for the Knapsack problem:

$$V[i, w] = \max \begin{cases} V[i-1, w] & (\text{Exclude item } i) \\ v_i + V[i-1, w-w_i] & (\text{Include item } i) \end{cases}$$

Where:

- $V[i, w]$ is the optimal value for the first i items with weight limit w.
- This recursive structure proves that the solution for size i relies on the optimal solution for size $i-1$.

---

4. 0/1 Knapsack Problem (Dynamic Programming)

Given Data:

- Capacity (W) = 7
- Items: 4

Item	Weight (w_i)	Value (v_i)
A	1	1
B	3	4
C	4	5
D	5	7

Export to Sheets

Solution: We construct a table $K[i][w]$ where i represents the items (0 to 4) and w represents the current weight capacity (0 to 7). Formula: $K[i][w] = \max(K[i-1][w], v_i + K[i-1][w-w_i])$ if $w_i \leq w$.

DP Table Construction:

i / w	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1 (A: 1,1)	0	1	1	1	1	1	1	1
2 (B: 3,4)	0	1	1	4	5	5	5	5
3 (C: 4,5)	0	1	1	4	5	6	6	9
4 (D: 5,7)	0	1	1	4	5	7	8	**9**

Export to Sheets

Detailed Logic for Optimal Value (Cell [4,7]):

- Item D (Weight 5, Value 7). Capacity 7.
- Option 1 (Exclude D): Take value from cell [3, 7] $\rightarrow$ **9**.
- Option 2 (Include D): Value of D (7) + Value of remaining capacity (7-5=2) from row 3. Cell [3, 2] is 1. Total = $7 + 1 = 8$.
- $\max(9, 8) = 9$.

Result: The maximum optimal profit is **9**. **Selected Items:** Backtracking from $K[4][7] = 9$.

1. Did 9 come from row 3? Yes (cell [3,7] is 9). Item D excluded.
2. Did 9 in [3,7] come from row 2? No (cell [2,7] is 5). Item C included.
   - Remaining weight: $7 - 4 = 3$.
3. Go to [2,3]. Value is 4. Did 4 come from row 1? No (cell [1,3] is 1). Item B included.
   - Remaining weight: $3 - 3 = 0$.
4. Remaining weight 0. Stop.

Optimal Set: { Item B, Item C }.

---

5. Job Scheduling to Minimize Penalty (Greedy Algorithm)

Problem Formulation: We have n tasks, each taking 1 unit of time. Each task has a deadline d_i and a penalty p_i if not completed by the deadline. We want to minimize the total penalty. *Note: Minimizing total penalty is mathematically equivalent to maximizing the total profit of the tasks that are completed.*

Algorithm Design (Greedy Strategy):

1. **Sort:** Sort all tasks in **descending order** of their penalties (or profits). High penalties represent high priority.
2. **Initialize:** Create an array `Slots` of size equal to the maximum deadline found in the task list. Initialize all slots as empty (null).
3. **Iterate:** For each task in the sorted list:
   - Check for a free time slot starting from the task's deadline d_i moving backwards to 1 (i.e., search $t = d_i, d_i - 1, \ldots, 1$).

- If a free slot is found at time t, assign the task to that slot.
- If no slot is found (all slots $1 \dots d_i$ are full), the task cannot be scheduled; add its penalty to the Total Penalty.

4. **Output:** The scheduled tasks and the calculated penalty.

Proof of Correctness (Exchange Argument): Let S be the schedule produced by the greedy algorithm. Let O be an optimal schedule. If S is different from O, there must be two jobs i and j where the greedy choice differs. Because we sorted by penalty, the greedy algorithm prioritized the task with the higher penalty (cost of failure). If we swap a high-penalty task included in S with a lower-penalty task from O, the total penalty incurred increases (or profit decreases). Thus, including the highest penalty tasks first whenever possible creates the optimal substructure.

---

6. Fractional Knapsack Problem (Greedy Approach)

Given Data:

- Capacity $W = 15$
- Objects:

Object	Weight (W_i)	Value (V_i)	Ratio (V_i/W_i)
O1	8	10	1.25
O2	6	8	1.33
O3	4	3	0.75
O4	2	4	2.00

Export to Sheets

Step-by-Step Solution:

1. **Calculate Ratios:**
   - O1: $10/8 = 1.25$
   - O2: $8/6 = 1.33$
   - O3: $3/4 = 0.75$
   - O4: $4/2 = 2.0$
2. **Sort by Ratio (Descending):**
   - Order: O4 (2.0), O2 (1.33), O1 (1.25), O3 (0.75).
3. **Fill Knapsack:**
   - **Take O4:** Weight = 2. Remaining Capacity = $15 - 2 = 13$. Acc Value = 4.
   - **Take O2:** Weight = 6. Remaining Capacity = $13 - 6 = 7$. Acc Value = $4 + 8 = 12$.
   - **Take O1:** Weight = 8.
     - Remaining Capacity is 7, but Object O1 weighs 8.
     - We take a **fraction** of O1: $\frac{7}{8}$.
     - Value added: $\frac{7}{8} \times 10 = 8.75$.

- Remaining Capacity = 0.

- **O3:** Capacity full. Skip.

Total Value: $4(\text{from O4}) + 8(\text{from O2}) + 8.75(\text{from fraction of O1}) = \mathbf{20.75}$.

---

7. Chain Matrix Multiplication (Dynamic Programming)

Given Dimensions: Sequence dimensions: $\{4, 10, 3, 12, 20, 7\}$. Matrices:

- $M_1 : 4 \times 10$
- $M_2 : 10 \times 3$
- $M_3 : 3 \times 12$
- $M_4 : 12 \times 20$
- $M_5 : 20 \times 7$

Objective: Find the multiplication sequence that minimizes total scalar multiplications. Let $m[i, j]$ be the minimum number of multiplications to compute the product chain from matrix i to j. Formula: $m[i, j] = \min_{i \le k < j}\{m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j\}$

Calculation Highlights: We compute chains of length $L = 2$ to $L = 5$.

1. **L=2 (Product of 2 matrices, cost is $p_{i-1}p_i p_{i+1}$):**
   - $m[1, 2](M_1M_2) = 4 \times 10 \times 3 = 120$
   - $m[2, 3](M_2M_3) = 10 \times 3 \times 12 = 360$
   - $m[3, 4](M_3M_4) = 3 \times 12 \times 20 = 720$
   - $m[4, 5](M_4M_5) = 12 \times 20 \times 7 = 1680$
2. **L=3 (Chains of 3):**
   - $m[1, 3]: \min((M_1M_2)M_3, M_1(M_2M_3))$
     - $k = 1 : 120 + 0 + (4 \times 3 \times 12) = 264$
     - $k = 2 : 0 + 360 + (4 \times 10 \times 12) = 840$
     - $\min = 264$.
   - $m[2, 4]: \min((M_2M_3)M_4, M_2(M_3M_4))$
     - $k = 2 : 360 + 0 + (10 \times 12 \times 20) = 2760$
     - $k = 3 : 0 + 720 + (10 \times 3 \times 20) = 1320$
     - $\min = 1320$.
   - $m[3, 5]: \min((M_3M_4)M_5, M_3(M_4M_5))$
     - $k = 3 : 720 + 0 + (3 \times 20 \times 7) = 1140$
     - $k = 4 : 0 + 1680 + (3 \times 12 \times 7) = 1932$
     - $\min = 1140$.
3. **L=4 (Chains of 4):**
   - $m[1, 4]$: Check splits at $k = 1, 2, 3$.
     - Minimizing calculation yields optimal split at $k = 3$ usually for this dataset.
     - $m[1, 3] + m[4, 4] + p_0p_3p_4 = 264 + 0 + 4 \times 12 \times 20 = 1224$.

- $m[2,5]$: Check splits.
  - Result is usually derived from $M_2(M_3(M_4M_5))$ logic. $m[2,2]+m[3,5]+10\times 3\times 7=0+1140+210=1350$.

4. **L=5 (Final Answer $m[1,5]$):**
   - We compare:
     - $k=1: m[1,1]+m[2,5]+4\times 10\times 7=0+1350+280=1630$.
     - $k=2: m[1,2]+m[3,5]+4\times 3\times 7=120+1140+84=1344$.
     - $k=3: m[1,3]+m[4,5]+4\times 12\times 7=264+1680+336=2280$.
     - $k=4: m[1,4]+m[5,5]+4\times 20\times 7=1224+0+560=1784$.

Optimal Result: The minimum number of multiplications is **1344**. The split occurred at $k=2$, meaning $(M_1M_2)\times(M_3M_4M_5)$.

---

8. Limitations of Dynamic Programming for Knapsack

Dynamic Programming (DP) is generally robust, but it struggles in the following situations for the Knapsack problem:

1. **Huge Capacity (W):** The time complexity of the DP solution is $O(n\times W)$. This is **Pseudo-Polynomial**, not true polynomial. If the capacity W is a very large number (e.g., 10^9) but n (items) is small, the algorithm becomes computationally infeasible because the DP table requires W columns.
2. **Non-Integer Weights:** The standard DP approach relies on using weights as **indices** for the array (table). If weights are floating-point numbers (e.g., 4.5 kg), they cannot be used as array indices directly. You would have to scale the numbers (multiply by 10 or 100), which increases W drastically, leading back to problem #1.

---

9. Control Abstraction for Greedy Method

Control Abstraction Code:

C

```
Algorithm Greedy(a, n)
{
    // a is the array of inputs, n is the number of inputs
    solution = 0; // Initialize empty solution set
    for (i = 1; i <= n; i++)
    {
        x = Select(a); // Select an input based on greedy criteria
        if (Feasible(solution, x))
        {
            solution = Union(solution, x);
        }
    }
    return solution;
}
```

Time Complexity: The time complexity depends primarily on the `Select` function, which usually

involves sorting the inputs.

- If sorting is used: $O(n \log n)$.
- The loop runs n times, and feasibility checks are usually $O(1)$.
- Therefore, the overall complexity is dominated by the sorting: $O(n \log n)$..

Uses of Control Abstraction:

1. **Standardization:** It provides a generic template to solve various maximization/minimization problems.
2. **Modularity:** Separates the logic of Selection and Feasibility from the main loop.
3. **Ease of Implementation:** Helps in mapping real-world problems (like wiring, routing) to algorithmic structures easily.

---

10. Comment on "Principle of Optimality"

Statement: "Problem which does not satisfy the principle of optimality cannot be solved by dynamic programming."

Comment: This statement is **TRUE**. Dynamic Programming works by breaking a complex problem into simpler subproblems. It solves each subproblem once and stores the result (memoization). However, this technique relies on the assumption that the global optimal solution is constructed from the optimal solutions of its subproblems.

Example of Failure (Longest Simple Path): Consider finding the **Longest Simple Path** between two nodes in a graph.

- Path $A \to C$: Assume the longest path goes through B ($A \to B \to C$).
- However, the longest simple path from $A \to B$ combined with the longest simple path from $B \to C$ might create a cycle (visiting a node twice), making it not "simple".
- Because the optimal solution for the subparts does not guarantee the optimal valid solution for the whole, DP cannot be used here..

---

11. Matrix Chain Multiplication (4 Matrices)

Given Data:

- A1: 3×5 ($p_0 = 3, p_1 = 5$)
- A2: 5×4 ($p_1 = 5, p_2 = 4$)
- A3: 4×2 ($p_2 = 4, p_3 = 2$)
- A4: 2×4 ($p_3 = 2, p_4 = 4$)
- Sequence: $p = \{3, 5, 4, 2, 4\}$

DP Calculation:

1. **L=2:**
   - $m[1, 2] = 3 \times 5 \times 4 = 60$
   - $m[2, 3] = 5 \times 4 \times 2 = 40$
   - $m[3, 4] = 4 \times 2 \times 4 = 32$

2. **L=3:**
   - $m[1,3]: \min\{(A1A2)A3, A1(A2A3)\}$
     - $k=1: 60+0+(3\times4\times2)=84$
     - $k=2: 0+40+(3\times5\times2)=70$ **(Select)**
   - $m[2,4]: \min\{(A2A3)A4, A2(A3A4)\}$
     - $k=2: 40+0+(5\times2\times4)=80$
     - $k=3: 0+32+(5\times4\times4)=112$
     - **Select 80**
3. **L=4 (Total):**
   - $m[1,4]$:
     - $k=1: m[1,1]+m[2,4]+3\times5\times4=0+80+60=140$
     - $k=2: m[1,2]+m[3,4]+3\times4\times4=60+32+48=140$
     - $k=3: m[1,3]+m[4,4]+3\times2\times4=70+0+24=94$

Result:

- **Minimum Multiplications: 94.**
- **Optimal Sequence:** Since the minimum occurred at $k=3$, the split is $(A1A2A3)(A4)$. Inside $(A1\dots A3)$, the optimal split was at $k=1$, so $A1(A2A3)$.
- **Final Parenthesization: ((A1(A2 A3)) A4).**

---

12. Job Scheduling Algorithm (Greedy)

Definition: Job scheduling is an optimization problem where we have a set of jobs, each with a deadline and a profit. The goal is to select a subset of jobs to execute within their deadlines to maximize total profit. Each job takes 1 unit of time.

Problem Instance:

Job	Deadline	Profit
J1	2	60
J2	1	100
J3	3	20
J4	2	40
J5	1	20

Export to Sheets

Solution Strategy:

1. **Sort** jobs by Profit (Descending). Order: J2(100), J1(60), J4(40), J3(20), J5(20).
2. **Determine Max Deadline:** 3. We have 3 time slots: [1], [2], [3].
3. **Assignment:**
   - **J2 (Deadline 1):** Slot 1 is empty. Assign J2 to Slot 1. (Slots: [J2, _, _])

- **J1 (Deadline 2):** Slot 2 is empty. Assign J1 to Slot 2. (Slots: [J2, J1, _])
- **J4 (Deadline 2):** Slot 2 is full. Check Slot 1. Full. Reject J4.
- **J3 (Deadline 3):** Slot 3 is empty. Assign J3 to Slot 3. (Slots: [J2, J1, J3])
- **J5 (Deadline 1):** Slot 1 full. Reject.

Final Job Sequence: J2 → J1 → J3.

Total Profit: $100 + 60 + 20 = 180$.

---

13. Job Sequencing High-Level Description & Instance

High-Level Description: The Job Sequencing algorithm with deadlines works as follows:

1. **Input:** A set of n jobs, each with profit P_i and deadline D_i.
2. **Sorting:** Sort the jobs in non-increasing order of their profits.
3. **Slot Initialization:** Find the maximum deadline D_{max}. Create a timeline of size D_{max}.
4. **Greedy Allocation:** For each job in the sorted list, look for the latest possible empty slot t such that $t \leq D_i$. If found, place the job there. If no such slot exists, discard the job.
5. **Output:** The set of scheduled jobs and total profit.

Instance Solution:

- $n = 5$
- Profit $P = \{20, 15, 10, 5, 1\}$
- Deadline $D = \{2, 2, 1, 3, 3\}$

The Profit vector is already sorted. Jobs paired: J1(20, D=2), J2(15, D=2), J3(10, D=1), J4(5, D=3), J5(1, D=3). Max Deadline = 3. Slots: [1, 2, 3].

1. **J1 (20, D=2):** Check slot 2. Empty. Assign J1. [_, J1, _]
2. **J2 (15, D=2):** Check slot 2 (Full). Check slot 1. Empty. Assign J2. [J2, J1, _]
3. **J3 (10, D=1):** Check slot 1 (Full). Reject.
4. **J4 (5, D=3):** Check slot 3. Empty. Assign J4. [J2, J1, J4]
5. **J5 (1, D=3):** Check slot 3 (Full). Slot 2 (Full). Slot 1 (Full). Reject.

Feasible Solution: {J2, J1, J4}

Optimal Profit: $15 + 20 + 5 = \mathbf{40}$.