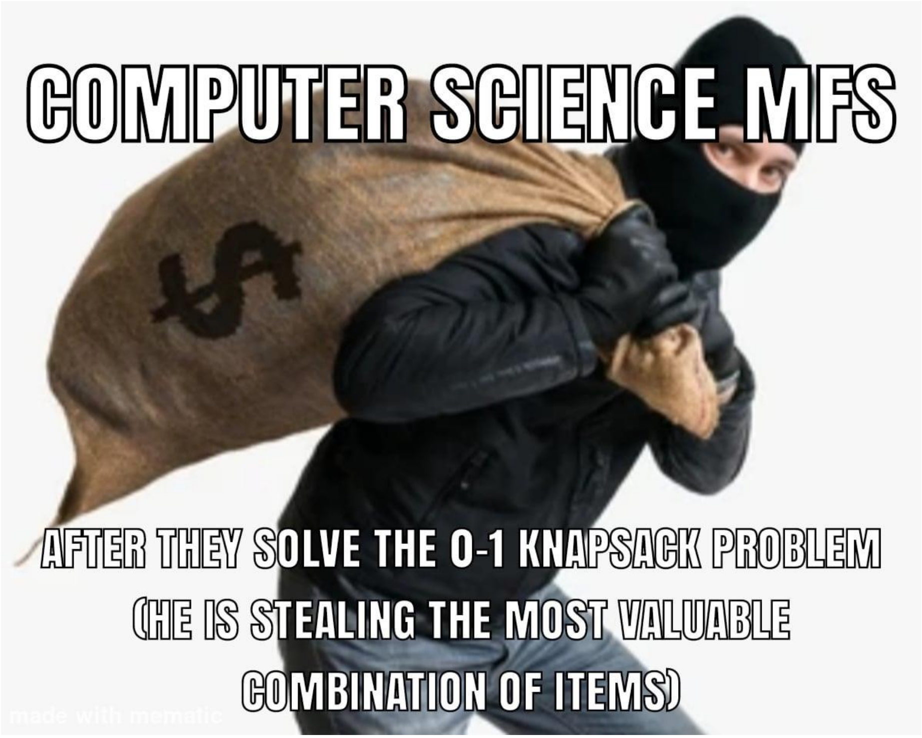
Extra Slides: 0-1 Knapsack



Sec 101: MW 3:00-4:00pm DOW 1018
IA: Eric Khiu

0-1 Knapsack Set-Up

- You have a set of n items, each with weight w_i and value v_i, and you have a knapsack with maximum weight capacity C

- Inputs:
  - n-length array of positive integer weights $W = [w_1, \dots, w_n]$
  - n-length array of positive integer values $V = [v_1, \dots, v_n]$
  - Capacity of the knapsack $C \in \mathbb{N}$

- **Goal**: pick a subset of items $S \subseteq \{1,2, \dots n\}$ that maximizes the value of the knapsack $\sum_{i \in S} v_i$, while staying within the capacity $\sum_{i \in S} w_i \leq C$

0-1 Knapsack Recurrence

	x_1	x_2	x_3	…	x_{n-2}	x_{n-1}	x_n
y_1							
y_2							
y_3							
$\vdots$							
y_{m-2}							
y_{m-1}							
y_m							

	x_1	x_2	x_3	…	x_{n-2}	x_{n-1}	x_n
y_1							
y_2							
y_3							
$\vdots$							
y_{m-2}							
y_{m-1}							
y_m							

- **Step 0: Dimensionality**
  - 2-dimensional: one for item and one for capacity
- **Step 1: Subject of recurrence**
  - Let $K(i, j)$ be an optimal knapsack solution using only items up to index i, and having capacity only up to j
- **Step 2: Base cases**
  - $i = 0$ (No item) or $j = 0$ (no space): $K(i, j) = 0$
  - $w_i > j$ (Not enough space to consider item i): $K(i, j) = K(i - 1, j)$

0-1 Knapsack Recurrence

- **Step 3: Optimal sub-solution**
  - [Sub-solution] For items 1, …, i, how do we reduce the problem?
    - Deal with [Items 1, …, i-1] and [Item i] separately
    - **Q:** What would happen if we take item i in the knapsack?
      - Capacity reduces by w_i, total value increases by v_i
      - Else: Both capacity and total value remain unchanged
  - [Optimal] *Choose* between whether to include the ith item
    - Maximization problem: Use max
    - Objective function? $K(i,j) = \max\{K(i-1, j-w_i) + v_i, K(i-1), j)\}$
- Recurrence relation

$$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1,j) & \text{if } w_i > j \\ \max\{K(i-1, j-w_i) + v_i, K(i-1), j)\} & \text{otherwise} \end{cases}$$

$$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1,j) & \text{if } w_i > j \\ \max\{K(i-1, j-w_i) + v_i, K(i-1), j)\} & \text{otherwise} \end{cases}$$

Top-down Recursion

- Implement the recursion as in the recurrence

Input: Integers n, C, arrays W, V. Again, note the 1-based indexing.
Output: The maximum total value of objects the Knapsack can hold

```
function KNAPSACK(n, C, W, V)
    if n = 0 or C = 0 then
        return 0
    if W[n] > C then
        return Knapsack(n − 1, C, W, V)
    return max(Knapsack(n − 1, C − W[n], W, V) + V[n], Knapsack(n − 1, C, W, V))
```

- **Runtime:** $O(2^n)$
- **Space:** $O(n)$

- Easy to translate from recurrence relation
- No additional data structures necessary

- A lot of recursive calls- may not be time-efficient
- Correctness proof is usually harder
  - Not as smooth as bottom up- imagine proving by induction $P(k) \Rightarrow P(k+1)$, but we can't do that with recursive top-down
- Additional concern on segmentation fault

$$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1,j) & \text{if } w_i > j \\ \max\{K(i-1, j-w_i) + v_i, K(i-1), j)\} & \text{otherwise} \end{cases}$$

Top-down Memoization

- Same as before, but include a memo recording results of previous recursive calls

Input: Integers n, C, arrays W, V, and lookup-table DP with values initialized to -1. Again, note the 1-based indexing.
Output: The maximum total value of objects the Knapsack can hold

```
function KNAPSACK(n, C, W, V, DP)
    if n = 0 or C = 0 then
        return 0
    if DP[n − 1][C] = −1 then
        DP[n − 1][C] ←Knapsack(n − 1, C, W, V, DP)
    if W[n] > C then
        return DP[n − 1][C]
    if DP[n − 1][C − W[n]] = −1 then
        DP[n − 1][C − W[n]] ←Knapsack(n − 1, C − W[n], W, V, DP)
    return max(DP[n − 1][C − W[n]] + V[n], DP[n − 1][C])
```

- **Runtime:** $O(n \cdot C)$
- **Space:** $O(n \cdot C)$

- Usually better time-complexity than top-down recursion
- While harder than recursion, the logic is often more intuitive than bottom-up

- May still be slower than bottom-up
- Correctness proof may still be harder than bottom-up
- Additional concern on segmentation fault

$$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1,j) & \text{if } w_i > j \\ \max\{K(i-1, j-w_i) + v_i, K(i-1), j)\} & \text{otherwise} \end{cases}$$

Bottom-up (Tabulation)

- Build the table without recursion
- Iterate over previous results to fill the cells

Input: Integers n, C, arrays W, V, and memo table DP.
Output: The maximum total value of objects the Knapsack can hold

```
function KNAPSACK(n, C, W, V)
    DP[n][C] ← −1                          ▷ Initialize values in lookup-table to -1
    for i = 0 : n do
        DP[i][0] = 0
    for j = 0 : C do
        DP[0][j] = 0
    for i = 1 : n do
        for j = 1 : C do
            if W[i] > j then
                DP[i][j] = DP[i − 1][j]
            else
                DP[i][j] = max(DP[i − 1][j − W[i]] + V[i], DP[i − 1][j])
    return DP[n][C]
```

- Almost always fastest in practice
- Segmentation fault is less likely

- Less intuitive
- Common mistake 1: wrong direction for for-loop
- Common mistake 2: wrong initialization

- **Runtime:** $O(n \cdot C)$
- **Space:** $O(n \cdot C)$

Visualizer