Dynamic Programming 2 - 0-1 Knapsack

▼ Problem definition

- Suppose you have an array of objects O[1...n], an array of their profits P[1...n], an array of their weights W[1...n], and a maximum weight capacity M.
- You want to find which items you can take such that the total profit is maximized without exceeding the maximum weight capacity M.

▼ First approach: Select maximum profit first

- The first approach you could take to this is to select all of the items in order of decreasing profit until your maximum weight capacity is reached.
- This doesn't produce the most optimal solution, however.
- Counter example:
  - P = [60, 45, 40]
  - W = [55, 49, 51]
  - M = 100
  - This algorithm would select item 1, which would not be optimal. Optimal solution is item 2 and 3.

▼ Second approach: Select highest profit to weight ratio first

- Select items in order of profit divided by weight.
- Counter example:
  - P = [60, 45, 40]
  - W = [55, 49, 51]
  - M = 100

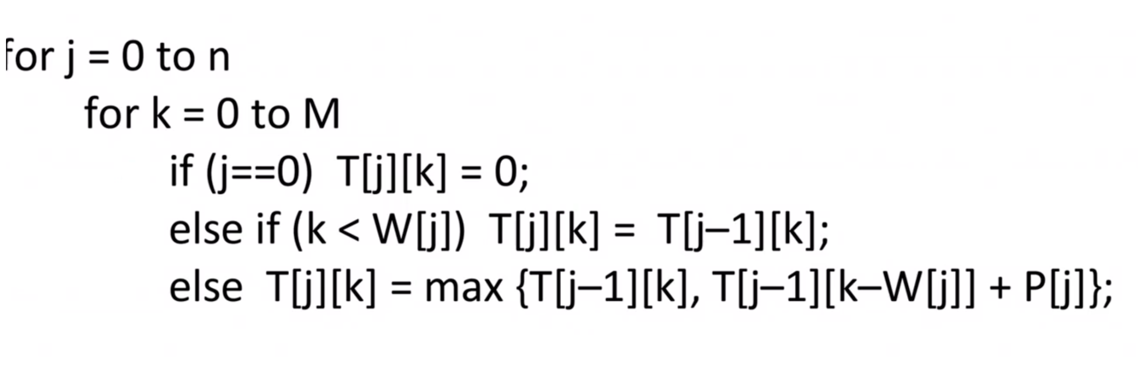
        - This algorithm would select item 1, which would not be optimal. Optimal solution is item 2 and 3.
    - This is the correct approach for the *fractional knapsack*, however.
- It's obvious that a *greedy algorithm* is not going to work for this. We need a recursive solution
- ▼ Recursive solution to the algorithm
    - Instead of trying to build a solution in the form T(n, M) (where we're using *all* of the items and *all* of the maximum weight capacity), let's define a function that only deals with a subset of the problem.
    - $T(j,k) =$ the max possible total profit such that we can only choose from objects $\{1...j\}$ and maximum weight capacity is k. ($0 \leq j \leq n, 0 \leq k \leq M$).
        - This lets us deal with solving a *simpler* problem.
    - ▼ Formula:
        - $T(j,k) = 0$ if $j = 0$.
        - $T(j,k) = T(j-1,k)$ if $j > 0$ and $k < W[j]$.
            - $k < W[j]$ means that the weight of j is greater than the currently selected weight capacity k.
        - $T(j,k) = max\ \{T(j-1,k), T(j-1,k-W[j]) + P[j]\ \}$ if $j > 0$ and $k \geq W[j]$.
            - We're taking the max out of two subproblems.
            - The first option, $T(j-1,k)$ is **not using j** (the max profit not using item j).
            - The second option, $T(j-1,k-W[j]) + P[j]$, **uses item j**.
                - We want to put item J in the knapsack, so we reduce the size of the capacity. The new capacity is $k - W[j]$.
                - We add the profit to this to put J in the knapsack.
    - ▼ Code for the formula:

```
const T = (j: number, k: number): number => {
  if (j === 0) return 0;
  if (k < W[j]) return T(j - 1, k);
  return max(T(j - 1, k), T(j - 1, k - W[j]) + P[j]);
}
```

▼ Dynamic programming algorithm

- Keep an array T[0…n][0…M] (2d array of all items M times).

▼ Pseudocode

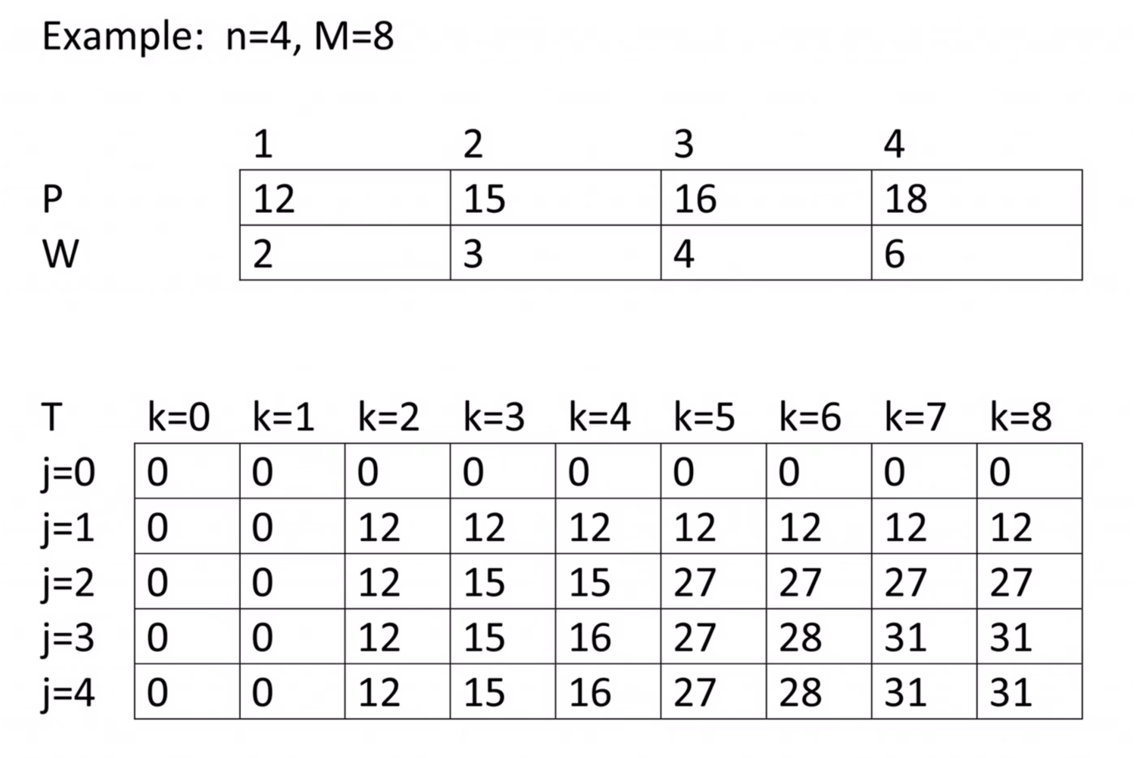


```
for j = 0 to n
    for k = 0 to M
        if (j==0)  T[j][k] = 0;
        else if (k < W[j])  T[j][k] =  T[j–1][k];
        else  T[j][k] = max {T[j–1][k], T[j–1][k–W[j]] + P[j]};
```

- This approach has a problem though, it solves a lot of the problems *over and over again*.

▼ Example

- We fill in the array row by row, then column by column. Filling a row at a time.



Example: n=4, M=8

	1	2	3	4
P	12	15	16	18
W	2	3	4	6

T	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
j=0	0	0	0	0	0	0	0	0	0
j=1	0	0	12	12	12	12	12	12	12
j=2	0	0	12	15	15	27	27	27	27
j=3	0	0	12	15	16	27	28	31	31
j=4	0	0	12	15	16	27	28	31	31

- The first row is all 0 because there can be no profit.
- Second row, we can use object 1. We look at the top row when the item can't fit and copy it down.
- When the item *can* fit (when W[j] ≤ k) then we look at the item above, and then the other thing (adding j to the stuff).

▼ Determining which items give us the optimal solution

- Note that *this doesn't tell us how to get the items that give us the max profit*, it only tells us the max profit we can get.
- **We trace our steps back**.
- We just trace our steps back with the max, (with j or without j). If the max is without j, that's not our solution. If it is with j, then we include it in our solution. We just walk back in the table!