FEATURE	GREEDY METHOD	DYNAMIC PROGRAMMING
Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.
Memorization	It is more efficient in terms of memory as it never look back or revise previous choices	It requires dp table for memorization and it increases it's memory complexity.
Time complexity	Greedy methods are generally faster. For example, Dijkstra's shortest pathalgorithm takes O(ELogV + VLogV) time.	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.
Example	Fractional knapsack .	0/1 knapsack problem

The Idea of Dynamic Programming

Dynamic programming is a method for solving optimization problems.

The idea: Compute the solutions to the subsub-problems *once* and store the solutions in a table, so that they can be reused (repeatedly) later.

Remark: We trade space for time.

0-1 Knapsack Problem

Formal description: Given two n-tuples of positive numbers

$$\langle v_1, v_2, \ldots, v_n \rangle \quad \text{and} \quad \langle w_1, w_2 \ldots, w_n \rangle,$$

and $W > 0$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of files to store) that

$$\text{maximizes} \quad \sum_{i \in T} v_i,$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Remark: This is an optimization problem.

Brute force: Try all 2^n possible subsets T.

Question: Any solution better than the brute-force?

The Idea of Developing a DP Algorithm

Step1: Structure: Characterize the structure of an optimal solution.

– Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.

Step2: Principle of Optimality: Recursively define the value of an optimal solution.

– Express the solution of the original problem in terms of optimal solutions for smaller problems.

The Idea of Developing a DP Algorithm

Step 3: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution: Construct an optimal solution from computed information.

Steps 3 and 4 may often be combined.

Developing a DP Algorithm for Knapsack

Step 1: Decompose the problem into smaller problems.

We construct an array $V[0..n, 0..W]$.
For $1 \leq i \leq n$, and $0 \leq w \leq W$, the entry $V[i, w]$ will store the maximum (combined) computing time of any subset of files $\{1, 2, \ldots, i\}$ of (combined) size at most w.

If we can compute all the entries of this array, then the array entry $V[n, W]$ will contain the maximum computing time of files that can fit into the storage, that is, the solution to our problem.

Developing a DP Algorithm for Knapsack

Step 2: Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial Settings: Set

$$V[0, w] = 0 \quad \text{for } 0 \leq w \leq W, \quad \text{no item}$$
$$V[i, w] = -\infty \quad \text{for } w < 0, \quad \text{illegal}$$

Recursive Step: Use

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

for $1 \leq i \leq n$, $0 \leq w \leq W$.

Correctness of the Method for Computing $V[i, w]$

Lemma: For $1 \leq i \leq n$, $0 \leq w \leq W$,

$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$.

Proof: To compute $V[i, w]$ we note that we have only two choices for file i:

Leave file i: The best we can do with files $\{1, 2, \ldots, i-1\}$ and storage limit w is $V[i-1, w]$.

Take file i (only possible if $w_i \leq w$): Then we gain v_i of computing time, but have spent w_i bytes of our storage. The best we can do with remaining files $\{1, 2, \ldots, i-1\}$ and storage $(w - w_i)$ is $V[i-1, w-w_i]$.
Totally, we get $v_i + V[i-1, w-w_i]$.

Note that if $w_i > w$, then $v_i + V[i-1, w-w_i] = -\infty$ so the lemma is correct in any case.

Developing a DP Algorithm for Knapsack

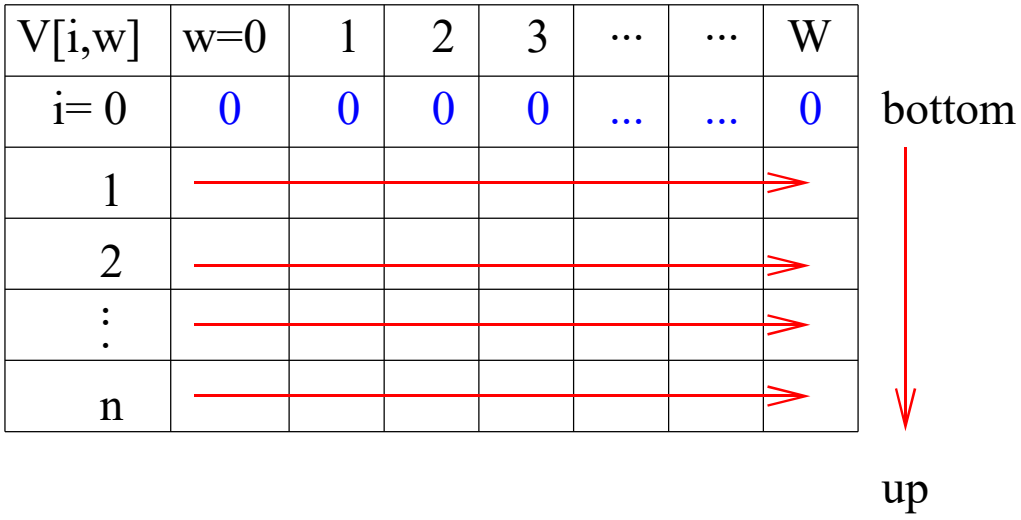
Step 3: Bottom-up computing $V[i, w]$ (using iteration, not recursion).

Bottom: $V[0, w] = 0$ for all $0 \leq w \leq W$.

Bottom-up computation: Computing the table using

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

row by row.



Example of the Bottom-up computation

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Remarks:

- The final output is $V[4, 10] = 90$.
- The method described does not tell which subset gives the optimal solution. (It is $\{2, 4\}$ in this example).

The Dynamic Programming Algorithm

```
KnapSack(v, w, n, W)
{
  for (w = 0 to W) V[0, w] = 0;
  for (i = 1 to n)
    for (w = 0 to W)
      if (w[i] ≤ w)
        V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
      else
        V[i, w] = V[i − 1, w];
  return V[n, W];
}
```

Time complexity: Clearly, $O(nW)$.

The Complete Algorithm for the Knapsack Problem

```
KnapSack(v, w, n, W)
{
    for (w = 0 to W) V[0, w] = 0;
    for (i = 1 to n)
        for (w = 0 to W)
            if ((w[i] ≤ w) and (v[i] + V[i − 1, w − w[i]] > V[i − 1, w]))
            {
                V[i, w] = v[i] + V[i − 1, w − w[i]];
                keep[i, w] = 1;
            }
            else
            {
                V[i, w] = V[i − 1, w];
                keep[i, w] = 0;
            }
    K = W;
    for (i = n downto 1)
        if (keep[i, K] == 1)
        {
            output i;
            K = K − w[i];
        }
    return V[n, W];
}
```