Dynamic Programming

02

KNAPSACK

“Given the weights and profits of ‘N’ items, we are asked to put these items in a knapsack that has a capacity ‘C’. The goal is to get the maximum profit from the items in the knapsack”

0-1 KNAPSACK PROBLEM

- A variation of a *bin packing* problem
  - You have a set of items
  - Each item has a weight and a value
  - You have a knapsack with a weight limit
  - **Goal:** Maximize the ***value*** of the items you put in the knapsack without exceeding the weight limit.
-
- In the 0-1 knapsack problem, we can't *exceed* the weight limit, but the optimal solution may be *less* than the weight limit

Example

Items: { Apple, Orange, Banana, Melon }

Weights: { 2, 3, 1, 4 }

Profits: { 4, 5, 3, 7 }

Knapsack capacity: 5

Different combinations of fruits in the knapsack, such that their total weight is not more than 5:

Apple + Orange (total weight 5) => 9 profit

Apple + Banana (total weight 3) => 7 profit

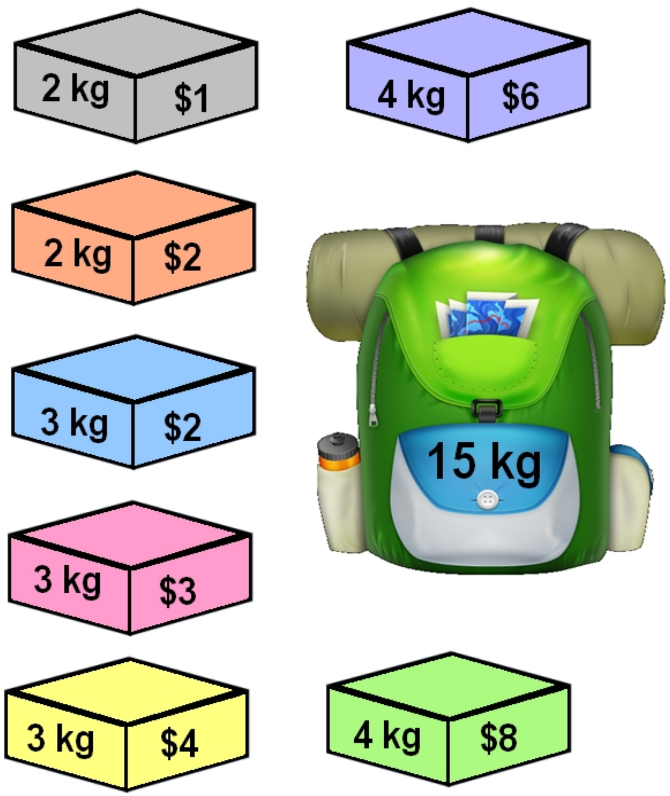
Orange + Banana (total weight 4) => 8 profit

Banana + Melon (total weight 5) => 10 profit

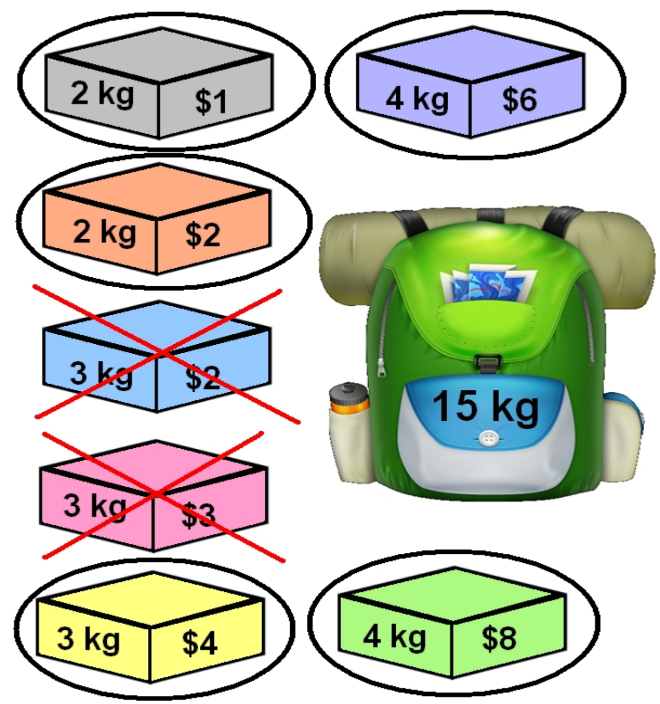
Banana + Melon is the best combination, as it gives us the maximum profit and the total weight does not exceed the capacity

Example

Problem



Solution



Example

- A thief breaks into a house, carrying a knapsack...
  - He can carry up to 25 pounds of loot
  - He has to choose which of N items to steal
  - Each item has some weight and some value
  - “0-1” because each item is stolen (1) or not stolen (0)
  - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds

Problem :

- Given two integer arrays to represent weights and values of ‘n’ items, we need to find a subset of these items which will give us maximum profit such that their cumulative weight is not more than a given number ‘C’
  - Input
    - Capacity ‘C’
    - n items with weights w_i and values v_i
  - Output: a set of items S such that
    - the sum of weights of items in S is at most C

      and the sum of values of items in S is maximized

Solution

The straight forward way:

Example:

$$n = 3$$
$$(p_1, \quad p_2, \quad p_3) = (1, \quad 2, \quad 5)$$
$$(w_1, \quad w_2, \quad w_3) = (2, \quad 3, \quad 4)$$
$$M = 6$$

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	0	0	0	0
0	0	1	4	5
0	1	0	3	2
0	1	1	-	-
1	0	0	2	1
$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{6}$	$\boxed{\underline{6} \leftarrow \textit{solution}}$
1	1	0	5	3
1	1	1	-	-

The complexity is *O(2^n)*

The Dynamic Programming way:

- Sub-problems:
  - Knapsack with a smaller knapsack.

- Recursive relationship

$$f_0(X) = 0$$

$$f_i(X) = \max\{ f_{i-1}(X), p_i + f_{i-1}(X - W_i)\}$$

The Dynamic Programming way:

$f_i(X)$ = max profit generated from $x_1, x_2, \ldots, x_i$
subject to the capacity X

$$\begin{cases} f_0(X) = 0 \\ f_i(X) = \max\left\{ \underbrace{f_{i-1}(X)}, \underbrace{p_i + f_{i-1}(X - W_i)} \right\} \end{cases}$$

Example:

$$\begin{pmatrix} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \end{pmatrix} = \begin{pmatrix} w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \end{pmatrix}$$
$$= \begin{pmatrix} 100 & 50 & 20 & 10 & 7 & 3 \end{pmatrix}$$

$$M = 165$$

Question: to find $f_6\left(165\right)$

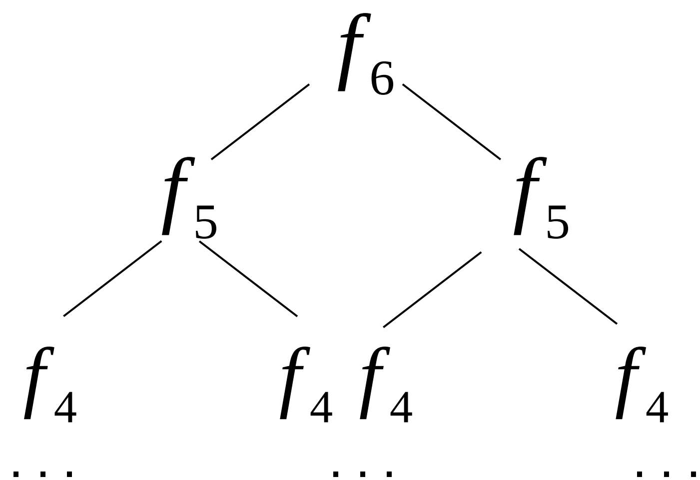
Use backward approach:

$$f_6(165) = \max\left\{ f_5(165), f_5(162) + 3 \right\} = \ldots$$
$$f_5(165) = \max\left\{ f_4(165), f_4(158) + 7 \right\} = \ldots$$

$$\ldots$$

The result:

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$



Therefore, the complexity of 0/1 Knapsack is $O(2^n)$

Example: $M = 6$

	Object 1	Object 2	Object 3	Object 4
p_i	3	4	8	5
w_i	2	1	4	3

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3
2	0	4	4	7	7	7	7
3	0	4	4	7	8	12	12
4	0	4	4	7	9	12	12

Max profit is 12

By tracing which entries lead to this max-profit solution, we can obtain the optimal solution - Object 1,2 and 4.

$M+1$

$n+1$

	0	1	…	j-w_i	…	j	…	M
0	0	0	0	0	0	0	0	0
1								
…								
i-1				$f_{i-1}(j-w_i)$		$f_{i-1}(j)$		
i				$+p_i$		$f_i(j)$		
…								
n								

$(M+1)(n+1)$ entries and constant time each $\Rightarrow O(Mn)$

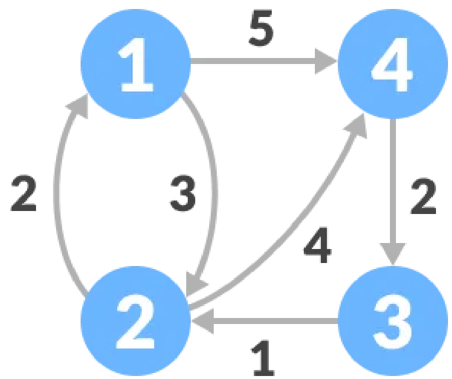
Floyd-Warshall algorithm

- Single-sourceshortest path in weighted graphs.
  - Bellman-Fordalgorithm
  - Dijkstra's algorithm!
- Floyd-Warshall algorithm
  - An "all-pairs" shortest path algorithm
  - Another example of dynamic programming

Floyd's Algorithm: All pairs shortest path

- All pairs shortest path
  - The problem: find the shortest path between every pair of vertices of a graph.
  - The graph: may contain negative edges but no negative cycles.
  -
  - A representation: a weight matrix where
    W(i,j)=0 if i=j.
    W(i,j)=∞ if there is no edge between i and j.
    W(i,j)="weight of edge"
  -

Example:



Step 1. Create a matrix dimension N*N and Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Step:2: Create a matrix A^1 using matrix A^0.

Let k be the intermediate vertex

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

A direct distance from the source to the destination is greater than the path through the vertex k, → **if (A[i][j] > A[i][k] + A[k][j])**

Step:3: Create a matrix A^2 using matrix A^1

k is the second vertex (i.e. vertex 2)

In this step, k is vertex 2. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Step:4: Create a matrix A^3 using matrix A^2

k is the second vertex (i.e. vertex 3)

In this step, k is vertex .

Step:5: Create a matrix A^4 using matrix A^3

k is the second vertex (i.e. vertex 4)

In this step, k is vertex .

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

A4 gives the shortest path between each pair of vertices.

Floyd’s Algorithm:

Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles

Pseudocode for this basic version follows:

```
let dist be a |V| × |V| array of minimum distances
initialized to ∞ (infinity)
for each edge (u,v)
   dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for each vertex v
   dist[v][v] ← 0
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
             dist[i][j] ← dist[i][k] + dist[k][j]
         end if
```

Subproblems ?

How can we define the shortest distance $d_{i,j}$ in terms of "smaller" problems?

Elements of dynamic programming

- Big problems break up into little problems.
  - eg, Shortest path with at most k edges
- The optimal solution of a problem can be expressed in terms of optimal solutions of smaller sub-problems.

eg, $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + weight(a,b)\} \}$

optimal sub-structurere

The sub-problems overlap a lot.

- Lots of different entries of $d^{(k)}$ ask for $d^{(k-1)}[a]$.

"We can save time by solving a sub-problem just once and storing the answer"

overlapping sub-problems

Floyd-Warshall Algorithm

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            A^k[i, j] = min (A^(k-1)[i, j], A^(k-1)[i, k] + A^(k-1)[k, j])
return A
```

- Time Complexity
  - There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is O(n3).
- Space Complexity
  - The space complexity of the Floyd-Warshall algorithm is O(n2)

Floyd Warshall Algorithm Applications

- To find the shortest path is a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite