The Knapsack problem

Competitive Programming

Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Fall 2023

Objectives

- Solve the 0,1 knapsack problem:
  - Using top-down DP
  - Using bottom up DP
  - Using memory saving bottom up DP
- Solve the knapsack with repetition problem

The Problem

- You have a sack of capacity W (usually thought of as a weight).
- There are N items with costs $c_0, c_1, \ldots, c_{n-1}$, and weights $w_0, w_1, \ldots, w_{n-1}$.
- We want to pick a subset of problems that maximizes the cost and has weight $\leq W$. Result is the maximum cost.
- Before we go on, make sure you can express:
  - Why this problem needs DP (cannot be greedy).
  - What is the "state" that we will keep track of?

0/1 Variation, recursive

- For each item i:
  - if the $w_i > W$ (item is larger than the remaining weight), return zero.
  - otherwise the choice: pick or don't pick.
    - If pick, recurse on $c_i + k(i+1, W - w_i)$
    - If no pick , recurse on $k(i_1, W)$.

```
int knap(vi &weights, vi &costs, int item, int remW) {
    if (item == -1 || remW <= 0) return 0;



    if (remW < weights[item])  // to big, can't take
        return knap(dp,weights,costs,item-1,remW);
    else return
           max(knap(dp,weights,costs,item+1, //take
                  remW - weights[item]) + costs[item],
               knap(dp,weights,costs,item-1,remW) ) ;
}
```

0/1 Variation, top down DP

- For each item i:
  - if the $w_i > W$ (item is larger than the remaining weight), return zero.
  - otherwise the choice: pick or don't pick.
    - If pick, recurse on $c_i + k(i+1, W - w_i)$
    - If no pick , recurse on $k(i_1, W)$.

```
int knap(vi &weights, vi &costs, int item, int remW) {
    if (item == N || remW <= 0) return 0;
    int & val = dp[item][remW];
    if (val != -1) return val; // do the DP
    if (remW < weights[item])  // to big, can't take
        return val = knap(dp,weights,costs,item+1,remW);
    else return val =
           max(knap(dp,weights,costs,item+1, //take
                    remW - weights[item]) + costs[item],
               knap(dp,weights,costs,item+1,remW) ) ;
}
```

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

$$\begin{array}{c|cccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & & & & & & \\
1 & & & & & & \\
2 & & & & & &
\end{array}$$

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

	0	1	2	3	4	5
0	0					
1				$(2 + x_1)$		
2						$(7 + x_2)$

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

	0	1	2	3	4	5
0	0			4		
1				$(2 + x_1)$		
2						$(7 + x_2)$

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

	0	1	2	3	4	5
0	0			4		
1				4		
2						$(7 + x_2)$

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

	0	1	2	3	4	5
0	0			4		
1				4		
2						11

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

	0	1	2	3	4	5
0	0		4	4		
1				4		$(2 + x_1)$
2						$max(11, x_2)$

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

	0	1	2	3	4	5
0	0		4	4		
1				4		6
2						$max(11, 6)$

Example

- Weights: $2, 3, 1$
- Costs: $7, 2, 4$
- Try to solve this for weight 5.

Matrix

	0	1	2	3	4	5
0	0		4	4		
1				4		6
2						11

Variation: What about repetition?

```
int knap(vi &weights, vi &costs, int item, int remW) {
    if (item == N || remW <= 0) return 0;
    int & val = dp[item][remW];
    if (val != -1) return val; // do the DP
    if (remW < weights[item])  // to big, can't take
        return val = knap(dp,weights,costs,item+1,remW);
    else return val =
           max(knap(dp,weights,costs,item, //take
                  remW - weights[item]) + costs[item],
               knap(dp,weights,costs,item+1,remW) ) ;
}
```

0/1 Variation, bottom-up down DP

- Row 0 is “didn’t take anything ever”.
- Row i: for each weight w, find max of
  - $row[i-1][w]$ = “don’t take”
  - $row[i-1][w-w_i]$ = take

Example

- Weights: $2, 3, 1, 4, 3, 2$
- Costs: $7, 2, 4, 3, 4, 5$

Matrix

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						
4	0						
5	0						
6	0						

Example

- Weights: $2, 3, 1, 4, 3, 2$
- Costs: $7, 2, 4, 3, 4, 5$

Matrix

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	7	7	7	7	7
2	0						
3	0						
4	0						
5	0						
6	0						

Example

- Weights: $2, 3, 1, 4, 3, 2$
- Costs: $7, 2, 4, 3, 4, 5$

Matrix

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

Example

- Weights: $2, 3, 1, 4, 3, 2$
- Costs: $7, 2, 4, 3, 4, 5$

Matrix

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	7	7	7	7	7
2	0	0	7	7	9	9	9
3	0	4	7	11	11	13	13
4	0						
5	0						
6	0						

Example

- Weights: $2, 3, 1, 4, 3, 2$
- Costs: $7, 2, 4, 3, 4, 5$

Matrix

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	7	7	7	7	7
2	0	0	7	7	9	9	9
3	0	4	7	11	11	13	13
4	0	4	7	11	11	13	13
5	0						
6	0						

Example

- Weights: $2, 3, 1, 4, 3, 2$
- Costs: $7, 2, 4, 3, 4, 5$

Matrix

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	7	7	7	7	7
2	0	0	7	7	9	9	9
3	0	4	7	11	11	13	13
4	0	4	7	11	11	13	13
5	0	4	7	11	11	13	15
6	0						

Example

- Weights: $2, 3, 1, 4, 3, 2$
- Costs: $7, 2, 4, 3, 4, 5$

Matrix

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	7	7	7	7	7
2	0	0	7	7	9	9	9
3	0	4	7	11	11	13	13
4	0	4	7	11	11	13	13
5	0	4	7	11	11	13	15
6	0	4	7	11	12	16	16

Bottom-Up Code

```
int knap(vi &weights, vi &costs, int items, int maxW) {
    vi dp = vi(maxW+1,0);

    for(int i=0; i<=items; ++i)
        for(int w=max@; w >= weights[i]; w--)
            dp[w] = max(dp[w], costs[i] + dp[w-weights[i]]
    return dp[maxW];
}
```

Variation: Multiple Values

```
int knap(vi &weights, vi &costs, int items, int maxW) {
    vi dp = vi(maxW+1,0);

    for(int i=0; i<=items; ++i) {
        int change = 1;
        while (change) {
            change = 0;
            for(int w=maxW; w >= weights[i]; --w)
                if (dp[w-weights[i]] + costs[i] > dp[w]) {
                    change = 1;
                    dp[w] = costs[i] + dp[w-weights[i]];
                }
        }
    }
    return dp[maxW];
}
```

Discussion

- The general version (where weights can be floats) is weakly NP (and not amenable to DP).
- The bottom-up case is necessary if the total weight can be large (e.g., 10^9).
- The top-down case can be much faster since the DP array will be very sparse.