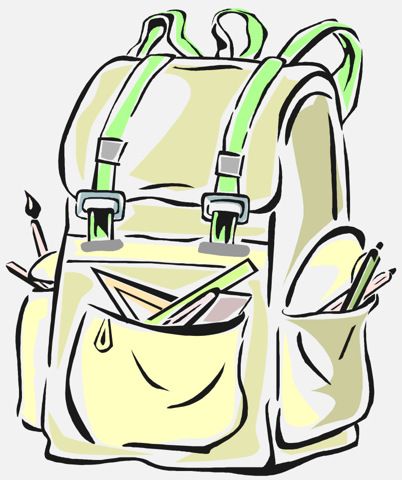
Dynamic Programming II Knapsack

Mark Matthews PhD

0/1 Knapsack Problem



Maximum weight: 5lbs

$0 Dollars

	Weight	Value
Item 1	5 lbs	$60
Item 2	3 lbs	$50
Item 3	2 lbs	$70
Item 4	1 lb	$30

Write a program that selects a subset of items that has a maximum value within a given weight constraint

0/1 means we can not split items

Naive Recursive Solution

```
function KnapSack (n, C)
//base case
if n == 0 or C == 0{
result = 0
}
else if w[n] > C{
result = KnapSack(n-1, C)
}
else {
var1 = KnapSack(n-1, C)
var2 = v[n] + KnapSack(n-1, C -w[n])
result = max{var1, var2}
return result
}
```

Complexity: Exponential!!!

Dynamic Programming Technique Recap

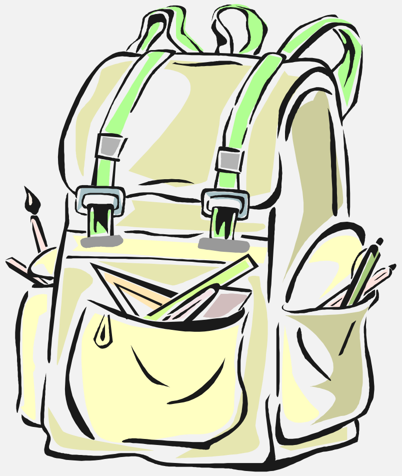
Applies to problems that at first appears to require exponential time to solve.

Key idea: identify solutions to sub-problems so you can find the optimal solution to larger solution.

Characteristics are:

- simple subproblems: subproblems can be defined in terms of a few variables
- subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
- subproblem overlap: subproblems are not independent but overlap

0/1 Knapsack: DP approach



Maximum weight: 5lbs

$0 Dollars

	Weight	Value
Item 1	5 lbs	$60
Item 2	3 lbs	$50
Item 3	2 lbs	$70
Item 4	1 lb	$30

Given a set S of n items with each item I having:

1. W_i - a positive weight
2. V_i - a positive value

Goal: Choose items with max total value but with a weight at most <= W.

Let T denote set of items we take. Our objective is to maximise: $\sum b_i$

Constraint $\sum w \leq W$

0/1 Knapsack: DP approach

S_k: Set of items numbered 1 to k.

Define B[k,w] to be the best selection from S_k with weight at most w

Good news: this has subproblem optimality

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

i.e., the best subset of S_k with weight at most w is either

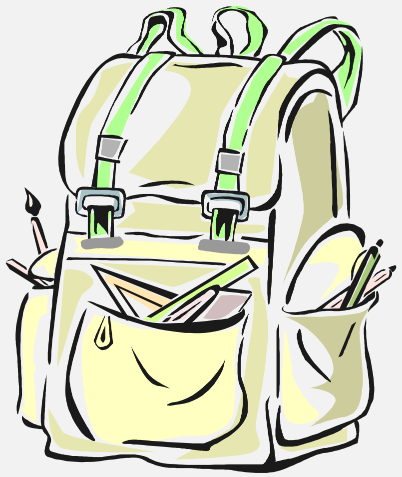
- the best subset of S_{k-1} with weight at most w or
- the best subset of S_{k-1} with weight at most $w-w_k$ plus item k

Dynamic Programming Approach

```
array[n][C] = undefined
def KnapSack (n, C)
If arr[n][C] != undefined: return arr[n]
[C]
//base case
If n == 0 or C == 0{
result = 0
}
else if w[n] > C{
result = KnapSack(n-1, C)
} else {
var1 = KnapSack(n-1, C)
var2 = v[n] + KnapSack(n-1, C - w[n])
result = max{var1, var2}
arr[n][C] = result
return result
}
```

Maximum weight: 5lbs

$0 Dollars



	Weight	Value
Item 1	5 lbs	$60
Item 2	3 lbs	$50
Item 3	2 lbs	$70
Item 4	1 lb	$30

Running time: O(nC).

Bottom up approach

	0	1	2	3	4	5
Item 1 Value $60 Weight: 5lbs						
Item 2 V $50 W 3lbs						
Item 3 V $70 W 4lbs						
Item 4 V $30 W 2lbs						

Bottom up approach

	0	1	2	3	4	5
Item 1 Value $60 Weight: 5lbs	0	0	0	0	0	$60
Item 2 V $50 W 3lbs						
Item 3 V $70 W 4lbs						
Item 4 V $30 W 2lbs						

Bottom up approach

	0	1	2	3	4	5
Item 1 Value $60 Weight: 5lbs	0	0	0	0	0	$60
Item 2 V $50 W 3lbs	0	0	0	$50	$50	$60
Item 3 V $70 W 4lbs						
Item 4 V $30 W 2lbs						

Bottom up approach

	0	1	2	3	4	5
Item 1 Value $60 Weight: 5lbs	0	0	0	0	0	$60
Item 2 V $50 W 3lbs	0	0	0	$50	$50	$60
Item 3 V $70 W 4lbs	0	0	0	$50	$70	$70
Item 4 V $30 W 2lbs						

Bottom up approach

	0	1	2	3	4	5
Item 1 Value $60 Weight: 5lbs	0	0	0	0	0	$60
Item 2 V $50 W 3lbs	0	0	0	$50	$50	$60
Item 3 V $70 W 4lbs	0	0	0	$50	$70	$70
Item 4 V $30 W 2lbs	0	0	$30	$50	$70	

Bottom up approach

	0	1	2	3	4	5
Item 1 Value $60 Weight: 5lbs	0	0	0	0	0	$60
Item 2 V $50 W 3lbs	0	0	0	$50	$50	$60
Item 3 V $70 W 4lbs	0	0	0	$50	$70	$70
Item 4 V $30 W 2lbs	0	0	$30	$50	$70	$80

Bottom up approach

	0	1	2	3	4	5
Item 1 Value $60 Weight: 5lbs	0	0	0	0	0	$60
Item 2 V $50 W 3lbs	0	0	0	$50	$50	$60
Item 3 V $70 W 4lbs	0	0	0	$50	$70	$70
Item 4 V $30 W 2lbs	0	0	$30	$50	$70	$80

Maximum value we get is $80 by choosing items 4 and 2.

Applications

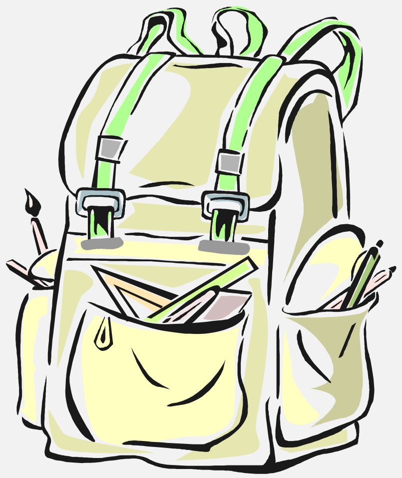
Though simply stated and simply solved, the knapsack problem can be mapped directly, if not used as a prototype for numerous practical problems.

Constrained optimizations are some of the most common puzzles in managing all kinds of operations.

The Dynamic Programming solution to the Knapsack problem is although straightforward has use in many practical applications including:

- a company like Amazon or Fedex trying to pack as much package volume into a transport plane without breaking the weight capacity,
- a sports team's desire to build a team that meets various statistical projections without breaking the salary cap
- an investment fund balancing risk while maximising potential gains

DP Knapsack complexity



Complexity: Where the naive function was exponential in complexity our DP approach gets us O(n C) where n is the number of items and W is the max weight the knapsack can hold.

Bottom-up / Top-down? Can be solved with either bottom-up or top-down DP. There are some minor performance trade-offs depending on the solution you choose, but generally the recommendation in terms of implementation is to use the one with which you are most comfortable conceptually.