Given n items, each with a value v_i and weight w_i, find the maximum value set of items that fit in a knapsack with capacity W. Only whole items can be taken.

Establish the problem.

- specifications – task, input, output, legal solution, optimal solution

Given n items, each with a value v_i and weight w_i, find the maximum value set of items that fit in a knapsack with capacity W. Only whole items can be taken.

Input: n items, each with a value $v_i > 0$ and weight $w_i > 0$; knapsack capacity $W > 0$

Output: a subset of the items

Legal solution: a subset of items with total weight $\leq W$

Optimization goal: maximize total value of items taken

- examples

Identify avenues of attack.

- targets
- approach

Series of choices.

- paradigms and patterns

Paradigm: dynamic programming

Flavor:

Pattern:

process input has a lower branching factor (2)

– for the next item, take it or not?

Define the algorithm.

- size
- generalize / define subproblems
  - partial solution

items picked so far and their total value

  - alternatives – for the next choice

take the next item or not

  - subproblem – solve the rest of the problem, returning the solution for the rest of the problem (only)

The subproblem's task is to solve the rest of the problem in light of the choices made so far — picking of items from amongst those not yet considered as to maximize the total value of those items, given the remaining capacity of the pack.

Task: `knapsack(`S'`,`W'`)` — find the highest-value subset of items in S' whose total weight does not exceed the remaining capacity W' of the knapsack

Input: set S' of items left to consider, remaining (unfilled) capacity of the knapsack W'

Output: total value (of those picked from S')

Legal solution: a subset of items with total weight $\leq W'$

Optimization goal: highest value

- memoization

set of items S'

remaining capacity W' – if W' (and wi weights) are integers…

goal: V[k][w] - S' is items S[k..n-1], remaining capacity w

- base case(s)

when k=n

V[n][w] = 0

- main case

V[k][w] = max { V[k+1][w-w_k]+v_k, V[k+1][w] }

// take

value1 ← knapsack(k+1,w-w_k) + v_k

// don't take

value2 ← knapsack(k+1,w)

return max(value1,value2)

- top level
  - initial subproblem - V[0][W]
  - setup
  - wrapup
- special cases
- algorithm – determine the order of computation and write the loops

```
algorithm knapsack(S,W) –
for w = 0 to W
   V[n][w] = 0
for k = n-1 downto 0
  for w = 0 to W
    V[k][w] = max { V[k+1][w-w_k]+v_k, V[k+1][w] }
return V[0][W]
```

Show termination and correctness.

- termination
  - making progress
  - the end is reached
- correctness
  - establish the base case(s)
  - show the main case
  - final answer

Determine efficiency.

- implementation
- time and space

array is n x W

O(nW) space,

O(1) per subproblem → O(nW) time

- room for improvement