Dynamic Programming Recipe

- **Step 1:** Devise simple recursive algorithm
  - Flavor: make “first choice”, then recursively solve subproblem
- **Step 2:** Write recurrence for optimal value
- **Step 3:** Design bottom-up iterative algorithm

- Weighted interval scheduling: first-choice is binary
- Rod-cutting: first choice has $n$ options
- Subset Sum: first choice is binary, but need to “add a variable” to recurrence

Subset Sum: Problem Formulation

- **Input**
  - Items $1, 2, \ldots, n$
  - Weights $w_i$ for all items (integers)
  - Capacity $W$
- **Goal:** select a subset $S$ whose total weight is as large as possible without exceeding $W$.
- Subset Sum: need to “add a variable” to recurrence

Step 1: Recursive Algorithm, Binary Choice

Let $O$ be optimal solution on items 1 through $j$. Is $j \in O$ or not?

```
SubsetSum(j)
if $j = 0$ then return 0
▷ Case 1: $j \notin O$
$v = SubsetSum(j - 1)$
▷ Case 2: $j \in O$
if $w_j \leq W$ then
  $v = \max(v, w_j + SubsetSum(j - 1))$
 ▷ else skip b/c can’t fit $w_j$
return $v$
```

```
Dynamic Programming Recipe

- **Step 1:** Devise simple recursive algorithm
  - Flavor: make “first choice”, then recursively solve subproblem
- **Step 2:** Write recurrence for optimal value
- **Step 3:** Design bottom-up iterative algorithm

- Weighted interval scheduling: first-choice is binary
- Rod-cutting: first choice has $n$ options
- Subset Sum: first choice is binary, but need to “add a variable” to recurrence

Subset Sum: Problem Formulation

- **Input**
  - Items $1, 2, \ldots, n$
  - Weights $w_i$ for all items (integers)
  - Capacity $W$
- **Goal:** select a subset $S$ whose total weight is as large as possible without exceeding $W$.
- Subset Sum: need to “add a variable” to recurrence

Step 1: Recursive Algorithm, Binary Choice

Let $O$ be optimal solution on items 1 through $j$. Is $j \in O$ or not?

```
SubsetSum(j)
if $j = 0$ then return 0
▷ Case 1: $j \notin O$
$v = SubsetSum(j - 1)$
▷ Case 2: $j \in O$
if $w_j \leq W$ then
  $v = \max(v, w_j + SubsetSum(j - 1))$
  ▷ else skip b/c can’t fit $w_j$
return $v$
```
SubsetSum(j)
if j = 0 then return 0
v = SubsetSum(j − 1) ▷ Case 1: j /∈ O
if wj ≤ W then ▷ Case 2: j ∈ O
v = max(v, wj + SubsetSum(j − 1))
return v

Step 2: Recurrence
Let OPT(j, w) be the maximum weight of any subset of items {1, . . . , j} that does not exceed w.

OPT(j, w) = \begin{cases} 
  OPT(j − 1, w) & w_j > w \\
  \max \left\{ \begin{array}{l}
  OPT(j − 1, w) \\
  w_j + OPT(j − 1, w − w_j)
  \end{array} \right\} & w_j \leq w 
\end{cases}

Base case: OPT(0, w) = 0 for all w = 0, 1, . . . , W.

Questions
- Do we need a base case for OPT(j, 0)? No
- What is overall optimum to original problem? OPT(n, W)

Step 1: Recursive Algorithm, Add a Variable
Find value of optimal solution O on items {1, 2, . . . , j} when the remaining capacity is w.

SubsetSum(j, w)
if j = 0 then return 0
v = SubsetSum(j − 1, w) ▷ Case 1: j /∈ O
if wj ≤ W then ▷ Case 2: j ∈ O
v = max(v, wj + SubsetSum(j − 1, w − wj))
return v

From Recurrence to Iterative (“Turn the Crank”)
OPT(j, w) = \begin{cases} 
  OPT(j − 1, w) & w_j > w \\
  \max \left\{ \begin{array}{l}
  OPT(j − 1, w) \\
  w_j + OPT(j − 1, w − w_j)
  \end{array} \right\} & w_j \leq w 
\end{cases}

What size memoization array? $M[j, w]$ for all values of j and w.
M[0 . . . n, 0 . . . W]

What order to fill entries? base case first; RHS before LHS for j from 0 → n, for w from 0 → W.

Which entry stores solution to overall problem? Want OPT(n, W): stored in M[n, W]
Step 3: Iterative Algorithm

SubsetSum(n,W)
Initialize array M[0..n, 0..W]
Set M[0, w] = 0 for w = 0, . . . , W
for j = 1 to n do
  for w = 1 to W do
    if wj > w then M[j, w] = M[j − 1, w]
    else M[j, w] = max(M[j − 1, w], wj + M[j − 1, w−wj])
  return M[n, W]

Running Time? Θ(nW).

Polynomial vs. Pseudo-polynomial

If numbers have m digits, input size is Θ(nm), runtime is Θ(n10m).

► Polynomial time: polynomial in input size (nm)
► Pseudo-polynomial: polynomial in number of items (n) and magnitude of numbers (10m)

For numeric problems, input size is log of magnitude of the numbers. Poly-time algorithm should be polynomial in n and log W.

Subset Sum:

► Our solution is pseudo-polynomial
► No polynomial algorithm is known

Subset Sum

Source: https://www.xkcd.com/287/
Knapsack Problem

Same as subset sum, but now items have value in addition to weight

Input

▶ Items 1, 2, . . . , n
▶ Weights $w_i$ for all items (integers)
▶ Values $v_i$ for all items (integers)
▶ Capacity $W$

Goal: select subset $S$ whose total value is as large as possible without exceeding $W$.

Clicker

Recall subset-sum recurrence:

$$\text{OPT}(j, w) = \begin{cases} 
\text{OPT}(j-1, w), & w_j > w \\
\max \{ \text{OPT}(j-1, w), w_j + \text{OPT}(j-1, w-w_j) \}, & w_j \leq w 
\end{cases}$$

How should the blue term be rewritten for the knapsack recurrence?

A. $w_j + \text{OPT}(j-1, w-w_j)$
B. $w_j + \text{OPT}(j-1, w-v_j)$
C. $v_j + \text{OPT}(j-1, w-v_j)$
D. $v_j + \text{OPT}(j-1, w-w_j)$

Clicker

Does our knapsack solution still work if the weights and/or values are real numbers instead of integers?

A. It still works if both the values and weights are real numbers.
B. It works if values are real numbers but weights are integers.
C. It works if weights are real numbers but values are integers.
D. It does not work if either the weights or values are real numbers.

Fractional knapsack problem allows partial objects (think grains, sand, fluid). Has simple greedy solution: choose highest value per weight.