Dynamic Programming Recipe

- **Step 1**: Devise simple recursive algorithm
  - Flavor: make “first choice”, then recursively solve remaining part of the problem
- **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: “add a variable” (one more dimension)

Subset Sum: Problem Formulation

- **Input**
  - Items 1, 2, ..., \( n \)
  - 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?
- \( \text{SubsetSum}(j) \)
  - if \( j = 0 \) then return 0
  - \( v = \text{SubsetSum}(j - 1) \)
    - Case 1: \( j \not\in O \)
      - \( v = \text{SubsetSum}(j - 1) \)
    - Case 2: \( j \in O \)
      - if \( w_j \leq W \) then
        - \( v = \max(v, w_j + \text{SubsetSum}(j - 1)) \)
      - else skip b/c can’t fit \( w_j \)
      - return \( v \)

Clicker Question

\[ \text{SubsetSum}(j) \]
\[
\text{if } j = 0 \text{ then return 0} \\
v = \text{SubsetSum}(j - 1) \\
\text{if } w_j \leq W \text{ then} \\
v = \max(v, w_j + \text{SubsetSum}(j - 1)) \\
\text{return } v
\]

Is there a problem in Case 2?
- A. No, it is correct.
- B. Yes, you need to consider that the \( j \)th may be selected multiple times.
- C. Yes, if we take item \( j \), the remaining capacity changes.

Second call to \( \text{SubsetSum}(j - 1) \) no longer has capacity \( W \).
Solution: must add extra parameter (problem dimension)

Step 1: Recursive Algorithm, Add a Variable

- Find value of optimal solution \( O \) on items \( \{1, 2, ..., j\} \)
  - when the remaining capacity is \( w \)
- \( \text{SubsetSum}(j, w) \)
  - if \( j = 0 \) then return 0
  - \( v = \text{SubsetSum}(j - 1, w) \)
    - Case 1: \( j \not\in O \)
      - \( v = \text{SubsetSum}(j - 1, w) \)
    - Case 2: \( j \in O \)
      - if \( w_j \leq w \) then
        - \( v = \max(v, w_j + \text{SubsetSum}(j - 1, w - w_j)) \)
      - return \( v \)
Recurrence

▶ Let $OPT(j, w)$ be the maximum weight of any subset of items $\{1, \ldots, j\}$ that does not exceed $w$.

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

▶ Base case: $OPT(0, w) = 0$ for all $w = 0, 1, \ldots, W$.

Questions

▶ Do we need a base case for $OPT(j, 0)$? No.

▶ What is overall optimum to original problem? $OPT(n, W)$

Step 3: Iterative Algorithm

SubsetSum($n, W$)

Initialize array $M[0..n, 0..W]$

Set $M[0, w] = 0$ for $w = 0, \ldots, W$

for $j = 1$ to $n$ do

for $w = 1$ to $W$ do

if $w_j > w$ then $M[j, w] = M[j - 1, w]$

else $M[j, w] = \max(M[j - 1, w], w_j + M[j - 1, w - w_j])$

return $M[n, W]$

Can we switch inner loop ($j$) and outer loop ($w$)? Yes.

▶ Running Time? $\Theta(nW)$.

Clicker Question

for $j = 1$ to $n$ do

for $w = 1$ to $W$ do

if $w_j > w$ then $M[j, w] = M[j - 1, w]$

else $M[j, w] = \max(M[j - 1, w], w_j + M[j - 1, w - w_j])$

Suppose we have $n$ items, and the capacity $W$ and weights $w_j$ each have $m$ decimal digits. Then the running time is:

A. $\Theta(n m)$

B. $\Theta(n \log_{10} m)$

C. $\Theta(n 10^m)$

D. $\Theta(10^m)$

Polynomial vs. Pseudo-polynomial

If numbers have $m$ digits, input size is $\Theta(n m)$, runtime is $\Theta(n 10^m)$.

▶ Polynomial time: polynomial in input size ($nm$)

▶ Pseudo-polynomial: polynomial in number of items ($n$) and magnitude of numbers ($10^m$)

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

0–1 Knapsack Problem

Introduce an additional parameter, $value$

▶ Input

▷ Items 1, 2, \ldots, $n$

▷ Weights $w_i$ for all items (integers)

▷ Values $v_i$ for all items (integers)

▷ Capacity $W$

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

▶ Does the solution change?
Recall subset-sum recurrence: \( \text{OPT}(j, w) \) 
\[
\text{OPT}(j, w) = \begin{cases} 
\text{OPT}(j-1, w) & w_j > w \\
\max \{ \text{OPT}(j-1, w), w_j + \text{OPT}(j-1, w - w_j) \} & w_j \leq w 
\end{cases}
\]

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. \( v_j + \text{OPT}(j-1, w-w_j) \)

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.

Weights are the second dimension: can index on ints, not reals. (B)

Real weights ⇒ consider all \( 2^n \) choices.

Fractional knapsack problem: allows partial objects
(think grains, sand, . . .)

Simple greedy solution: choose highest value per weight