COMPSCI 311: Introduction to Algorithms

Lecture 14: Dynamic Programming

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

18 March 2019

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 $1, 2, \ldots, n$
  - 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.
- Example: $W = 6$, items: $w_1 = 2, w_2 = 2, w_3 = 3$.

Step 1: Recursive Algorithm, Binary Choice

- Let O be optimal solution on items 1 to j.
  Is $j \in O$ or not?
- SubsetSum(j)

```
if j = 0 then return 0
▷ Case 1: j ∉ O
vmax = SubsetSum(j-1)
▷ Case 2: j ∈ O
if w_j ≤ W then                          ▷ else skip, can't fit w_j
    vmax = max(vmax, w_j + SubsetSum(j-1) ?)
return vmax
```

Clicker Question

SubsetSum(j)

```
if j = 0 then return 0
vmax = SubsetSum(j-1)                    ▷ Case 1: j ∉ O
if w_j ≤ W then                          ▷ Case 2: j ∈ O
    vmax = max(vmax, w_j + SubsetSum(j-1)?)
return vmax
```

Is there a problem in Case 2?

A. No. It is correct

B. Yes, must consider selecting j^{th} item multiple times

C. Yes, if we take item j, the remaining capacity changes

Second call to 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 $\{1, 2, \ldots, j\}$ when the remaining capacity is w
- SubsetSum(j,w)

```
if j = 0 then return 0
▷ Case 1: j ∉ O
vmax = SubsetSum(j − 1, w)
▷ Case 2: j ∈ O
if w_j ≤ w then
    vmax = max(vmax, w_j + SubsetSum(j − 1, w − w_j))
return vmax
```

Recurrence

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

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

- Base case: $\mathrm{OPT}(0, w) = 0$ for all $w = 0, 1, \ldots, W$.
- Questions
  - Do we need a base case for $\mathrm{OPT}(j, 0)$? No
  - What is overall optimum to original problem? $\mathrm{OPT}(n, W)$

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 total capacity has m decimal digits. Then the complexity is:

A. $\Theta(nm)$

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

C. $\Theta(n10^m)$

D. $\Theta(10^{nm})$

The capacity is $W \simeq 10^m$. The complexity is $\Theta(n10^m)$.

Polynomial vs. Pseudo-polynomial

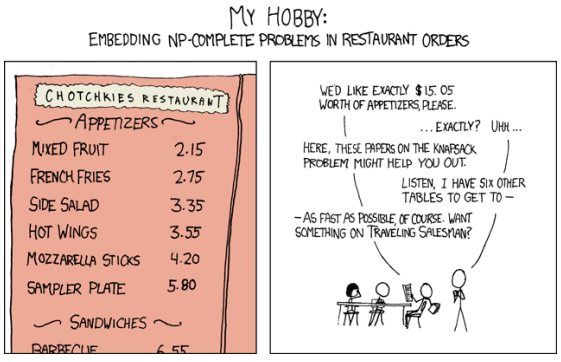
If numbers have m digits, input size is $\Theta(nm)$, runtime is $\Theta(n10^m)$.

- **Polynomial time** means polynomial in input size (nm)

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 is **pseudo-polynomial**: polynomial in number of items (n) and *magnitude* of numbers ($W \simeq 10^m$)
  thus exponential in input size of numbers (digit count m)
- No polynomial-time algorithm is known
  discuss again with NP-completeness

Subset Sum



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

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 ?

Clicker Question

Recall recurrence for subset sum: $\mathrm{OPT}(j, w)$

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

What is the correct change for the blue term in the recurrence?

A. $w_j + \mathrm{OPT}(j-1, w-w_j)$

B. $w_j + \mathrm{OPT}(j-1, w-v_j)$

C. $v_j + \mathrm{OPT}(j-1, w-v_j)$

D. $v_j + \mathrm{OPT}(j-1, w-w_j)$

Optimum *adds values* v_j, *subtracts weights* w_j from capacity (D)

Clicker Question

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 weights are real numbers but values are integers.

C. It works if values are real numbers but weights are integers.

D. It does not work if either weights or values are real numbers.

Weights are the second dimension: can index on ints, not reals. (C)
Real weights $\Rightarrow$ consider all 2^n choices.

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