CS3000: Algorithms & Data
Paul Hand

Lecture 10:

- Dynamic Programming: Knapsack Problems

Feb 11, 2019

Dynamic Programming

Dynamic Programming Recap

- Express the optimal solution as a **recurrence**
  - Identify a small number of subproblems
  - Relate the optimal solution on subproblems

- Efficiently solve for the **value** of the optimum
  - Simple implementation is exponential time
  - Top-Down: store solution to subproblems
  - Bottom-Up: iterate through subproblems in order

- Find the **solution** using the table of **values**

Dynamic Programming: Knapsack Problems

Tug-of-War

- We have n students with weights $w_1, \ldots, w_n \in \mathbb{N}$, need to split as evenly as possible into two teams
  - e.g. $\{21,42,33,52\}$

The Knapsack Problem

- **Input:** n items for your knapsack
  - value v_i and a weight $w_i \in \mathbb{N}$ for n items
  - capacity of your knapsack $T \in \mathbb{N}$
- **Output:** the most valuable subset of items that fits in the knapsack
  - Subset $S \subseteq \{1, \ldots, n\}$
  - Value $V_S = \sum_{i \in S} v_i$ as large as possible
  - Weight $W_S = \sum_{i \in S} w_i$ at most T

- **SubsetSum:** $v_i = w_i$

Solve this Knapsack by hand

- Total Weight T = 10
  What collection of items maximizes value with total weight of at most T?

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

Is Dynamic Programming Necessary?

- Want to maximize **bang-for-buck**, right?
  - Items with large $\frac{v_i}{w_i}$ seem like good choices
  - Design a Knapsack problem where selecting items in decreasing order of bang-for-buck (subject to the weight constraint) gives the incorrect result.

Dynamic Programming

- Let $O \subseteq \{1, \ldots, n\}$ be the **optimal** subset of items for a knapsack of size T
- **Case 1:** $n \notin O$
- **Case 2:** $n \in O$

Dynamic Programming

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \dots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
- **Case 2:** $j \in O_{j,S}$

Dynamic Programming

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \dots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - Use opt. solution for items 1 to j-1 and size S
- **Case 2:** $j \in O_{j,S}$
  - Use i + opt. solution for items 1 to j-1 and size $S - w_j$

Dynamic Programming

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $i \notin O_{j,S}$
  - Use opt. solution for items 1 to j-1 and size S
- **Case 2:** $i \in O_{j,S}$
  - Use i + opt. solution for items 1 to j-1 and size $S - w_j$

Recurrence:

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

Base Cases:

$$\mathrm{OPT}(j, 0) = \mathrm{OPT}(0, S) = 0$$

Activity

$$\mathrm{OPT}(j,S) = \begin{cases} \max\{OPT(j-1,S), v_j + OPT(j-1, S-w_j)\} & \text{if } w_j \le S \\ OPT(j-1,S) & \text{if } w_j > S \end{cases}$$

- Input: $T = 8, n = 3$
  - $w_1 = 1 , v_1 = 4$
  - $w_2 = 3 , v_2 = 5$
  - $w_3 = 5 , v_3 = 8$

items

3									
2									
1									
0									
-	0	1	2	3	4	5	6	7	8

capacities

Knapsack ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n,T):
  M[0,S] ← 0, M[j,0] ← 0

  for (j = 1,…,n):
    for (s = 1,…,T):
      if (w_j > S): M[j,S] ← M[j-1,S]
      else: M[j] ← max{M[j-1,S],v_j + M[j-1,S-w_j]}

  return M[n,T]
```

Activity: What is the runtime of this algorithm?

How much memory does it take?

Dynamic Programming

- Let $\boldsymbol{O_{j,S}}$ be the **optimal subset of items** $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - Use opt. solution for items 1 to j-1 and size S
- **Case 2:** $j \in O_{j,S}$
  - Use i + opt. solution for items 1 to j-1 and size $S - w_j$

Filling the Knapsack

```
// All inputs are global vars
// M[0:n,0:T] contains solutions to subproblems
FindSol(M,n,T):
  if (n = 0 or T = 0): return Ø
  else:
    if (w_n > T): return FindSol(M,n-1,T)
    else:
      if (M[n-1,T] > v_n + M[n-1,T-w_n]):
        return FindSol(M,n-1,T)
      else:
        return {n} + FindSol(M,n-1,T-w_n)
```

Knapsack Wrapup

- Can solve knapsack problems in time/space $O(\quad\quad)$
  - Brute force algorithms run in time $O(2^n)$
- Dynamic Programming:
  - Decide whether the n^th item goes in the knapsack
- Can solve subset-sum and tug-of-war