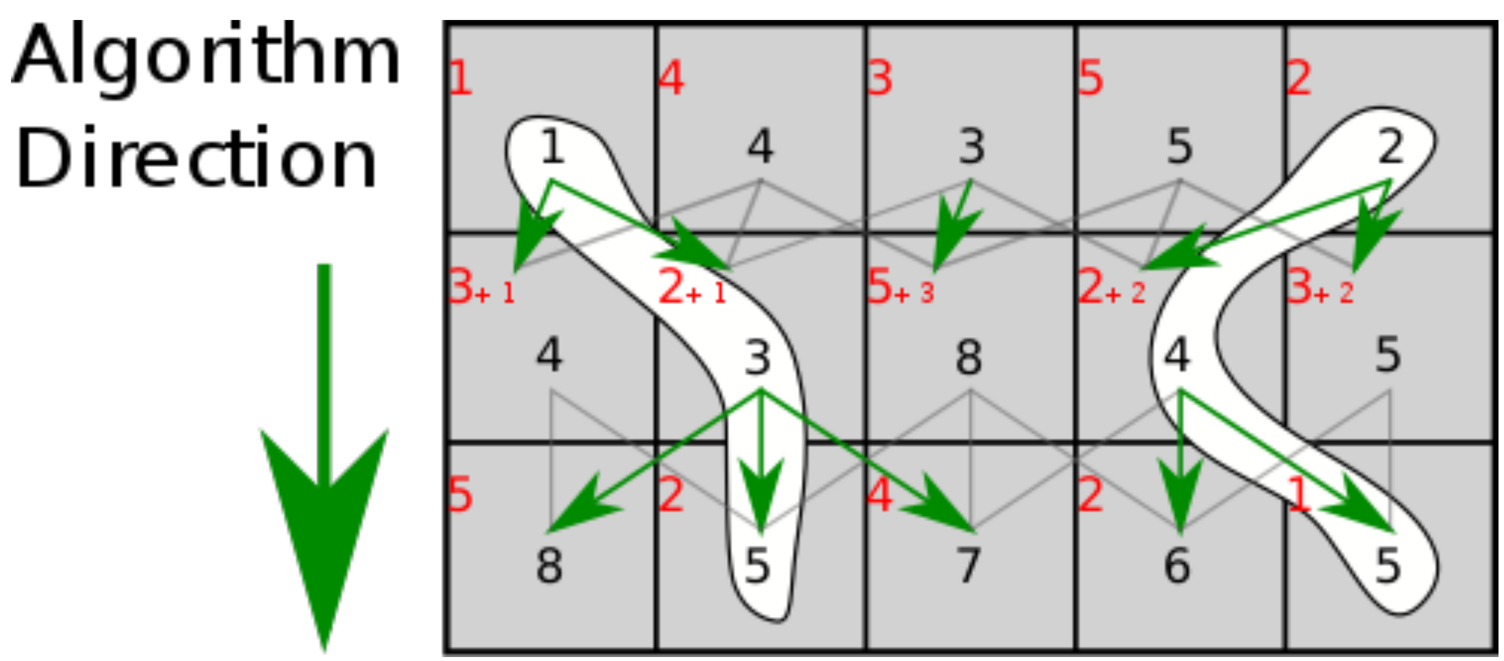
Dynamic Programming III: Knapsack Problem



Admin

- Assignment 5 is due a day early
    - Office hours: M 2.00-3.30pm, T 3-5 pm
    - TA hours: today 3.30-5, 9-10 pm
    - TA hours: tomorrow 5-10 pm
    - Late work may not be graded in time
- **Midterm** next Friday **(April 2);** no class
- Exam be released 10.00 am Friday; can be taken in any 24 hour period between **10.00 am Friday to 10.00 am Sunday**

Knapsack Problem

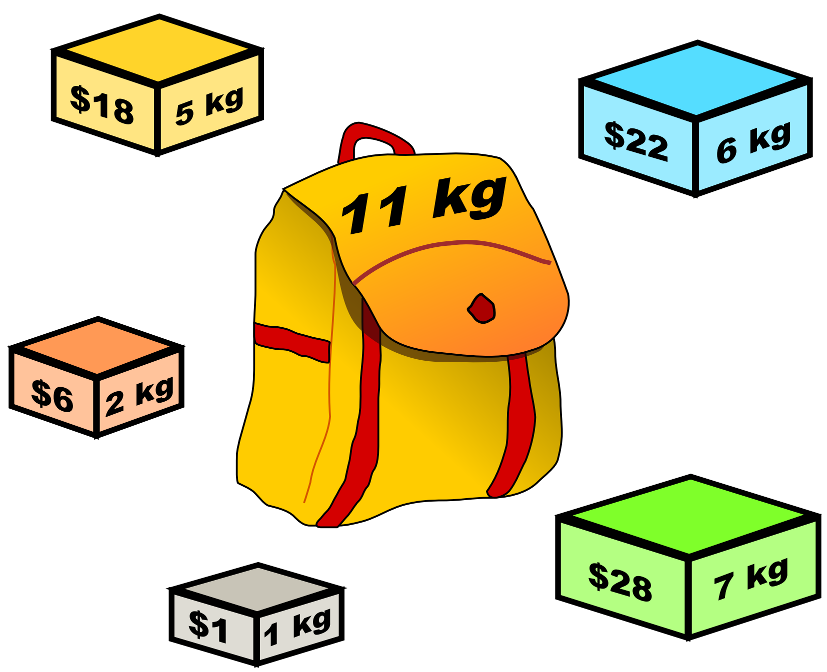
Reading: Chapter 6.4, KT

Knapsack Problem

- **Problem**. Pack a knapsack to maximize total value
- There are n items, each with weight w_i and value v_i
- Knapsack has total capacity C
- For any set of items T they fit in the Knapsack iff
    - **Capacity constraint**: $\sum_{i \in T} w_i \leq C$
- **Goal**: Find subset S of items that fit in the knapsack (satisfy the capacity constraint) **and maximize** the total value $\sum_{i \in S} v_i$
- **Assumption**. All weights and values are non-negative integers

Knapsack Problem

- Does greedily picking the highest value item work?
- Does greedily picking the lowest weight item work?

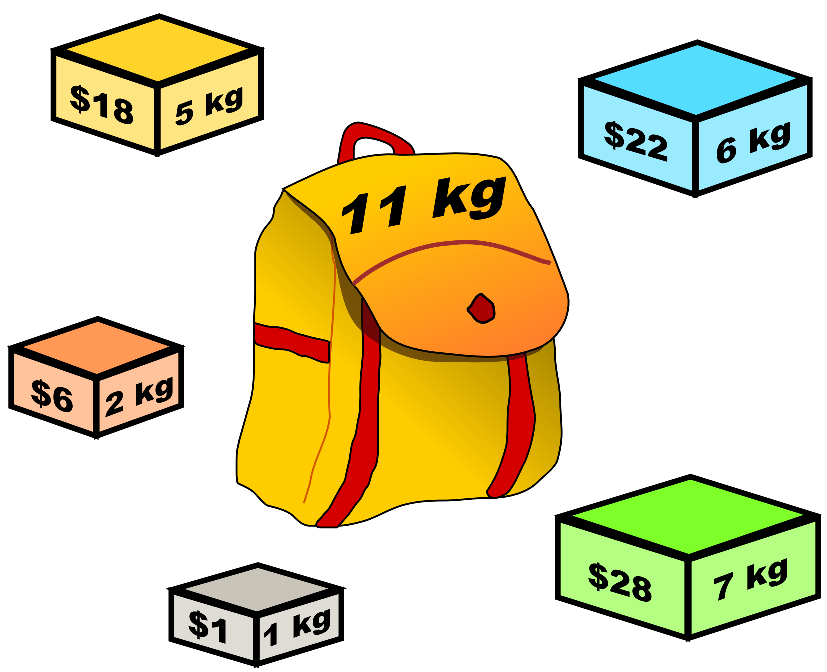


i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

Knapsack instance
(weight limit C = 11 kg)

Knapsack Problem

- Example (Knapsack capacity C = 11)
  - Optimal: {3, 4} has value $40 (and weight 11)

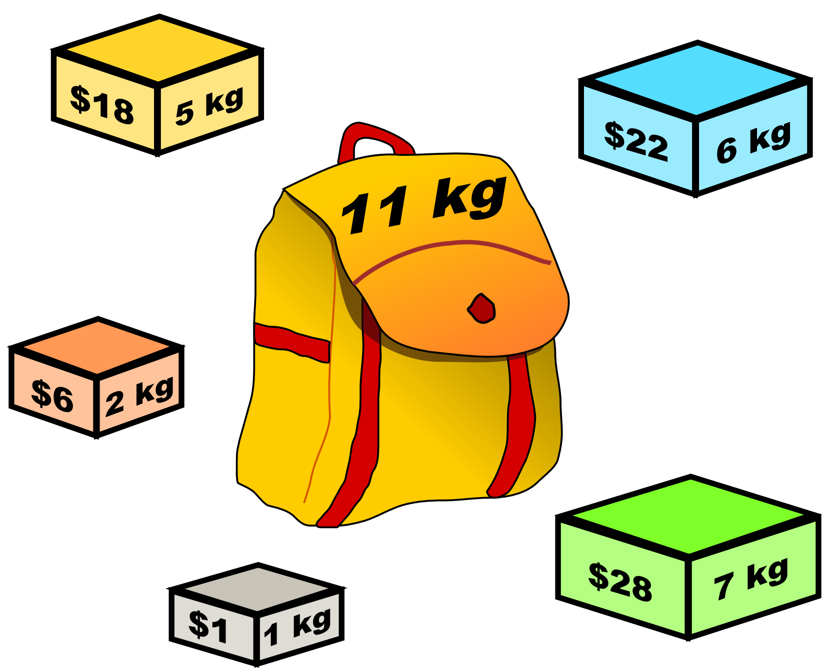


i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

knapsack instance
(weight limit W = 11)

Exponential Possibilities

- Given S items, how many subsets of items are there total?
    - 2^S: exponential possibilities
- Dynamic programming trades of space for time, and through memoization gives an efficient solution



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit W = 11)

Towards a Subproblem

- Idea 1: Keep track of capacity
    - **Subproblem**. Let $T[c]$ denote the value of the optimal solution that uses capacity $\leq c$.
- Optimal solution: $T[C]$
- How do come up with a recurrence?
- Not obvious with just capacities

Subproblems and Optimality

- When items are selected we need to fill the remaining capacity optimally
- Subproblem associated with a given remaining capacity can be solved in different ways

Partial Selection #1 Partial Selection #2

- In both cases, remaining capacity: 11 but items left are different

Subproblem:
Optimal Substructure

Subproblem

- **Subproblem**

$\boxed{\text{OPT}(j, c)\text{: value of optimal solution using items } \{1,2,\ldots,j\} \text{ with total capacity } \leq c\text{, for } 0 \leq j \leq n,\ \ 0 \leq c \leq C}$

- **Final answer**

$\boxed{\text{OPT}(n, C)}$

Base Cases

- Let us think about which rows/columns can we fill initially
- OPT$(i,0)$: Value of optimal solution that uses first i items and total capacity at most 0
- OPT$(0,c)$: Value of optimal solution that uses zero items and total capacity at most c

$$\boxed{\begin{array}{l} \text{OPT}(i,\ 0) = 0 \ \text{ for } 0 \leq i \leq n \\[2ex] \text{OPT}(0,\ c) = 0 \text{ for } 0 \leq c \leq C \end{array}}$$

Optimal Substructure

- OPT(i, c): Imagine the optimal solution that uses items $\{1,2,\ldots,i\}$ and capacity at most c
- What are the possibilities for the last ith item:
  - Either it is in the optimal solution or not, we consider both cases
- **Case 1.** Suppose it is **not** in the optimal solution, what is the optimal way to solve the remaining problem?
  - OPT(i, c) $=$ OPT$(i-1, c)$

Optimal Substructure

- OPT(i, c): Imagine the optimal solution that uses items $\{1,2,\ldots,i\}$ and capacity at most c
- What are the possibilities for the last ith item:
  - Either it is in the optimal solution or not, we consider both cases
- **Case 2.** Suppose it **is** in the optimal solution, what is the recurrence of the optimal solution?
  - OPT(i, c) $= v_i +$ OPT$(i-1,\ c-w_i)$ (assuming $c \geq w_i$, if not, this case is not possible)

Final Recurrence

- For $1 \leq i \leq n$ and $1 \leq c \leq C$, we have:

$$\boxed{\begin{aligned}&\text{OPT}(i, c) = \\ &\max\{\text{OPT}(i-1,\ c),\ v_i + \text{OPT}(i-1,\ c - w_i)\}\end{aligned}}$$

- **Memoization structure**: We store OPT$[i, c]$ values in a 2-D array or table using space $O(nC)$
- **Evaluation order**: In what order should we fill in the table?
    - Row-major order (row-by-row)

Running Time

- Takes $O(1)$ to fill out a cell, $O(nC)$ total cells
- Is this polynomial? By which I mean polynomial in the *size of the input*
- How large is the input to knapsack?
  - Store n items, plus need to store C
  - $O(n + \log C)$
- Is $O(nC)$ polynomial?
  - No!
  - "Pseudopolynomial" - polynomial in the *value* of the input
- To think about: does this work if the weights are not integers?

Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure
- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem
- **State the base case(s)**. The subproblem thats so small we know the answer to it!
- **State the final answer.** (In terms of the subproblem)
- **Choose a memoization data structure.** Where are you going to store already computed results? (Usually a table)
- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which ones? Using these dependencies, identify an evaluation order
- **Analyze space and running time.** As always!

Partitioning Books

Reading: Linked on GLOW

Dynamic Programming Practice

- Suppose we have to scan through a shelf of books, and this task can be split between k workers
- We do not want to reorder/rearrange the books, so instead we divide the shelf into k regions
- Each worker is assigned one of the regions
- What is the fairest way to divide the shelf up?

DP: Dividing Work

- Suppose we have to scan through a shelf of books, and this task can be split between k workers
- We do not want to reorder/rearrange the books, so instead we divide the shelf into k regions
- Each worker is assigned one of the regions
- What is the fairest way to divide the shelf up?
- If the books are equal length, we can just give each worker the same number of books
- What if books are not equal size?
    - How can we find the fairest partition of work?

The Linear Partition Problem

- **Input.** A input arrangement S of nonnegative integers $\{s_1, \ldots, s_n\}$ and an integer k
- **Problem.** Partition S into k ranges such that the maximum sum over all the ranges is minimized
- **Example.**
  - Consider the following arrangement

    100 200 300 400 500 600 700 800 900

  - Suppose $k = 3$, where should we partition to minimize the maximum sum over all ranges?

    100 200 300 400 500 | 600 700 | 800 900

Optimal Substructure

- Notice that the kth partition starts after we place the $(k-1)$st “divider”
- Let us try to construct an optimal solution. Where can we place the *last* divider?
  - Between some elements, suppose between ith and $(i+1)$st element where $1 \leq i \leq n-1$
  - What is the cost of placing the last divider here? Max of:
    - Cost of the last partition $\sum_{j=i+1}^{n} s_j$
    - Cost of the optimal way to partition the elements to the “left” — this is a smaller version of the same problem!
- **Question:** Can you come up with the **subproblem** for the dynamic program?

Dividing Work: DP Algorithm

- **Subproblem**. $M[i, j]$ be the minimum cost over all partitions of first i books into j partitions, $1 \leq i \leq n$, $1 \leq j \leq k$
- **Base cases**.
  - $M[1, j] = s_1$ for all $1 \leq j \leq k$
  - $M[i, 1] = \sum_{t=1}^{i} s_t$ for all $1 \leq i \leq n$
- **Recurrence**.
  - Dictates how we go from one subproblem to the next
  - Now we have a two dimensional table so we also need to think about which order to go in (what the dependencies are…)

Dividing Work: DP Algorithm

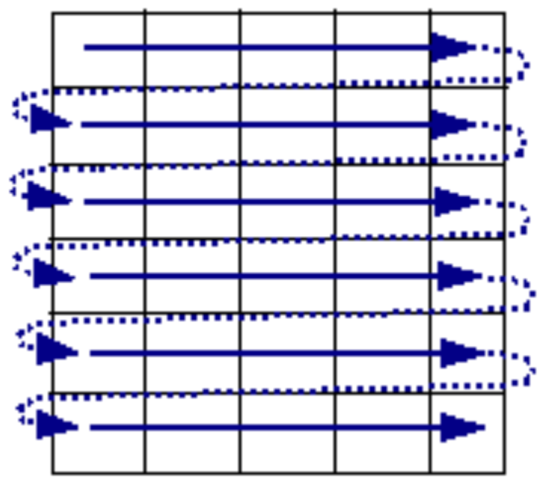
- **Subproblem**. $M[i, j]$ be the minimum cost over all partitions of first i books into j partitions, $1 \leq i \leq n$, $1 \leq j \leq k$
- **Base cases**.
  - $M[1, j] = s_1$ for all $1 \leq j \leq k$
  - $M[i, 1] = \sum_{t=1}^{i} s_t$ for all $1 \leq i \leq n$
- **Recurrence**. $M[i, j] = \min_{1 \leq i' \leq i} \max\{M(i', j-1), \sum_{t=i'+1}^{i} s_t\}$
- **Final solution**. $M[n, k]$
- **Memoization structure**. Two-dimensional array.
- **Evaluation order**. ?

Evaluation Order

- What do we need filled in so that we can fill in $M[i, j]$?
- For all $i' < i$, need $M[i', j - 1]$
- Plan: fill in all $M[i,1]$, then all $M[i,2]$ (in increasing order of i), then all $M[i,3]$, and so on
- Let's draw out M where each value of j is a row of M

Dividing Work: Final Pieces

- Evaluation order.
    - To fill out one cell, we need to take min over the values to the left in the previous row
    - Thus, we fill out rows one-by-one
    - Called row major order
- Running time?
    - Size of table: $O(k \cdot n)$
    - How long to compute a single cell?
        - Depends on n other cells
    - $O(n^2 \cdot k)$ time



Row-major order

Running Time

- Running time
  - Size of table: $O(k \cdot n)$
  - How long to compute a single cell?
    - Depends on n other cells
  - $O(n^2 \cdot k)$ time
- Is this a polynomial running time?
- How big can k get?
  - At most n non-empty partitions of n elements
  - $O(n^3)$ algorithm in the worst case

Acknowledgments

- Some of the material in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)