Week 9 - Wednesday

COMP 4500

Last time

- What did we talk about last time?
- Dynamic programming
- Segmented least squares

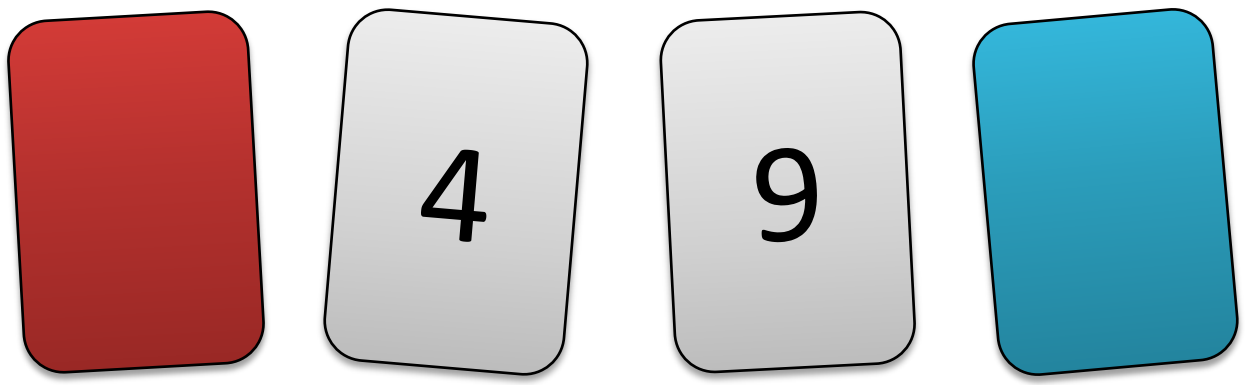
Questions?

Assignment 5

Logical warmup

- A deck of cards has positive integers on one side and either red or blue on the other side.
- Consider the following hypothesis:

  If a card shows an even number on one side, it's red on the other side.
- Which cards must you turn over to test this hypothesis?



Three-Sentence Summary of Subset Sum and Knapsack

Subset Sum

Subset sum

- Let's say that we have a series of ***n*** jobs that we can run on a single machine
- Each job ***i*** takes time w_i
- We must finish all jobs before time ***W***
- We want to keep the machine as busy as possible, working on jobs until as close to ***W*** as we can

Other ways of looking at it

- This fundamental problem can be looked at in many ways:
  - Try to fill up a knapsack with objects where each has weight w_i and the knapsack can only hold W
  - Take a set of numbers and find a subset whose sum is as close as possible to a target value

Greedy doesn't work

- No one knows a natural greedy solution for this problem
- Always take the biggest that fits doesn't work:
  - Consider set {$W/2 + 1$, $W/2$, $W/2$}
- Always take the smallest that still fits doesn't work:
  - Consider set {1, $W/2$, $W/2$}

Another algorithm that doesn't work

- Like before, we could consider the optimal value OPT(***n***) of all jobs up to ***n***
- If ***n*** is not in the solution, OPT(***n***) = OPT(***n*** – 1)
  - So far, so good
- If ***n*** is in the solution …
  - Crap.
  - We don't get very much information about what other jobs can't be in the solution
- We need to add more information

Adding another variable

- We want to think about weights
- If we have job ***n*** in the solution, then there will only be $\boldsymbol{W - w_n}$ capacity left
- Let's assume that all the weights are integers
- Then, we could define a class of optimal values:

$$\mathrm{OPT}(i, w) = \max_S \sum_{j \in S} w_j \text{ , such that } \sum_{j \in S} w_j \leq w$$

- We're going to store optimal values for sets of jobs {1, 2,…, ***i***} that do not exceed weight ***w***, for all possible jobs ***i*** and weights ***w***

A new recurrence

- If job n is not in the optimal set, OPT(n, W) = OPT($n - 1$, W)
- If job n is in the optimal set, OPT(n, W) = w_n + OPT($n - 1$, $W - w_n$)
- We can make the full recurrence for all possible weight values:
  - If $w < w_i$, then OPT(i, w) = OPT($i - 1$, w)
  - Otherwise, OPT(i, w) = max(OPT($i - 1$, w), w_i + OPT($i - 1$, $w - w_i$))

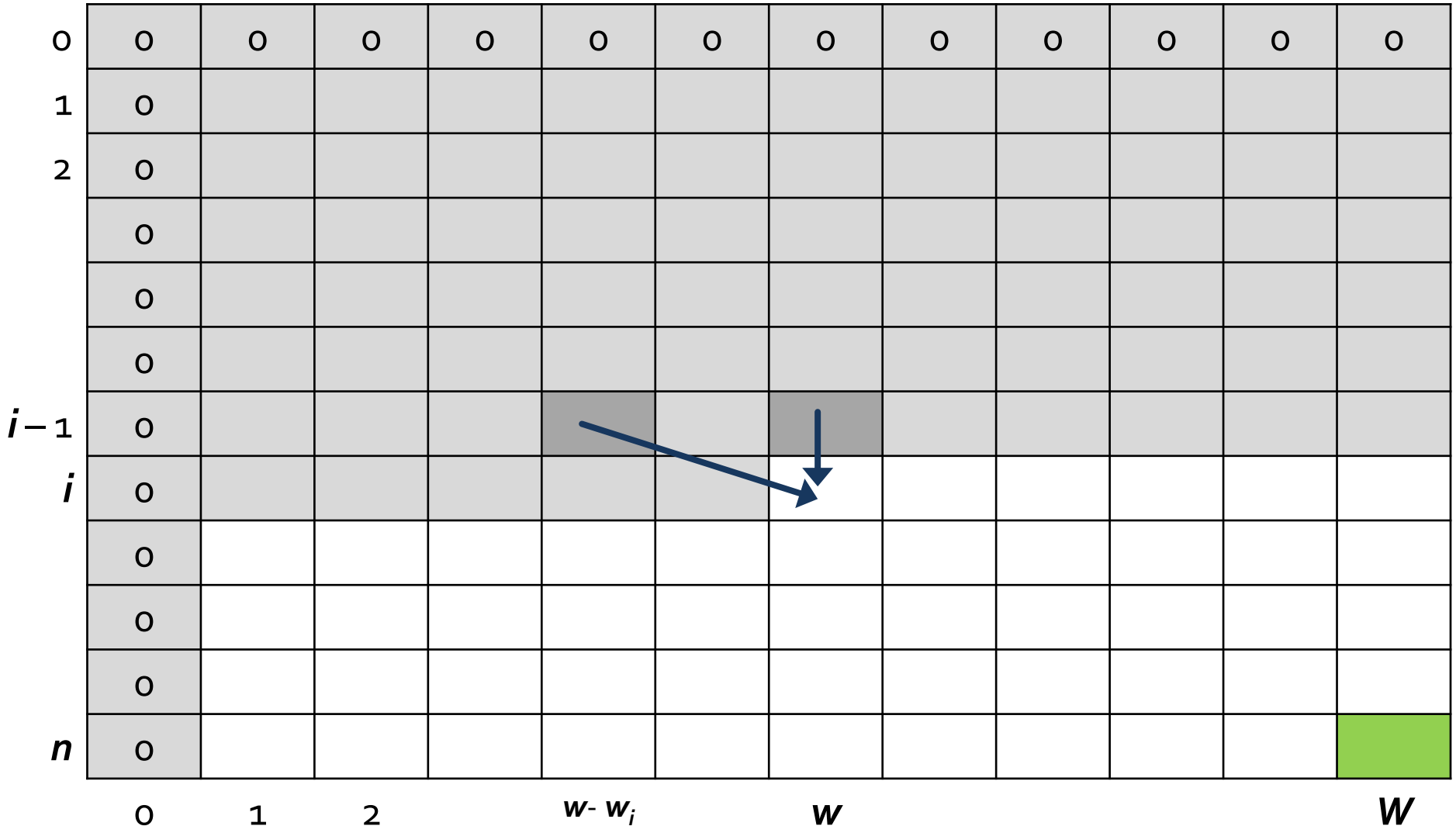
Subset-Sum(n, W)

- Create 2D array $M[0 \ldots n][0 \ldots W]$
- For w from 1 to W
  - Initialize $M[0][w] = 0$
- For i from 1 to n
  - For w from 0 to W
    - If $w < w_i$, then
      - $OPT(i, w) = OPT(i - 1, w)$
    - Else
      - $OPT(i, w) = \max(OPT(i - 1, w), w_i + OPT(i - 1, w - w_i))$
- Return $M[n][W]$

What does that look like?

- We're building a big 2D array
- Its size is ***nW***
  - ***n*** is the number of items
  - ***W*** is the maximum weight
  - Actually, it's got one more row and one more column, just to make things easier
- The book makes this array with row 0 at the bottom
- I've never seen anyone else do that
- I'm going to put row 0 at the top

Table *M* of OPT values



Running time

- The algorithm has a simple nested loop
  - The outer loop runs $n + 1$ times
  - The inner loop runs $W + 1$ times
- The total running time is $O(nW)$
- The space needed is also $O(nW)$
- Note that this time is **not** polynomial in terms of n
- It's polynomial in n and W, but W is the maximum weight
  - Which could be huge!
- We call running times like this **pseudo-polynomial**
- Things are fine if W is similar to n, but it could be huge!

Reconstructing the answer

- Like the other dynamic programming problems, the hard part is finding the actual value of the optimal solution
- We can trace back from $M[n][W]$, depending on whether the value was included or not
- Given a filled in table M, we can find an optimal set of jobs in $O(n)$ time

Subset sum example

- Weights: 2, 7, 1, 3, 8, 4
- Maximum: 19
- Create the table to find all of the optimal values that include items 1, 2,…, ***i*** for every possible weight ***w*** up to 19

Table to fill in

i	w_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	**2**	0																			
2	**7**	0																			
3	**1**	0																			
4	**3**	0																			
5	**8**	0																			
6	**4**	0																			

Knapsack

Knapsack

- The knapsack problem is a classic problem that extends subset sum a little
- As before, there is a maximum capacity W and each item has a weight w_i
- Each item also has a value v_i
- The goal is to maximize the value of objects collected without exceeding the capacity
- … like Indiana Jones trying to put the most valuable objects from a tomb into his limited-capacity knapsack

An easy extension

- The knapsack problem is really the same as subset sum, except that we are concerned with maximum **value** instead of maximum **weight**
- We need only to update the recurrence to keep the maximum value:
  - If $w < w_i$, then OPT(i, w) = OPT($i - 1$, w)
  - Otherwise, OPT(i, w) = max(OPT($i - 1$, w), v_i + OPT($i - 1$, $w - w_i$))

Knapsack example

- Items (w_i, v_i):
  - (7, 9)
  - (3, 4)
  - (2, 3)
  - (6, 2)
  - (4, 5)
  - (5, 7)
- Maximum weight: 10
- Create the table to find all of the optimal values that include items 1, 2,…, ***i*** for every possible weight ***w*** up to 10

Fill in the table

i	w_i	v_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	7	9	0										
2	3	4	0										
3	2	3	0										
4	6	2	0										
5	4	5	0										
6	5	7	0										

Upcoming

Next time…

- Sequence alignment

Reminders

- Work on Homework 5
- Read section 6.6