Objectives

- Dynamic Programming
  - Knapsack
  - Sequence Alignment

Review

- What is the new algorithm design technique we're learning?
- What is the least segmented squares problem?
- What was our solution to the problem?

Knapsack Problem

- Given *n* objects and a "knapsack"
- Item *i* weighs $w_i > 0$ kilograms and has value $v_i > 0$
  - Example: jobs require w_i time
- Knapsack has capacity of *W* kilograms
  - Example: *W* is time interval that resource is available

Goal: fill knapsack so as to maximize total value

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Towards a Recurrence…

- What do we know about the knapsack with respect to item *i*?

Towards a Recurrence…

- What do we know about the knapsack with respect to item *i*?
  - Either select item *i* or not
  - If don't select
    - Pick optimum solution of remaining items
  - Otherwise

What happens?
How does problem change?
Formulate the recurrence

Dynamic Programming: False Start

- Def. OPT(i) = max profit subset of items 1, …, i
  - Case 1: OPT does not select item i
    - OPT selects best of { 1, 2, …, i-1 }
  - Case 2: OPT selects item i
    - Accepting item *i* does not immediately imply that we will have to reject other items
      - No known conflicts
    - Without knowing what other items were selected before *i*, we don't even know if we have enough room for *i*

➡Need more sub-problems!

Dynamic Programming: Adding a New Variable

- Def. OPT(i, **w**) = max profit subset of items 1, …, i with weight limit **w**
  - Case 1: OPT does not select item *i*
    - OPT selects best of { 1, 2, …, i-1 } using weight limit ***w***
  - Case 2: OPT selects item *i*
    - new weight limit = $w - w_i$
    - OPT selects best of { 1, 2, …, i–1 } using new weight limit, $\boldsymbol{w - w_i}$

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

```
Input: N, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to N
   for w = 1 to W
      if wi > w :
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], vi + M[i-1, w-wi] }

return M[n, W]
```


Knapsack Problem: Bottom-Up

- Fill up an n-by-W array

```
Input: N, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to N     # for all items
   for w = 1 to W  # for all possible weights
      if wi > w :  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], vi + M[i-1, w-wi] }

return M[n, W]
```


Knapsack Algorithm

Represents weight in knapsack

W → # of entries: W + 1

i → # of entries: n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0											
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

Represents item id

OPT:
Solution =

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

i = 1

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

OPT:
Solution =

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

i = 2

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

OPT:
Solution =

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

i = 3

W + 1

n + 1	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0											
{1, 2, 3, 4, 5}	0											

OPT:
Solution =

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

i = 4

W + 1

n + 1	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0											

OPT:
Solution =

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

i = 5

W + 1

n + 1	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

OPT:
Solution =

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

What is the optimal solution?

Knapsack Algorithm

W + 1

n + 1	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

OPT: 40 = 22 + 18
Solution={4, 3}

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Analyzing Solution

How do we figure out the optimal *solution*?

```
Input: N, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to N      # for all items
   for w = 1 to W   # for all possible weights
      if wi > w :   # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], vi + M[i-1, w-wi] }

return M[n, W]
```

Costs?

Analyzing Solution

```
Input: N, w1,…,wN, v1,…,vN

for w = 0 to W                                  O(W)
   M[0, w] = 0

for i = 1 to N      # for all items
   for w = 1 to W   # for all possible weights    O(N W)
      if wi > w :   # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], vi + M[i-1, w-wi] }

return M[n, W]
```

Knapsack Problem: Running Time

- Running time. $\Theta(n W)$
  - **Not** polynomial in input size!
  - "Pseudo-polynomial"
    - Reasonably efficient when W is reasonably small
  - Decision version of Knapsack is NP-complete [Chapter 8]
- Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Review: Dynamic Programming

- What are the key ideas?
- What is our approach to solve a problem using dynamic programming?

Review: Dynamic Programming

- What is the key idea?
  - Memoization: remember the answer for subproblems
    - Improves running time
    - Tradeoff in space
  - Can calculate answer to problem from subproblems
- What is our approach to solve a problem using dynamic programming?
  - Figure out what we're optimizing
  - Figure out how to break the problem into subproblems
  - Figure out how to compute solution from subproblems
  - Define the recurrence relation between the problems

What was the Key to Solving each of these Problems?

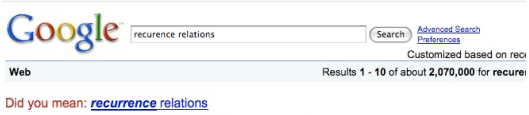
- Weighted interval scheduling
- Segmented least squares
- Knapsack

What was the Key to Solving each of these Problems?

- Weighted interval scheduling
  - Binary decision: job was in or wasn't
  - Know conflicts→ reduce problem
- Segmented least squares
  - Knew last point was definitely in one segment
    - Could reduce
  - Multiway decision→ many possibilities for segment starting point
- Knapsack
  - If select an item, reduce available size by item's size
    - Find opt solution for smaller weight, remaining items

SEQUENCE ALIGNMENT

Has This Ever Happened To You?



How does Google know what I really meant?

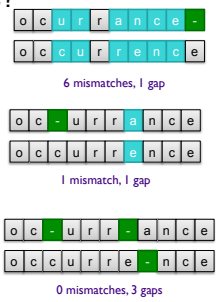
String Similarity

- How similar are two strings?
  - ocurrance
  - occurrence
- We intuitively can tell that these two are similar
  - Systematic measurement?

String Similarity

- How similar are two strings?
  - ocurrance
  - occurrence
- Measurements
  - Gap (-): add a letter
  - Mismatch

Which is the best alignment?



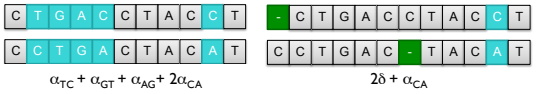
Applications of String Similarity

- Basis for Unix `diff`
  - Longest common subsequence
- Spam filters
  - Similarity to known spam message
- Computational biology
  - Ex: Figuring out how similar two genomes (sequences of A, C, G, T) are
- Alignment with **non** English/natural language strings are less obvious how to align

Edit Distance

- [Levenshtein 1966, Needleman-Wunsch 1970]
  - Gap penalty: δ
  - Mismatch penalty: α_{pq}
    - If p and q are the same, then mismatch penalty is 0
  - **Cost** = sum of gap and mismatch penalties

Parameters allow us to tweak cost



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$ $\quad$ $2\delta + \alpha_{CA}$

Oracle of Bacon

The Oracle cannot find "*Dan Akroid*." You have probably misspelled your entry, but it's also possible that the actor or actress you seek is not in our database. Below are 12 close matches. If you don't find what you're looking for in that list, try searching at www.imdb.com.

- Dan Aykroyd
- Dan Akre
- Dan Alford
- Dan Arkin
- Dan Broad
- Dan Krohn
- Dan Kroon
- Dan Marois
- Dan Sakraida
- Dan Stroud
- Dana Roi
- Daniel Krogh

Looking Ahead

- Exam2 due today at 5 p.m.
- Wiki due Tuesday
  - Chap 6: 6.1-6.4
- PS8 due Friday