Objectives

- Dynamic Programming
  - Knapsacks
  - RNA Substructure

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?

Dynamic Programming: False Start

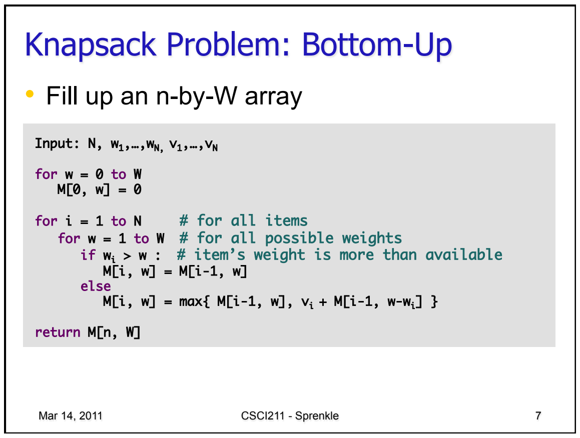
- Def. OPT(i) = max profit subset of items 1, …, i
  - Case 1: OPT does not select item i
    - OPT selects best of { 1, 2, …, i-1 }
  - Case 2: OPT selects item i
    - Accepting item *i* does not immediately imply that we will have to reject other items
      - No known conflicts
    - Without knowing 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

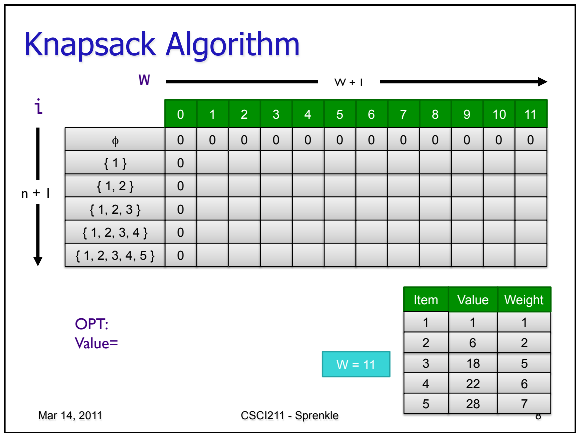
- Fill up an n-by-W array

```
Input: N, w_1,…,w_N, v_1,…,v_N

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 w_i > w :  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], v_i + M[i-1, w-w_i] }

return M[n, W]
```



Knapsack Algorithm

w

w + 1

i

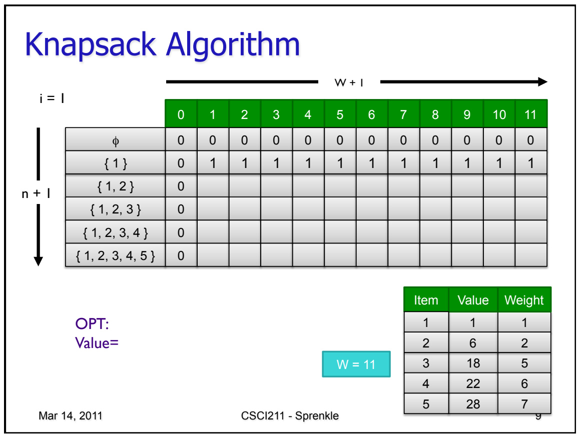
n + 1

	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											

OPT:
Value=

W = 11

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



Knapsack Algorithm

w + 1

i = 1

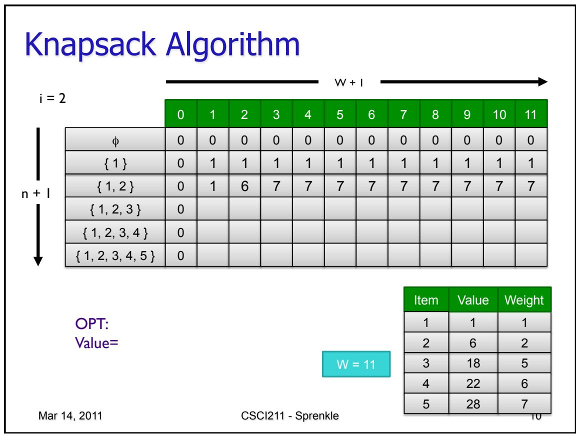
n + 1

	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:
Value=

W = 11

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



Knapsack Algorithm

w + 1

i = 2

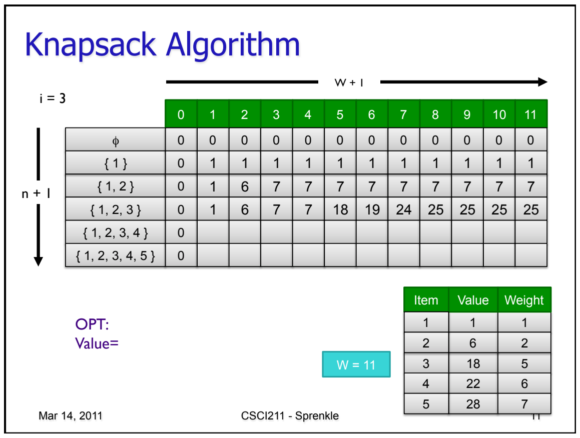
n + 1

	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:
Value=

W = 11

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



Knapsack Algorithm

w + 1

i = 3

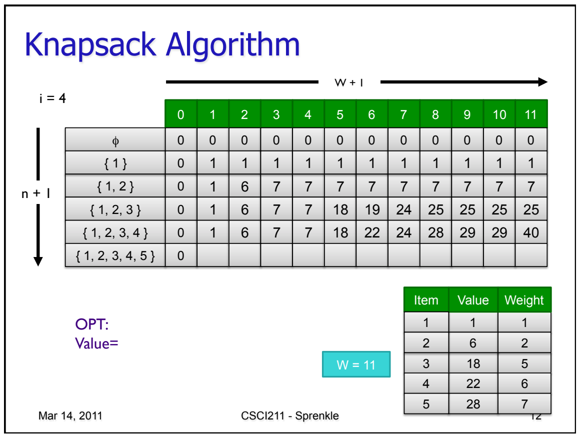
n + 1

	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:
Value=

W = 11

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



Knapsack Algorithm

w + 1

i = 4

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:
Value=

W = 11

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

Knapsack Algorithm

W + 1

i = 5

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:
Value=

W = 11

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

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
Value={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, $w_1,…,w_N$, $v_1,…,v_N$

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 $w_i$ > w :  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], $v_i$ + M[i-1, w-$w_i$] }

return M[n, W]
```

Costs?

Analyzing Solution

- Costs?

```
Input: N, $w_1,…,w_N$, $v_1,…,v_N$

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 $w_i$ > w :  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], $v_i$ + M[i-1, w-$w_i$] }

return M[n, W]
```

Knapsack Problem: Running Time

- Running time. Θ(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 is the key idea?
- 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
- 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

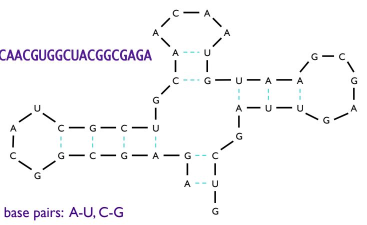
Applications of Dynamic Programming to Computational Biology

RNA SECONDARY STRUCTURE

RNA Secondary Structure

- RNA. String B = $b_1 b_2 \ldots b_n$ over alphabet { A, C, G, U }
- Secondary structure. RNA is *single-stranded* so it tends to loop back and form base pairs with itself
  - This structure is essential for understanding behavior of a molecule.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

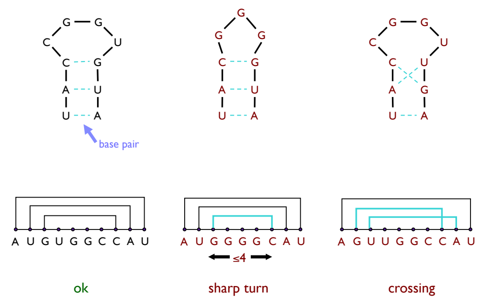


complementary base pairs: A-U, C-G

RNA Secondary Structure: Which Pairs Can We Combine?

- A set of pairs S = { (b_i, b_j) } that satisfy:
  - [Watson-Crick] S is a *matching* and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C
  - [No sharp turns] The ends of each pair are separated by *at least 4* intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$
  - [Non-crossing] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$

Examples of RNA Secondary Structure

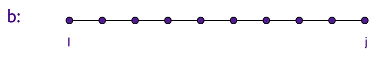


RNA Secondary Structure

- A set of pairs S = { (b_i, b_j) } that satisfy:
  - [Watson-Crick] S is a *matching* and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C
  - [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If (b_i, b_j) $\in$ S, then $i < j - 4$
  - [Non-crossing] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$
- Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the *optimum total free energy*. ← *approximate by number of base pairs*
- Goal. Given an RNA molecule B = $b_1b_2…b_n$, find a secondary structure S that *maximizes the number of base pairs*

Toward a Solution: First Attempt

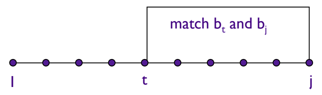
- OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1b_2…b_j$



- Towards a recurrence relation…
  - What are the possibilities?
    - What does b_j match with?
  - What are the subproblems?

Toward a Solution: First Attempt

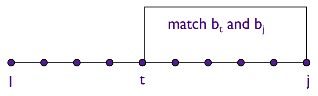
- OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1b_2…b_j$



- Relation:
  - If j isn't involved in a pair
  - If j is involved, results in two sub-problems

Toward a Solution: First Attempt

- OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1b_2…b_j$



- Relation:
  - If j isn't involved in a pair: Opt(j-1)
  - If j is involved, results in two sub-problems
    - Finding secondary structure in: $b_1b_2…b_{t-1}$ ← OPT(t-1)
    - Finding secondary structure in: $b_{t+1}b_{t+2}…b_{j-1}$ ← need more subproblems

Doesn't match our subproblem (doesn't start at 1)
Need to start *anywhere*

Dynamic Programming Over Intervals

- OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_ib_{i+1}…b_j$
  - What are the different cases?
  - How does it affect the recurrence relation?
    - For example, when will we know that there isn't a pair?

Dynamic Programming Over Intervals

- OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$
  - Case 1. If $i \geq j - 4$
    - OPT(i, j) = 0 by *no-sharp turns* condition
  - Case 2. Base b_j is not involved in a pair
    - OPT(i, j) = OPT(i, j-1)
  - Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$
    - *non-crossing* constraint decouples resulting sub-problems
    - OPT(i, j) = 1 + $\max_t$ { OPT(i, t-1) + OPT(t+1, j-1) }

pairing

take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements

Recurrence Relation

- Putting it all together…

j not in a base pair in optimal solution

```
➢ Opt(i,j) = max( Opt(i,j-1),
  max_t( 1+Opt(i,t-1)+Opt(t+1,j-1) ) )
```

j in a base pair in optimal solution
Adds 1 pair
Look at remaining letters

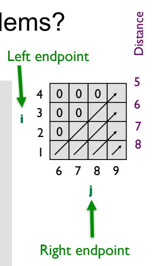
RNA Algorithm

- What order to solve the sub-problems?
  - Do shortest intervals first

```
Initialize M[i,j] = 0 for i >= j-4

RNA(b_1,…,b_n):
   for k = 5, 6, …, n-1          (distances)
      for i = 1, 2, …, n-k        (start)
         j = i + k        (end)
         M[i, j] = max(M[i,j-1],
                max_t(1+M[i,t-1]+M[t+1,j-1]) )

   return M[1, n]
```



Costs?

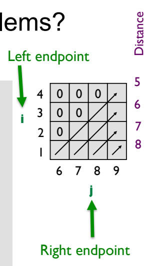
RNA Algorithm

- What order to solve the sub-problems?
  - Do shortest intervals first

```
Initialize M[i,j] = 0 for i >= j-4

RNA(b_1,…,b_n):
   for k = 5, 6, …, n-1          (distances)
      for i = 1, 2, …, n-k        (start)
         j = i + k        (end)
         M[i, j] = max(M[i,j-1],
                max_t(1+M[i,t-1]+M[t+1,j-1]) )

   return M[1, n]
```



- Running time: $O(n^3)$

Dynamic Programming Summary

- Recipe
  - Characterize structure of problem
  - Recursively define *value* of optimal solution
  - Compute value of optimal solution
  - Construct *optimal solution* from computed information
- Dynamic programming techniques
  - Binary choice: weighted interval scheduling
  - Multi-way choice: segmented least squares
  - Adding a new variable: knapsack
  - Dynamic programming over intervals: RNA secondary structure
- Top-down vs. bottom-up: different people have different intuitions

This Week

- Wed: Wiki
  - Chapter 5.5; 6, up to and including 6.4
  - Jan Cuny's visit
    - 3 p.m. – reception to meet Jan
    - 4 p.m. – Broadening Participation in Computing
- Friday: Problem Set 7 due
  - Looks short but lots of parts
  - Exam 2 will be handed out