6. Dynamic Programming

Those who cannot remember the
past are condemned to repeat it

-Santayana

Algorithmic Paradigms

Greed. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

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

Greedy: repeatedly add item with maximum ratio v_i / w_i.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

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
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. 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 this new weight limit

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

Knapsack Problem: Bottom-Up

Knapsack. 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 w = 1 to W
      if (w_i > w)
         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 + 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	34	40

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

W = 11

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

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- 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]

Line Breaking

Breaking a Paragraph into Lines

Given a sequence of words and a line length, distribute the words accross multiple lines, minimizing the sum of the costs of the extra spaces (except for the last line)

Example

- Cost =(# spaces at the end of the line)^3

```
Those who cannot
remember the        ← 4^3 +
past are            ← 8^3 +
condemned to        ← 4^3
repeat it.          = 640
```

```
Those who           ← 7^2 +
cannot remember     ← 1^2 +
the past are        ← 4^2 +
condemned to        ← 4^2
repeat it.          = 472
```

Greedy
(Microsoft Word, for example)

Non-greedy
(TEX, for example)

Linebreaking

Input:

- Sequence of word lengths $w_1,..,w_n$ and line width W where W and each w_i have an implied space at the end

Output:

- breakpoints $b_1, ..., b_i,..b_m$, specifying last word to be put on the ith line where:
  - Words on each line i contain words $(b_{i-1}, b_i]$
  - Penalty for words $w_j..w_k$ on a line = $(W - (w_j + ... + w_k))^3$

Example
W = 17

Words	w_i
Those	6
who	4
cannot	7
remember	9
the	4
past	5
are	4
condemned	10
to	3
repeat	7
it.	4

Linebreaking

First thought

- Decompose into subproblems
  - Break words 1..k into lines
  - Break words k+1..n into lines
  - Iterate over all choices of k
- Our subproblem would be finding
  - OPT(i, j) (min. penalty linebreaks for words i through j inclusive)

Problem

- # of sub-problems: about n^2 (size of array)
- Will take O(n) to compute each array entry
- Total will be $O(n^3)$. Yikes!

Linebreaking

Second thought

- Instead of trying to solve subproblems for general (i, j), solve only for (i, n)
  - OPT(i, n) = min penalty linebreaks for words i through n
- When called to linebreak, try all possibilities of breaking this line calling recursively to place the rest
- Only need to try w/2 possibilities for the linebreak

Code (Assumes memoization is happening automatically)

- OPT(i, n) // puts words into line L,.... returns
  // penalty for words from i..n

  if (w_i + ... + $w_n \leq W$)

  put all words in line L and return 0

  for all k ≥i where w_i+...+$w_k \leq W$

  $penalty_k = (W - w_i + \ldots + w_k)^3 + OPT(k+1, n)$

  let k_{min} = k that produces minimum $penalty_k$

  Put words i..k_{min} in line L

  Return minimum penalty

Linebreaking

Total running time: $\Theta(Wn)$. If W is considered a constant, running time is $\Theta(n)$

Total space: $\Theta(n)$ (for the memoization dictionary)

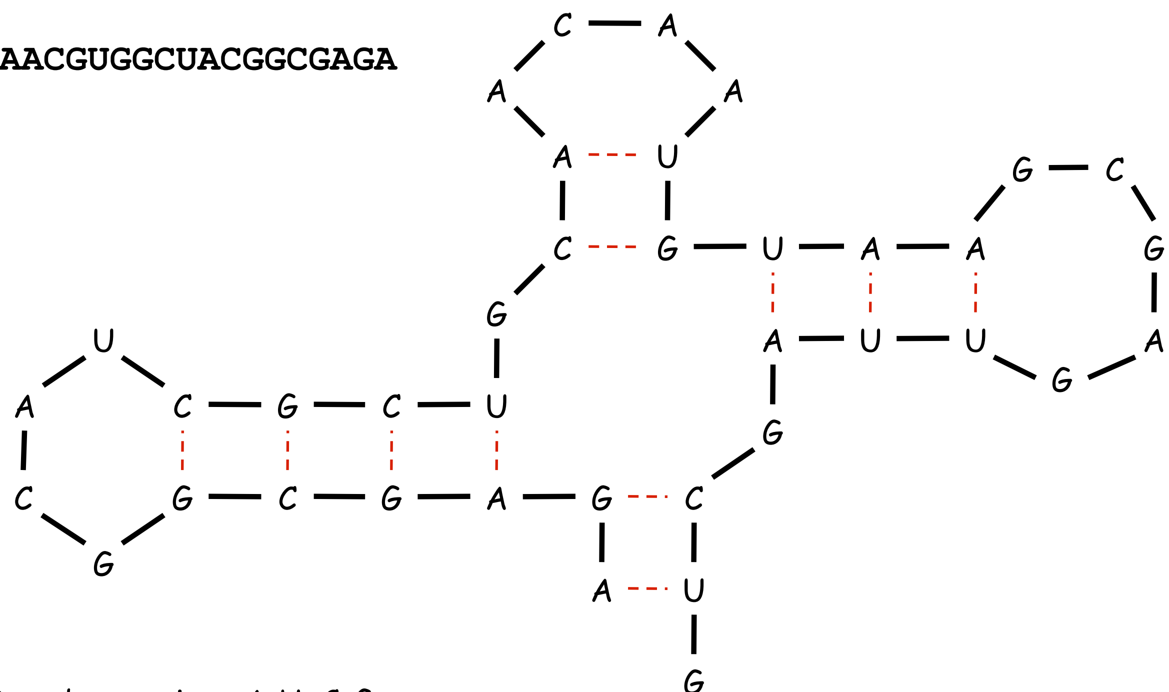
6.5 RNA Secondary Structure

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 molecule.

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



complementary base pairs: A-U, C-G

RNA Secondary Structure

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 $i < k < j < l$.

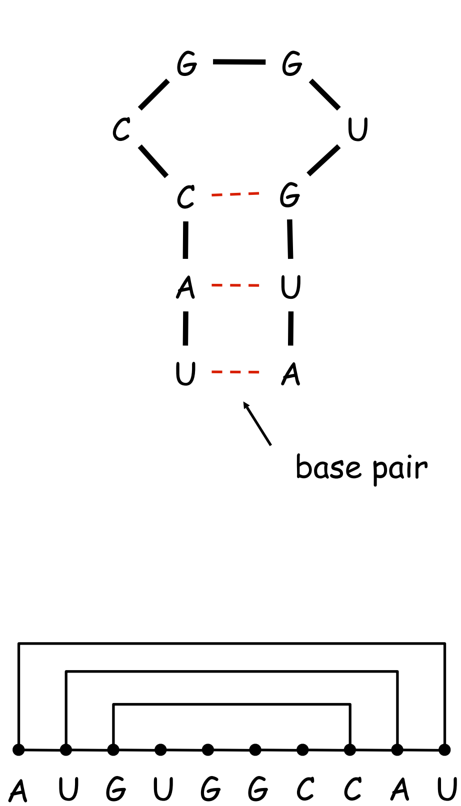
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

approximate by number of base pairs

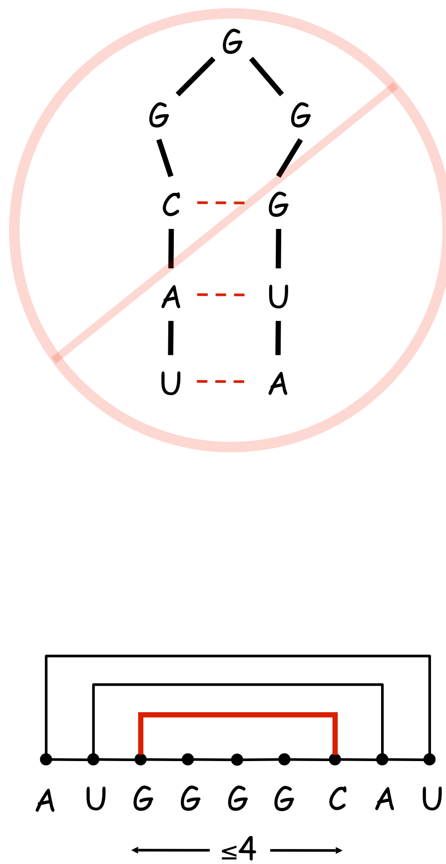
Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

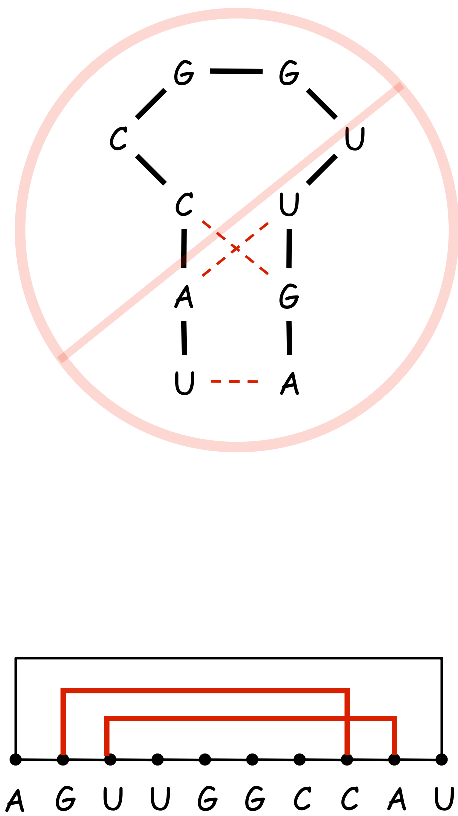
Examples.



ok



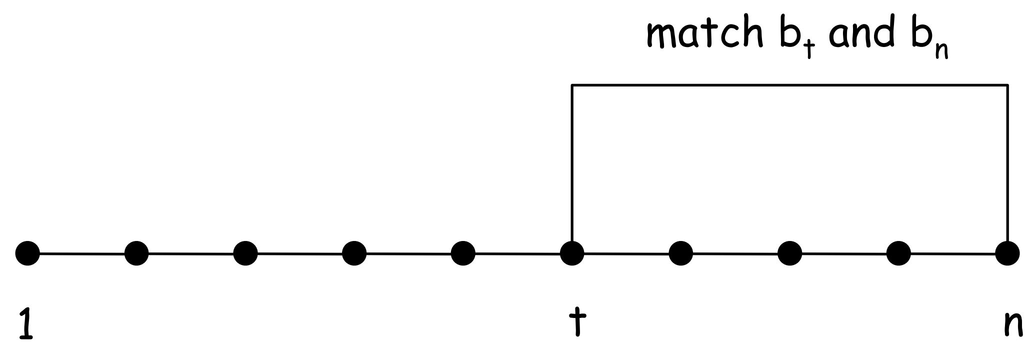
sharp turn



crossing

RNA Secondary Structure: Subproblems

First attempt. OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1b_2...b_j$.



Difficulty. Results in two sub-problems.

- Finding secondary structure in: $b_1b_2...b_{t-1}$. ← OPT(t-1)
- Finding secondary structure in: $b_{t+1}b_{t+2}...b_{n-1}$. ← need more sub-problems

Dynamic Programming Over Intervals

Notation. $OPT(i, j)$ = 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
  - $OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

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

Remark. Same core idea in CKY algorithm to parse context-free grammars.

Bottom Up Dynamic Programming Over Intervals

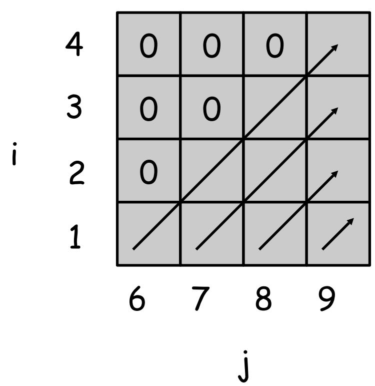
Q. What order to solve the sub-problems?

A. Do shortest intervals first.

```
RNA(b1,…,bn) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute M[i, j]

   return M[1, n]
}
```

using recurrence



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.