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.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

\( W = 11 \)

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.** $OPT(i) = \text{max profit subset of items 1, \ldots, i}.$

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$

- **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. \( \text{OPT}(i, w) = \text{max profit subset of items 1, ..., i with weight limit } w \).

- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{1, 2, ..., i-1\} \) using weight limit \( w \)

- Case 2: \( \text{OPT} \) selects item \( i \).
  - new weight limit = \( w - w_i \)
  - \( \text{OPT} \) selects best of \( \{1, 2, ..., i-1\} \) using this new weight limit

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

OPT: \{ 4, 3 \}

value = 22 + 18 = 40

W = 11
Knapsack Problem: Running Time

Running time. $\Theta(nW)$.
- 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 across multiple lines, minimizing the sum of the costs of the extra spaces (except for the last line)

Example

$\text{Cost } = (\text{# spaces at the end of the line})^3$

<table>
<thead>
<tr>
<th>Those who cannot remember the past are condemned to repeat it.</th>
<th>4$^3$ + 8$^3$ + 4$^3$ = 640</th>
</tr>
</thead>
<tbody>
<tr>
<td>Those who cannot remember the past are condemned to repeat it.</td>
<td>7$^2$ + 1$^2$ + 4$^2$ + 4$^2$ = 472</td>
</tr>
</tbody>
</table>

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 $i$th 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
$W = 17$

<table>
<thead>
<tr>
<th>Words</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Those</td>
<td>6</td>
</tr>
<tr>
<td>who</td>
<td>4</td>
</tr>
<tr>
<td>cannot</td>
<td>7</td>
</tr>
<tr>
<td>remember</td>
<td>9</td>
</tr>
<tr>
<td>the</td>
<td>4</td>
</tr>
<tr>
<td>past</td>
<td>5</td>
</tr>
<tr>
<td>are</td>
<td>4</td>
</tr>
<tr>
<td>condemned</td>
<td>10</td>
</tr>
<tr>
<td>to</td>
<td>3</td>
</tr>
<tr>
<td>repeat</td>
<td>7</td>
</tr>
<tr>
<td>it</td>
<td>4</td>
</tr>
</tbody>
</table>
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 \((i, n)\)
  - \(\text{OPT}(i, n) = \text{min penalty linebreaks for words } i \text{ through } n\)
- When called to linebreak, try all possibilities of breaking \textit{this line}
calling recursively to place the rest
- Only need to try \(w/2\) possibilities for the linebreak

Code (Assumes memoization is happening automatically)

\[
\text{OPT}(i, n) \quad // \text{puts words into line } L, \ldots \text{ returns } \\
\quad // \text{penalty for words from } i..n
\]

if \((w_i + \ldots + w_n \leq W)\)
  put all words in line \(L\) and return 0
for all \(k \geq i\) where \(w_i + \ldots + w_k \leq W\)
  penalty\(_k\) = \((W - w_i + \ldots + w_k)^3 + \text{OPT}(k+1, n)\)
let \(k_{\text{min}} = k\) that produces minimum penalty\(_k\)
Put words \(i..k_{\text{min}}\) 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 $B = b_1b_2...b_n$ 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: GUCGAUUGAGCGAAUGUAACACGUGGCUACGGCGAGA

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 $B = b_1b_2\ldots b_n$, find a secondary structure $S$ that maximizes the number of base pairs.
Examples.

RNA Secondary Structure: Examples

Examples.

- A G U G G C C A U
  - base pair
  - 
  - ok

- A U G U G G C C A U
  - sharp turn
  - ≤4

- A G U G G G C C A U
  - crossing
RNA Secondary Structure: Subproblems

First attempt. \( \text{OPT}(j) = \) 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: \( b_1b_2...b_{t-1} \).  \( \text{OPT}(t-1) \)
- Finding secondary structure in: \( b_{t+1}b_{t+2}...b_{n-1} \).  \( \) need more sub-problems
Dynamic Programming Over Intervals

Notation. \( \text{OPT}(i, j) = \) maximum number of base pairs in a secondary structure of the substring \( b_i b_{i+1} \ldots b_j \).

- Case 1. If \( i \geq j - 4 \).
  - \( \text{OPT}(i, j) = 0 \) by no-sharp turns condition.

- Case 2. Base \( b_j \) is not involved in a pair.
  - \( \text{OPT}(i, j) = \text{OPT}(i, j-1) \)

- Case 3. Base \( b_j \) pairs with \( b_t \) for some \( i \leq t < j - 4 \).
  - non-crossing constraint decouples resulting sub-problems
  - \( \text{OPT}(i, j) = 1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \} \)
    
    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.

Running time. $O(n^3)$. 

\[
\begin{align*}
\text{RNA}(b_1, \ldots, b_n) \{ \\
& \text{for } k = 5, 6, \ldots, n-1 \\
& \quad \text{for } i = 1, 2, \ldots, n-k \\
& \quad \quad j = i + k \\
& \quad \quad \text{Compute } M[i, j] \\
& \quad \text{return } M[1, n] \\
\}
\]
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.