Chapter 6
Dynamic Programming

Algorithmic Paradigms

- **Greed.** Build up a solution incrementally, only 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.

Dynamic Programming History

- Richard Bellman. Pioneered the systematic study of dynamic programming in the 1950s.
- **CHOICE OF THE NAME DYNAMIC PROGRAMMING.**
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
  - "It's impossible to use dynamic in a pejorative sense."
  - "Something not even a Congressman could object to."


Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ....
- Some famous dynamic programming algorithms.
  - Bellman-Ford for shortest path routing in networks.
  - Smith-Waterman for sequence alignment.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems.
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem.
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

Steps in Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.
Longest Common Subsequence

- **Problem:** Given 2 sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, find a common subsequence whose length is maximum.

Naïve Algorithm
- For every subsequence of $X$, check whether it’s a subsequence of $Y$.
- **Time:** $\Theta(n2^m)$.
- $2^m$ subsequences of $X$ to check.
- Each subsequence takes $\Theta(n)$ time to check: scan $Y$ for first letter, for second, and so on.

Longest Common Subsequence

- **Problem:** Given 2 sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, find a common subsequence whose length is maximum.

Optimal Substructure
- **Theorem:**
  Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of $X$ and $Y$.
  1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
  2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and $Z$ is an LCS of $X_m$ and $Y$.
  3. or $z_k \neq y_n$ and $Z$ is an LCS of $X$ and $Y_{n-1}$.

Optimal Substructure
- **Proof:** (case 1: $x_m = y_n$)
  Any sequence $Z'$ that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,
  1. longest common subsequence (LCS) $Z$ must end in $x_m = y_n$.
  2. $Z_{k-1}$ is a common subsequence of $X_{m-1}$ and $Y_{n-1}$, and
  3. there is no longer CS of $X_{m-1}$ and $Y_{n-1}$, or $Z$ would not be an LCS.
Recursive Solution

- Define $c[i, j] =$ length of LCS of $X_i$ and $Y_j$, where $X_i = (x_{i-1}, x_i)$ is the first $i$ letters of $X$.
- We want $c[m, n]$.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem. But does it solve it well?

Cost: $O(?)$

Steps in Dynamic Programming

2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

Computing the length of an LCS

```
LCS-LENGTH (X, Y)
1: m ← length[X]
2: n ← length[Y]
3: for i ← 1 to m do
4:   do b[i, 0] ← 0
5: for j ← 0 to n do
6: for i ← 1 to m do
7:   do for j ← 1 to n do
8:     do if x[i] = y[j] then
9:         do b[i, j] ← b[i-1, j-1] + 1
10:        else if b[i, j-1] ≥ b[i-1, j] then
11:         do b[i, j] ← b[i, j-1]
12:        else b[i, j] ← b[i-1, j]
13:    end if
14: end if
15: end for
16: end for
17: c[m, n] contains the length of an LCS of X and Y.
18: Time: $O(mn)$
```

Constructing an LCS

```
PRINT-LCS (b, X, i, j)
1: if i = 0 or j = 0 then return
2: if b[i, j] = "\" then
3:   print x[i]
4: elseif b[i, j] = "↑" then
5:   PRINT-LCS(b, X, i-1, j)
6: else
7:   PRINT-LCS(b, X, i, j-1)
8: end if
9: end if
```

- Initial call is PRINT-LCS (b, X, m, n).
- When $b[i, j] = \\$, we have extended LCS by one character. So LCS = entries with \$ in them.
- Time: $O(m+n)$
Seam carving

Energy function

$$e_1(I) = |\frac{\partial}{\partial x} I| + |\frac{\partial}{\partial y} I|$$

The Operator

- Given an energy function $e$, we can define the cost of a seam as
- look for the optimal seam $s^*$ that minimizes this seam cost:
  $$E(s) = E(I_s) = \sum_{i=1}^{n} e(I(s_i))$$
- The optimal seam can be found using dynamic programming,
  $$s^* = \min_{s} E(s) = \min_{s} \sum_{i=1}^{n} e(I(s_i))$$

Bonus

Figure 5: Comparing aspect ratio change. From left to right in the bottom: the image resized using seam removals, scaling and cropping.

Figure 11: Simple object removal: the user marks a region for removal (green), and possibly a region to protect (red), on the original image (see inset in left image). On the right image, consecutive vertical seam were removed until no ‘green’ pixels were left.
Weighted Interval Scheduling

- Weighted interval scheduling problem.
- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

Unweighted Interval Scheduling

- Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

- Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Weighted Interval Scheduling

- Notation. Label jobs by finishing time: $f_1 \leq f_2 \ldots \leq f_n$.
- Let $p(j) =$ largest index $i (<j)$ s.t. job $i$ is compatible with $j$.
- Ex: $p(8) = 5$, $p(7) = 6$, $p(2) = 0$.
Dynamic Programming: Binary Choice

- Let \( \text{OPT}(j) \) = value of optimal solution to the problem consisting of job requests \( 1, 2, ..., j \).
  - Case 1: \( \text{OPT} \) selects job \( j \).
    - Can't use incompatible jobs \( \{ p(j) + 1, p(j) + 2, ... \} \).
    - Must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, ..., p(j) \).
  - Case 2: \( \text{OPT} \) does not select job \( j \).
    - Must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, ..., j-1 \).

\[ \text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + \text{OPT}(p(j)), \text{OPT}(j-1) \} & \text{otherwise} \end{cases} \]

Weighted Interval Scheduling: Brute Force

- Brute force algorithm.

\[
\text{Cost} = \frac{1 + \sqrt{5}}{2}
\]

- Example. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.
- Observation. Recursive algorithm fails spectacularly because of redundant sub-problems \( \Rightarrow \) exponential algorithms.

Weighted Interval Scheduling: Memoization

- Memoization. Store results of each sub-problem in a cache; lookup as needed.

Weighted Interval Scheduling: Running Time

- Claim. Memorized version of alg takes \( O(n \log n) \) time.
  - Sort by finish time: \( O(n \log n) \).
  - Computing \( p(\cdot) \): \( O(n) \) after sorting by start time.
  - \( \text{M-Compute-Opt} \): each invocation takes \( O(1) \) time & either
    - (i) returns an existing value \( M[j] \)
    - (ii) fills in one new entry \( M[j] \) and makes two recursive calls
  - Progress measure \( \Phi = \# \) nonempty entries of \( M[\cdot] \).
    - Initially \( \Phi = 0 \), throughout \( \Phi \leq n \).
    - (iii) increases \( \Phi \) by 1 \( \Rightarrow \) at most 2\( n \) recursive calls.
  - Overall running time of \( \text{M-Compute-Opt}(\cdot) \) is \( O(n) \).
- Note: \( O(n) \) if jobs are presorted by start & finish times.
Problem: Find mutually compatible jobs that maximize total value.

Weighted Interval Scheduling: Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.

Input: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$

Iterative-Compute-Opt {
  $M[0] = 0$
  for $j = 1$ to $n$
  $M[j] = \max(v_j + M[p(j)], M[j-1])$
}

Cost of calculating $p(i)$

<table>
<thead>
<tr>
<th>$n = 1$</th>
<th>$n = 4$</th>
<th>$n = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 2$</td>
<td>$n = 3$</td>
<td>$n = 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start (sorted)</th>
<th>Finish (sorted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

$O(n)$

Problem: Find mutually compatible jobs that maximize total value.

Exercise

<table>
<thead>
<tr>
<th>$v_1 = 2$</th>
<th>$v_2 = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_3 = 4$</td>
<td>$v_4 = 7$</td>
</tr>
<tr>
<td>$v_5 = 2$</td>
<td>$v_6 = 1$</td>
</tr>
</tbody>
</table>

Problem: Find mutually compatible jobs that maximize total value.

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.

<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>

Let $W = 11$

- Greedy method: repeatedly add item with max ratio $v_i/w_i$.
- Ex: $\{7, 2, 1\}$ achieves only value $= 35 \Rightarrow$ optimal?
- Ex: $\{3, 4\}$ has value 40.
Steps in Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

Dynamic Programming: False Start

- Let \( \text{OPT}(i) = \text{max profit for subset of items 1, \ldots, i} \).
  - Case 1: \( \text{OPT} \) does not select item \( i \).
    - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using weight limit \( w \).
  - Case 2: \( \text{OPT} \) 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 info about sub-problems!

Dynamic Programming

- Let \( \text{OPT}(i, w) = \text{max profit for subset of items 1, \ldots, i with weight limit } w \).
  - Case 1: \( \text{OPT} \) does not select item \( i \).
    - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using weight limit \( w \).
  - Case 2: \( \text{OPT} \) selects item \( i \).
    - new weight limit = \( w - w_i \)
    - \( \text{OPT} \) selects best of \( \{ 1, \ldots, i-1 \} \) using the 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: \( n, w_1, \ldots, w_N, v_1, \ldots, v_N \)
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 Problem: Running Time

- Running time: \( \Theta(n W) \).
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.
- Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.
Coin-Changing: Dynamic programming

- $140 postage
- Greedy: 100, 37, 1, 1, 1.
- Optimal: 70, 70.
- Greedy algorithm failed.
- ?