Chapter 6
Dynamic Programming

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.

Dynamic Programming History

**Bellman.** Pioneered the systematic study of dynamic programming in the 1950s.

**Etymology.**
- 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.**
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.
### 6.1 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 Review**

**Recall.** 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: $f_1 \leq f_2 \leq \ldots \leq f_n$.

**Def.** $p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.

**Ex:** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$. 

---

**Diagram:**
- Time line with intervals representing jobs.
- Different weights indicated by different colors.
- Example of how intervals are chosen based on compatibility and weight.
Dynamic Programming: Binary Choice

Notation. \( OPT(j) \) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - can’t use incompatible jobs \{ p(j) + 1, p(j) + 2, ..., j - 1 \}
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

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

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

Input: \( n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n, v_1, v_2, ..., v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq ... \leq f_n \).

Compute \( p(1), p(2), ..., p(n) \)

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

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

Input: \( n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n, v_1, v_2, ..., v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq ... \leq f_n \).

Compute \( p(1), p(2), ..., p(n) \)

\[
\text{M-Compute-Opt}(j) = \begin{cases} 
\text{max} & \text{if } (M[j] \text{ is empty}) \\
M[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1)) & \text{otherwise}
\end{cases}
\]

return \( M[j] \)
Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes \(O(n \log n)\) time.
- Sort by finish time: \(O(n \log n)\).
- Computing \(p(\cdot)\): \(O(n)\) after sorting by start time.

- \(M\)-Compute-Opt\((j)\): each invocation takes \(O(1)\) time and 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 \(\Phi \leq n\).
  - (ii) increases \(\Phi\) by 1 \(\Rightarrow\) at most \(2n\) recursive calls.

- Overall running time of \(M\)-Compute-Opt\((n)\) is \(O(n)\). 

Remark. \(O(n)\) if jobs are pre-sorted by start and finish times.

Automated Memoization

Automated memoization. Many functional programming languages
(e.g., Lisp) have built-in support for memoization.

Q. Why not in imperative languages (e.g., Java)?

\[
\begin{align*}
\text{(defun } F (n) \\
& \quad \text{ (if } \quad) \\
& \quad \quad \quad \text{ (n } \leq 1) \\
& \quad \quad \quad \quad \text{ n} \\
& \quad \quad \quad \quad \quad \text{ (+ (F (n - 1))) (F (- n 2)))))
\end{align*}
\]

Java (exponential)

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

Sort jobs by finish time so that \(f_1 \leq f_2 \leq \ldots \leq f_n\).

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

Iterative-Compute-Opt \(\{\)
\[\begin{align*}
M[0] &= 0 \\
\text{ for } j = 1 \text{ to } n \\
M[j] &= \text{max}(v_j + M[p(j)], M[j-1])
\end{align*}\]

Bottom-up dynamic programming. Unwind recursion.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if
we want the solution itself?
A. Do some post-processing.

Run \(M\)-Compute-Opt\((n)\)
Run \(\text{Find-Solution}(n)\)

\[
\text{Find-Solution}(j) \{ \\
\quad \text{ if } (j = 0) \\
\quad \quad \text{ output nothing} \\
\quad \text{ else if } (v_j + M[p(j)] > M[j-1]) \\
\quad \quad \quad \text{ print } j \\
\quad \quad \quad \text{ Find-Solution}(p(j)) \\
\quad \text{ else } \\
\quad \quad \text{ Find-Solution}(j-1) \\
\}
\]

- # of recursive calls \(\leq n \Rightarrow O(n)\).
6.3 Segmented Least Squares

Segmented least squares.
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) with \(x_1 < x_2 < \ldots < x_n\), find a sequence of lines that minimizes \(f(x)\).

Q. What’s a reasonable choice for \(f(x)\) to balance accuracy and parsimony?

\[
\begin{align*}
\text{Segmented Least Squares} \\
\text{Least squares.} \\
\text{Foundational problem in statistic and numerical analysis.} \\
\text{Given n points in the plane: } (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n). \\
\text{Find a line } y = ax + b \text{ that minimizes the sum of the squared error:}
\end{align*}
\]

\[
\text{SSE} = \sum_{i=1}^{n} (y_i - ax_i - b)^2
\]

Solution. Calculus \(\Rightarrow\) min error is achieved when

\[
\begin{align*}
a &= \frac{n \sum x_i y_i - (\sum x_i)(\sum y_i)}{n \sum x_i^2 - (\sum x_i)^2}, \\
b &= \frac{\sum y_i - a \sum x_i}{n}
\end{align*}
\]

Q. What’s a reasonable choice for \(f(x)\) to balance accuracy and parsimony?

\[
\begin{align*}
\text{Tradeoff function: } E + c L, \text{ for some constant } c > 0.
\end{align*}
\]
6.4 Knapsack Problem

Dynamic Programming: Multiway Choice

Notation.
- \( OPT(j) \) = minimum cost for points \( p_1, p_{i+1}, \ldots, p_j \).
- \( e(i, j) \) = minimum sum of squares for points \( p_i, p_{i+1}, \ldots, p_j \).

To compute \( OPT(j) \):
- Last segment uses points \( p_i, p_{i+1}, \ldots, p_j \) for some \( i \).
- Cost = \( e(i, j) + c + OPT(i-1) \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\min_{i \in S} \{ e(i, j) + c + OPT(i-1) \} & \text{otherwise}
\end{cases}
\]

Segmented Least Squares: Algorithm

**INPUT:** \( n, p_1, \ldots, p_n, c \)

**Segmented-Least-Squares()**
1. \( M[0] = 0 \)
2. for \( j = 1 \) to \( n \)
   1. for \( i = 1 \) to \( j \)
      1. compute the least square error \( e_{ij} \) for the segment \( p_i, \ldots, p_j \)
   2. \( M[j] = \min_{1 \leq i \leq j} (e_{ij} + c + M[i-1]) \)
3. return \( M[n] \)

Running time. \( O(n^2) \). can be improved to \( O(n^3) \) by pre-computing various statistics.

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.** $OPT(i) =$ 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, 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 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]

## 6.5 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.

**Goal.** Given an RNA molecule \( B = b_1b_2\ldots b_n \), find a secondary structure \( S \) that maximizes the number of base pairs.
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 \( i = j - 4 \),
  - \( \text{OPT}(i, j) = 0 \) by no-sharp turns condition.
- **Case 2.** Base \( b_i \) 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 \( i + 1 \leq t < j - 4 \).
  - 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 \( i + 1 \leq t < j - 4 \)
    
    \( b_t \) and \( b_j \) are Watson-Crick complements

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

RNA Secondary Structure: Examples

**Examples.**

RNA Secondary Structure: Subproblems

**First attempt.** \( \text{OPT}(j) = \) maximum number of base pairs in a secondary structure of the substring \( b_1 b_2 \ldots b_j \).

- match \( b_t \) and \( b_n \)
- \( 1 \leq t < n \)

**Difficulty.** Results in two sub-problems.
- Finding secondary structure in: \( b_1 b_2 \ldots b_{t-1} \)
- Finding secondary structure in: \( b_{t+1} b_{t+2} \ldots b_{n-1} \)

**Bottom Up Dynamic Programming Over Intervals**

**Q.** What order to solve the sub-problems?
**A.** Do shortest intervals first.

```java
RNA \{ b_1, \ldots, b_n \} \{
    for k = 5, 6, \ldots, n-1
    for i = 1, 2, \ldots, n-k
        j = i + k
        Compute \text{M}[i, j]
    return \text{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.

**String Similarity**

**How similar are two strings?**
- occurrence
- occurrence

<table>
<thead>
<tr>
<th>5 mismatches, 1 gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>occurrence</td>
</tr>
<tr>
<td>occurrence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 mismatch, 1 gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>occurrence</td>
</tr>
<tr>
<td>occurrence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0 mismatches, 3 gaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>occurrence</td>
</tr>
<tr>
<td>occurrence</td>
</tr>
</tbody>
</table>

**Edit Distance**

**Applications.**
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

**Edit distance.** [Levenshtein 1966, Needleman-Wunsch 1970]
- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$
- Cost = sum of gap and mismatch penalties.

$\alpha_{cc} + \alpha_{gt} + \alpha_{ag} + 2\alpha_{ca}$

$2\delta + \alpha_{ca}$
Sequence Alignment

**Goal:** Given two strings \( X = x_1 x_2 \ldots x_m \) and \( Y = y_1 y_2 \ldots y_n \) find alignment of minimum cost.

**Def.** An alignment \( M \) is a set of ordered pairs \( x_i - y_j \) such that each item occurs in at most one pair and no crossings.

**Def.** The pair \( x_i - y_j \) and \( x_i - y_j \) are **crosses** if \( i < i' \), but \( j > j' \).

\[
\text{cost}(M) = \sum_{(x_i, y_j) \in M} \alpha_{x_i, y_j} + \sum_{x_i \text{ unmatched}} \delta + \sum_{y_j \text{ unmatched}} \delta
\]

**Ex:** CTACCG vs. TACATG.

**Sol:** \( M = x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_5 \).

---

Sequence Alignment: Problem Structure

**Def.** \( \text{OPT}(i, j) = \text{min cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j \).

- **Case 1:** \( \text{OPT} \) matches \( x_i - y_j \).
  - Pay mismatch for \( x_i - y_j \) + min cost of aligning two strings \( x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j \).
- **Case 2a:** \( \text{OPT} \) leaves \( x_i \) unmatched.
  - Pay gap for \( x_i \) and min cost of aligning \( x_1 x_2 \ldots x_i \) and \( y_1 y_2 \ldots y_j \).
- **Case 2b:** \( \text{OPT} \) leaves \( y_j \) unmatched.
  - Pay gap for \( y_j \) and min cost of aligning \( x_1 x_2 \ldots x_i \) and \( y_1 y_2 \ldots y_{j-1} \).

\[
\text{OPT}(i,j) = \begin{cases} 
  j\delta & \text{if } i = 0 \\
  \alpha_{x_i,y_j} + \text{OPT}(i-1,j-1) & \text{if } j > 0 \\
  \min \left( \begin{array}{c}
    \delta + \text{OPT}(i-1,j) \\
    \delta + \text{OPT}(i,j-1) \\
  \end{array} \right) & \text{otherwise}
\end{cases}
\]

---

Sequence Alignment: Algorithm

```c
Sequence-Alignment(m, n, x_1 x_2 \ldots x_m, y_1 y_2 \ldots y_n, \alpha, \delta) {
  for i = 0 to m
    M[0, i] = i\delta
  for j = 0 to n
    M[j, 0] = j\delta
  for i = 1 to m
    for j = 1 to n
      M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1], \delta + M[i-1, j], \delta + M[i, j-1])
  return M[m, n]
}
```

**Analysis.** \( \Theta(mn) \) time and space.

**English words or sentences:** \( m, n < 10 \).

**Computational biology:** \( m = n = 100,000 \). 10 billions ops OK, but 10GB array?
Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.
- Compute $OPT(i, \cdot)$ from $OPT(i-1, \cdot)$.
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Edit distance graph.
- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Can compute $f(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.

Edit distance graph.
- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and $(m, n)$.
Edit distance graph.

Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.

Can compute $g(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.

Observation 1. The cost of the shortest path that uses $(i, j)$ is $f(i, j) + g(i, j)$.

Observation 2. Let $q$ be an index that minimizes $f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to $(m, n)$ uses $(q, n/2)$.

Sequence Alignment: Linear Space

Divide: find index $q$ that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

Align $x_q$ and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.
Theorem. Let \( T(m, n) = \) max running time of algorithm on strings of length at most \( m \) and \( n \). \( T(m, n) = O(mn \log n) \).

Remark. Analysis is not tight because two sub-problems are of size \((q, n/2)\) and \((m - q, n/2)\). In next slide, we save \( \log n \) factor.

Sequence Alignment: Running Time Analysis Warmup

Theorem. Let \( T(m, n) = \) max running time of algorithm on strings of length at most \( m \) and \( n \). \( T(m, n) = O(mn) \).

Pf. (by induction on \( n \))

\[
T(m, n) \leq 2T(m, n/2) + O(mn) \Rightarrow T(m, n) = O(mn \log n)
\]

Sequence Alignment: Running Time Analysis

Theorem. Let \( T(m, n) = \) max running time of algorithm on strings of length at most \( m \) and \( n \). \( T(m, n) = O(mn) \).

Pf. (by induction on \( n \))

\[
\begin{align*}
T(m, 2) & \leq cm \\
T(2, n) & \leq cn \\
T(m, n) & \leq cmn + T(q, n/2) + T(m - q, n/2)
\end{align*}
\]

\[
T(m, n) \leq T(q, n/2) + T(m - q, n/2) + cmn
\]

\[
\leq 2cm/2 + 2c(m-q)n/2 + cmn
\]

\[
= cqn + cmn - cqn + cmn
\]

\[
= 2cmn
\]