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.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- 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.

Notation.
- Label jobs by finishing time: \( f_1 < f_2 < \ldots < f_n \).
- \( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with \( j \).
- Example: \( p(8) = 5, p(7) = 3, p(2) = 0 \).

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.
**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:** $n$, $s_1, ..., s_n$, $f_1, ..., f_n$, $v_1, ..., v_n$

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

**Compute** $p(1)$, $p(2)$, ..., $p(n)$

**Compute-Opt($j$)**
- if ($j = 0$) return 0
- else return $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$

**Weighted Interval Scheduling: Memoization**

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

**Input:** $n$, $s_1, ..., s_n$, $f_1, ..., f_n$, $v_1, ..., v_n$

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

**Compute** $p(1)$, $p(2)$, ..., $p(n)$

for $j = 1$ to $n$
- $M[j] = \text{empty}$ ← global array
- $M[j] = 0$

**M-Compute-Opt($j$)**
- if ($M[j]$ is empty)
  - $M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$
- 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 $p()$: $O(n)$ after sorting by start time.

- $M$-Compute-Opt$: 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 $M[]$.
  - 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)?

A. 

```
(defun F (n)
  (if (<= n 1)
      n
      (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficient)

```
static int F(int n) {
  if (n <= 1) return n;
  else return F(n-1) + F(n-2);
}
```

Java (exponential)

**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 Find-Solution(n)

Find-Solution(j) {
  if (j = 0)
    output nothing
  else if ($v_j + M[p(j)] > M[j-1]$)
    print j
    Find-Solution(p(j))
  else
    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?

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

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

Tradeoff function: \( E + cL \), for some constant \( c > 0 \).
6.4 Knapsack Problem

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

Ex: \( \{3, 4\} \) has value 40.

\[
\begin{array}{|c|c|c|}
\hline
\text{Item} & \text{Value} & \text{Weight} \\
\hline
1 & 1 & 1 \\
2 & 6 & 2 \\
3 & 18 & 5 \\
4 & 22 & 6 \\
5 & 28 & 7 \\
\hline
\end{array}
\]

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{5, 2, 1\} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.

Running time. \( O(n^2) \). \( \text{can be improved to } O(n^3) \text{ by pre-computing various statistics} \)
- Bottleneck = computing \( e(i, j) \) for \( O(n^2) \) pairs, \( O(n) \) per pair using previous formula.

Knapsack—summarized.
- \( \text{OPT}(j) = \min \{ e(i, j) + c + \text{OPT}(i-1), 0 \} \)
- \( \text{OPT}(0) = 0 \)
- \( \text{OPT}(j) = \min \{ e(i, j) + c + \text{OPT}(i-1), 0 \} \) for \( j = 1, 2, \ldots, n \).

Optimal substructure. If \( i \) is chosen, \( \text{OPT}(j) \) optimal.
- \( \text{OPT}(j) = \min \{ e(i, j) + c + \text{OPT}(i-1), 0 \} \) for \( j = 1, 2, \ldots, n \).
- \( \text{OPT}(0) = 0 \).

Knapsack—summarized.
- \( \text{OPT}(j) = \min \{ e(i, j) + c + \text{OPT}(i-1), 0 \} \) for \( j = 1, 2, \ldots, n \).
- \( \text{OPT}(0) = 0 \).
- Optimal substructure. If \( i \) is chosen, \( \text{OPT}(j) \) optimal.

Algorithm:

\[
\text{OPT}(j) = \min \{ e(i, j) + c + \text{OPT}(i-1), 0 \} \text{ for } j = 1, 2, \ldots, n \\
\text{OPT}(0) = 0
\]

Bottleneck = computing \( e(i, j) \) for \( O(n^2) \) pairs, \( O(n) \) per pair using previous formula.

Knapsack—summarized.
- \( \text{OPT}(j) = \min \{ e(i, j) + c + \text{OPT}(i-1), 0 \} \) for \( j = 1, 2, \ldots, n \).
- \( \text{OPT}(0) = 0 \).
- Optimal substructure. If \( i \) is chosen, \( \text{OPT}(j) \) optimal.

Algorithm:

\[
\text{OPT}(j) = \min \{ e(i, j) + c + \text{OPT}(i-1), 0 \} \text{ for } j = 1, 2, \ldots, n \\
\text{OPT}(0) = 0
\]
Dynamic Programming: False Start

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

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

- 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 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 \( n \times W \) array.

<table>
<thead>
<tr>
<th>Input: ( n, w_1, ..., w_n, v_1, ..., v_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ( w = 0 ) to ( W )</td>
</tr>
<tr>
<td>( M[0, w] = 0 )</td>
</tr>
<tr>
<td>for ( i = 1 ) to ( n )</td>
</tr>
<tr>
<td>for ( w = 1 ) to ( W )</td>
</tr>
<tr>
<td>if ( w_i &gt; w )</td>
</tr>
<tr>
<td>( M[i, w] = M[i-1, w] )</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>( M[i, w] = \max { M[i-1, w], v_i + M[i-1, w-w_i] } )</td>
</tr>
<tr>
<td>return ( M[n, W] )</td>
</tr>
</tbody>
</table>

Knapsack Algorithm

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

\( W = 11 \)

\( \text{OPT: } \{4, 3\} \)

value = 22 + 18 = 40
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**: GUCCGAAUGGGGUAAGCACCGUGCCUACGCGGAGA

complementary base pairs: A-U, C-G

---

6.5 RNA Secondary Structure

**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]

---

**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.
RNA Secondary Structure: Examples

Examples.

**RNA Secondary Structure: Subproblems**

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

![Diagram](image1)

Difficulty. Results in two sub-problems:
- Finding secondary structure in: $b_1b_2\ldots b_{t-1}$.
- Finding secondary structure in: $b_{t+1}b_{t+2}\ldots b_{n-1}$.

![Diagram](image2)

Bottom Up Dynamic Programming Over Intervals

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

```
RNA(b_1\ldots b_n) {
    for k = 5, 6, \ldots, n-1
    for i = 1, 2, \ldots, n-k
        j = i + k
        Compute M[i, j] using recurrence
    return M[1, n]
}
```

Running time. $O(n^2)$.

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 $i = j - 4$.
  - $OPT(i, j) = 0$ by no-sharp turns condition.

- Case 2. Base $b_i$ is not involved in a pair.
  - $OPT(i, j) = OPT(i, \ j-1)$

- Case 3. Base $b_i$ 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 $b_i$ are Watson-Crick complements

**Remark.** Same core idea in CKY algorithm to parse context-free grammars.
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>occurrence</th>
<th>occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 mismatches, 1 gap</td>
<td></td>
</tr>
<tr>
<td>occurrence</td>
<td>occurrence</td>
</tr>
<tr>
<td>1 mismatch, 1 gap</td>
<td></td>
</tr>
<tr>
<td>occurrence</td>
<td>occurrence</td>
</tr>
<tr>
<td>0 mismatches, 3 gaps</td>
<td></td>
</tr>
</tbody>
</table>

Edit Distance

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

- Gap penalty δ; mismatch penalty α_{pq}
- Cost = sum of gap and mismatch penalties.

\[ \alpha_{TC} + \alpha_{CT} + \alpha_{AG} + 2\alpha_{CA} \]

\[ 2\delta + \alpha_{CA} \]
Sequence Alignment

**Goal:** Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$, find the 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 $x_i, y_j'$ cross if $i < i'$, but $j > j'$.

\[
\text{cost}(M) = \sum_{(x_i, y_j) \in M} \alpha_{x_i, y_j} + \sum_{i, y_j \text{ unmatched}} \delta_j + \sum_{j, x_i \text{ unmatched}} \delta_i
\]

**Ex:** CTACCG vs. TACAG.

**Sol:** $M = x_2\cdot y_1, x_3\cdot y_2, x_4\cdot y_3, x_5\cdot y_4, x_6\cdot y_6.$

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x_1 \ldots x_m, y_1 \ldots y_n, \delta, \alpha) {
    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.** $O(mn)$ time and space.

**English words or sentences:** $m, n \leq 10$.

**Computational biology:** $m = n = 100,000$. 10 billions ops OK, but 10GB array?

Sequence Alignment: Problem Structure

**Def.** $\text{OPT}(i, j) = \min$ cost of aligning strings $x_1 x_2 \ldots x_i$ 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 $x_1 x_2 \ldots x_i$ 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 $y_1 y_2 \ldots y_{j-1}$.

- Case 2b: $\text{OPT}$ leaves $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \ldots x$ and $y_1 y_2 \ldots y_j$.

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

6.7 Sequence Alignment in Linear Space
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.
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**

$T(m, n) = 2T(m, n/2) + O(mn) \Rightarrow T(m, n) = O(mn \log n)$

**Sequence Alignment: Running Time Analysis**

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

**Pf.** (by induction on $n$)
- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index $q$.
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant $c$ so that:
  - Base cases: $m = 2$ or $n = 2$.
  - Inductive hypothesis: $T(m, n) \leq 2cmn$.

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

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