6. Dynamic Programming I

- weighted interval scheduling
- segmented least squares
- knapsack problem
- RNA secondary structure

Dynamic programming history

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

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense had pathological fear of mathematical research.
- Bellman sought a “dynamic” adjective to avoid conflict.

Dynamic programming applications

Application areas.

- Computer science: AI, compilers, systems, graphics, theory, ....
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

Some famous dynamic programming algorithms.

- Avidan–Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in TeX.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

Algorithmic paradigms

Greed. Process the input in some order, myopically making irrevocable decisions.

Divide-and-conquer. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.

fancy name for caching intermediate results in a table for later reuse
6. Dynamic Programming I

- weighted interval scheduling
- segmented least squares
- knapsack problem
- RNA secondary structure

Sections 6.1–6.2

Weighted interval scheduling

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight $w_j > 0$.
- Two jobs are compatible if they don’t overlap.
- Goal: find max-weight subset of mutually compatible jobs.

Earliest-finish-time first algorithm

Earliest finish-time first.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.
Weighted interval scheduling

**Convention.** Jobs are in ascending order of finish 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.** \( p(8) = 1, p(7) = 3, p(2) = 0 \).

$$f \leq \text{time} \leq W \text{Y}_{a b c d e f YW YYYY}$$

**j is leftmost interval that ends before \( j \) begins**

![Weighted interval scheduling diagram](image)

**Dynamic programming: binary choice**

**Def.** \( OPT(j) = \text{max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs } 1, 2, \ldots, j \).

**Goal.** \( OPT(n) = \text{max weight of any subset of mutually compatible jobs.} \)

**Case 1.** \( OPT(j) \) does not select job \( j \).
- Must be an optimal solution to problem consisting of remaining jobs \( 1, 2, \ldots, j - 1 \).

**Case 2.** \( OPT(j) \) selects job \( j \).
- Collect profit \( w_j \).
- Can’t use incompatible jobs \{ \( p(j) + 1, p(j) + 2, \ldots, j - 1 \) \}.
- Must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, p(j) \).

**Bellman equation.**

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

Weighted interval scheduling: brute force

**BRUTE-FORCE** \((n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)\)

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

Compute \( p[1], p[2], \ldots, p[n] \) via binary search.

RETURN \( \text{COMPUTE-OPT}(n) \).

**COMPUTE-OPT** \((j)\)

IF \((j = 0)\)
RETURN 0.
ELSE
RETURN \( \max \{ \text{COMPUTE-OPT}(j - 1), w_j + \text{COMPUTE-OPT}(p(j)) \} \).

Dynamic programming: quiz 1

What is running time of \( \text{COMPUTE-OPT}(n) \) in the worst case?

A. \( \Theta(n \log n) \)
B. \( \Theta(n^2) \)
C. \( \Theta(1.618^n) \)
D. \( \Theta(2^n) \)
Weighted interval scheduling: brute force

**Observation.** Recursive algorithm is spectacularly slow because of overlapping subproblems ⇒ exponential-time algorithm.

**Ex.** Number of recursive calls for family of “layered” instances grows like Fibonacci sequence.

---

Weighted interval scheduling: memoization

**Top-down dynamic programming (memoization).**

- Cache result of subproblem $j$ in $M[j]$.
- Use $M[j]$ to avoid solving subproblem $j$ more than once.

\[
\text{TOP-DOWN}(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)
\]

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

\[
M[0] \leftarrow 0.
\]

**RETURN** $M$-COMPUTE-OPT($n$).

\[
\text{M-COMPUTE-OPT}(j)
\]

If $(M[j]$ is uninitialized)

\[
M[j] \leftarrow \max \{ M-\text{COMPUTE-OPT}(j-1), w_j + M-\text{COMPUTE-OPT}(p[j]) \}.
\]

**RETURN** $M[j]$.

---

Weighted interval scheduling: running time

**Claim.** Memoized version of algorithm takes $O(n \log n)$ time.

**Pf.**

- Sort by finish time: $O(n \log n)$ via mergesort.
- Compute $p[j]$ for each $j$: $O(n \log n)$ via binary search.

- $M$-COMPUTE-OPT($j$): each invocation takes $O(1)$ time and either
  - (1) returns an initialized value $M[j]$
  - (2) initializes $M[j]$ and makes two recursive calls

- Progress measure $\Phi = \#$ initialized entries among $M[1..n]$.
  - initially $\Phi = 0$; throughout $\Phi \leq n$.
  - (2) increases $\Phi$ by 1 ⇒ $\leq 2n$ recursive calls.

- Overall running time of $M$-COMPUTE-OPT($n$) is $O(n)$.

---

Those who cannot remember the past are condemned to repeat it.

- Dynamic Programming
Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find optimal solution?
A. Make a second pass by calling \textsc{Find-Solution}(n).

\begin{verbatim}
\textsc{Find-Solution}(j)
\begin{align*}
&\text{If } ( j = 0 ) \\
&\quad \text{Return } \emptyset . \\
&\text{Else if } ( w_j + M[p[j]] > M[j-1] ) \\
&\quad \text{Return } ( j ) \cup \textsc{Find-Solution}(p[j]). \\
&\text{Else} \\
&\quad \text{Return } \textsc{Find-Solution}(j-1). \\
&\end{align*}

M[j] = \max \{ M[j-1], w_j + M[p[j]] \}.
\end{verbatim}

\textbf{Analysis}. \# of recursive calls \( n \Rightarrow O(n) \).

---

### Maximum Subarray Problem

\textbf{Goal}. Given an array \( x \) of \( n \) integer (positive or negative), find a contiguous subarray whose sum is maximum.

\begin{align*}
&187
\end{align*}

\textbf{Applications}. Computer vision, data mining, genomic sequence analysis, technical job interviews, ....

---

### Kadane’s Algorithm

\textbf{Def}. \( OPT(i) = \max \text{ sum of any subarray of } x \text{ whose rightmost index is } i. \)

\textbf{Goal}. \( \max_i OPT(i) \)

\textbf{Bellman equation}. \( OPT(i) = \begin{cases} 
  x_i & \text{if } i = 1 \\
  \max \{ x_i, x_i + OPT(i-1) \} & \text{if } i > 1
\end{cases} \)

\textbf{Running time}. \( O(n). \)
**Maximum rectangle problem**

**Goal.** Given an \( n \times n \) matrix \( A \), find a rectangle whose sum is maximum.

\[
A = \begin{bmatrix}
-2 & 5 & 0 & -5 & -2 & 2 & -3 \\
4 & -3 & -1 & 3 & 2 & 1 & -1 \\
-5 & 6 & 3 & -5 & -1 & -4 & -2 \\
-1 & -1 & 3 & -1 & 4 & 1 & 1 \\
3 & -3 & 2 & 0 & 3 & -3 & -2 \\
-2 & 1 & -2 & 1 & 1 & 3 & -1 \\
2 & -4 & 0 & 1 & 0 & -3 & -1
\end{bmatrix}
\]

**Applications.** Databases, image processing, maximum likelihood estimation, technical job interviews, ...

---

**6. Dynamic Programming I**

- weighted interval scheduling
- segmented least squares
- knapsack problem
- RNA secondary structure

---

**Least squares**

**Least squares.** Foundational problem in statistics.

- Given \( n \) points in the plane: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\).
- Find a line \( y = ax + b \) that minimizes the sum of the squared error:

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

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

\[
a = \frac{n \sum x \sum y - (\sum x)(\sum y)}{n \sum x^2 - (\sum x)^2}, \quad b = \frac{\sum y - a \sum x}{n}
\]
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 is a reasonable choice for $f(x)$ to balance accuracy and parsimony?

\[ y \]
\[ x \]

Dynamic programming: multiway choice

Notation.
- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e_{ij}$ = SSE for 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 $i \leq j$.
- Cost = $e_{ij} + c + OPT(i-1)$.

Bellman equation.

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\min_{1 \leq i \leq j} \{ e_{ij} + c + OPT(i-1) \} & \text{if } j > 0 
\end{cases}
\]
Segmented least squares analysis

**Theorem.** [Bellman 1961] DP algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

**Pf.** Bottleneck = computing SSE $e_{ij}$ for each $i$ and $j$.

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{\sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

- $O(n)$ to compute $e_{ij}$.

**Remark.** Can be improved to $O(n^2)$ time.

- For each $i$: precompute cumulative sums $\sum_k x_k$, $\sum_k y_k$, $\sum_k x_k^2$, $\sum_k x_k y_k$.

- Using cumulative sums, can compute $e_{ij}$ in $O(1)$ time.

Knapsack problem

**Goal.** Pack knapsack so as to maximize total value.

- There are $n$ items: item $i$ provides value $v_i > 0$ and weighs $w_i > 0$.
- Knapsack has weight capacity of $W$.

**Assumption.** All input values are integral.

**Ex.** {1, 2, 5} has value $35$ (and weight 10).

**Ex.** {3, 4} has value $40$ (and weight 11).

<table>
<thead>
<tr>
<th></th>
<th>$v_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1$</td>
<td>$1$ kg</td>
</tr>
<tr>
<td>2</td>
<td>$6$</td>
<td>$2$ kg</td>
</tr>
<tr>
<td>3</td>
<td>$18$</td>
<td>$5$ kg</td>
</tr>
<tr>
<td>4</td>
<td>$22$</td>
<td>$6$ kg</td>
</tr>
<tr>
<td>5</td>
<td>$28$</td>
<td>$7$ kg</td>
</tr>
</tbody>
</table>

**Dynamic programming: adding a new variable**

**Def.** $OPT(i, w) = \max$ value of any subset of items $1, \ldots, i$ with weight limit $w$.

**Goal.** $OPT(n, W)$.

**Case 1.** $OPT(i, w)$ does not select item $i$.

- $OPT(i, w)$ selects best of $\{1, 2, \ldots, i-1\}$ using weight limit $w$.

**Case 2.** $OPT(i, w)$ selects item $i$.

- Collect value $v_i$.
- New weight limit $= w - w_i$.
- $OPT(i, w)$ selects best of $\{1, 2, \ldots, i-1\}$ using this new weight limit.

**Bell equation.**

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i \leq w \\ \max \{ OPT(i-1, w), v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$
Knapsack problem: bottom-up dynamic programming

Knapsack \( (n, W, w_1, \ldots, w_n, v_1, \ldots, v_n) \)

For \( w = 0 \) to \( W \)
\[
M[0, w] \leftarrow 0.
\]

For \( i = 1 \) to \( n \)
For \( w = 0 \) to \( W \)
If \( (w_i > w) \) \( M[i, w] \leftarrow M[i-1, w] \).
Else \( M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \} \).\
Return \( M[n, W] \).

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

Knapsack problem: running time

Theorem. The DP algorithm solves the knapsack problem with \( n \) items and maximum weight \( W \) in \( O(nW) \) time and \( \Theta(nW) \) space.

Pf.
- Takes \( O(1) \) time per table entry.
- There are \( \Theta(nW) \) table entries.
- After computing optimal values, can trace back to find solution:
  \( OPT(i, w) \) takes item \( i \) iff \( M[i, w] > M[i-1, w] \).

Remarks.
- Algorithm depends critically on assumption that weights are integral.
- Assumption that values are integral was not used.

Dynamic programming: quiz 4

Does there exist a poly-time algorithm for the knapsack problem?

A. Yes, because the DP algorithm takes \( \Theta(nW) \) time.
B. No, because \( \Theta(nW) \) is not a polynomial function of the input size.
C. No, because the problem is \( NP \)-hard.
D. Unknown.
**Coin Changing**

**Problem.** Given \( n \) coin denominations \( \{ c_1, c_2, \ldots, c_n \} \) and a target value \( V \), find the fewest coins needed to make change for \( V \) (or report impossible).

**Recall.** Greedy cashier’s algorithm is optimal for U.S. coin denominations, but not for arbitrary coin denominations.

**Ex.** \{ 1, 10, 21, 34, 70, 100, 350, 1295, 1500 \}.

**Optimal.** \( 140€ = 70 + 70 \).

---

**Coin Changing**

**Def.** \( OPT(v) = \min \) number of coins to make change for \( v \).

**Goal.** \( OPT(V) \).

**Multiway choice.** To compute \( OPT(v) \).

- Select a coin of denomination \( c_i \) for some \( i \).
- Select fewest coins to make change for \( v - c_i \).

**Bellman equation.**

\[
OPT(v) = \begin{cases} 
\infty & \text{if } v < 0 \\
0 & \text{if } v = 0 \\
\min_{1 \leq i \leq n} \{ 1 + OPT(v - c_i) \} & \text{otherwise}
\end{cases}
\]

**Running time.** \( O(nV) \).

---

**RNA secondary structure**

**RNA.** String \( B = b_1b_2\ldots b_n \) 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.
**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 $i < k < j < l$.

\begin{align*}
B &= ACUUGGCCAU \\
S &= \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}
\end{align*}

**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.
- [Non-crossing] If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs in $S$, then we cannot have $i < k < j < l$.

\begin{align*}
B &= AUGUGGCCAU \\
S &= \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}
\end{align*}
**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 $(b_k, b_\ell)$ are two pairs in $S$, then we cannot have $i < k < j < \ell$.

**Free-energy hypothesis.** RNA molecule will form the secondary structure with the minimum total free energy.

approximate by number of base pairs
(more base pairs $\Rightarrow$ lower free energy)

*Goal.* Given an RNA molecule $B = b_1b_2…b_n$, find a secondary structure $S$ that maximizes the number of base pairs.

**Dynamic programming: quiz 5**

Is the following a secondary structure?

A. Yes.
B. No, violates Watson–Crick condition.
C. No, violates no-sharp-turns condition.
D. No, violates no-crossing condition.

**Dynamic programming: quiz 6**

Which subproblems?

A. $OPT(j) =$ max number of base pairs in secondary structure of the substring $b_1b_2…b_j$.
B. $OPT(j) =$ max number of base pairs in secondary structure of the substring $b_jb_{j+1}…b_n$.
C. Either A or B.
D. Neither A nor B.

**RNA secondary structure: subproblems**

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

*Goal.* $OPT(n)$.

*Choice.* Match bases $b_i$ and $b_\ell$.

**Difficulty.** Results in two subproblems (but one of wrong form).
- Find secondary structure in $b_1b_2…b_{j-1}$.
- Find secondary structure in $b_{i+1}b_{i+2}…b_{j-1}$.

need more subproblems
(first base no longer $b_i$)
Dynamic programming over intervals

**Def.** \( OPT(i, j) \) = 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 \( b_i \) for some \( i \leq t < j - 4 \).
- Non-crossing condition decouples resulting two subproblems.
- \( OPT(i, j) = 1 + \max \{ OPT(i, t - 1) + OPT(t + 1, j - 1) \} \).

![Diagram of dynamic programming over intervals]

In which order to compute \( OPT(i, j) \)?

- **A.** Increasing \( i \), then \( j \).
- **B.** Increasing \( j \), then \( i \).
- **C.** Either A or B.
- **D.** Neither A nor B.

Bottom-up dynamic programming over intervals

**Q.** In which order to solve the subproblems?
- **A.** Do shortest intervals first—increasing order of \( |j - i| \).

**RNA-SECONDARY-STRUCTURE(\( n, b_1, \ldots, b_n \))**

```
FOR \( k = 5 \) TO \( n - 1 \)
    FOR \( i = 1 \) TO \( n - k \)
        \( j \leftarrow i + k \).
        Compute \( M[i, j] \) using formula.
    RETURN \( M[1, n] \).
```

**Theorem.** The DP algorithm solves the RNA secondary structure problem in \( O(n^3) \) time and \( O(n^2) \) space.

Dynamic programming summary

**Outline.**
- Define a collection of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from “smallest” to “largest” that enables determining a solution to a subproblem from solutions to smaller subproblems.

**Techniques.**
- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Adding a new variable: knapsack problem.
- Intervals: RNA secondary structure.

**Top-down vs. bottom-up dynamic programming.** Opinions differ.