Chapter 6
Dynamic Programming
Topics

• Dynamic programming overview
• Dynamic programming examples
  – Weighted interval scheduling
  – Segmented squares
  – Knapsack problem
Algorithmic Paradigms

**Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

**Divide-and-conquer.** Break up a problem into 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. [1950s] Pioneered the systematic study of dynamic programming.

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, compilers, systems, ....

Some famous dynamic programming algorithms.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic 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

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: \( p(8) = 5, p(7) = 3, p(2) = 0. \)
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.
  - collect profit \( v_j \)
  - 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

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

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

Compute-Opt(\( j \)) {
  if (\( j = 0 \))
    return 0
  else
    return \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}
**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

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

$$p(1) = 0, \; p(j) = j-2$$
Weighted Interval Scheduling: Memoization

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

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

```
for j = 1 to n
    M[j] = empty  # global array
M[0] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max($v_j + M$-Compute-Opt($p(j)$), $M$-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 $p(\cdot)$: $O(n \log n)$ via 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 $M[]$.
  - 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.
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 ≤ n ⇒ O(n).
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 $n$
    $M[j] = \max(v_j + M[p(j)], M[j-1])$
}

6.3 Segmented Least Squares
Segmented Least Squares

Least squares.
- Foundational problem in statistic and numerical analysis.
- 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

\[
a = \frac{n \sum_i x_i y_i - (\sum_i x_i) (\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}
\]
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?

Diagram: A graph showing the trade-off between the number of lines and goodness of fit.
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:
  - the sum of the sums of the squared errors \( E \) in each segment
  - the number of lines \( L \)
- Tradeoff function: \( E + cL \), for some constant \( c > 0 \).
Dynamic Programming: Multiway Choice

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

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

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e(i, j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$
Segmented Least Squares: Algorithm

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

Segmented-Least-Squares() {
    \( M[0] = 0 \)
    for \( j = 1 \) to \( n \)
        for \( i = 1 \) to \( j \)
            compute the least square error \( e_{ij} \) for the segment \( p_i, \ldots, p_j \)

    for \( j = 1 \) to \( n \)
        \( M[j] = \min_{1 \leq i \leq j} (e_{ij} + c + M[i-1]) \)

    return \( M[n] \)
}

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

- Bottleneck = computing \( e(i, j) \) for \( O(n^2) \) pairs, \( O(n) \) per pair using previous formula.
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>#</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, ..., $i-1$ \}

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

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

- **Case 2:** $OPT$ selects item $i$.
  - new weight limit = $w - w_i$
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$ 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.

\[
\text{Input: } n, W, w_1,\ldots,w_N, v_1,\ldots,v_N
\]

\[
\text{for } w = 0 \text{ to } W \\
\quad M[0, w] = 0
\]

\[
\text{for } i = 1 \text{ to } n \\
\quad \text{for } w = 1 \text{ to } W \\
\quad \quad \text{if } (w_i > w) \\
\quad \quad \quad M[i, w] = M[i-1, w] \\
\quad \quad \text{else} \\
\quad \quad \quad M[i, w] = \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}
\]

\text{return } M[n, W]
Knapsack Algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\phi)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>19</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>24</td>
<td>28</td>
<td>29</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>28</td>
<td>29</td>
<td>34</td>
<td>34</td>
<td>40</td>
</tr>
</tbody>
</table>

OPT: \{4, 3\}
value = 22 + 18 = 40

\[ W = 11 \]

<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>
Knapsack Problem: Running Time

**Running time.** $\Theta(n W)$.
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

**Knapsack approximation algorithm.** There exists a poly-time 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

**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:** GUCGAUUGAGCGAAUGUAACAAACGUGGCUACGCGGAGA

![RNA Secondary Structure Diagram]

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.

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.

- Base pair
- ok
- sharp turn
- 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: \( b_1b_2...b_{t-1} \).
- Finding secondary structure in: \( b_{t+1}b_{t+2}...b_{n-1} \).

\( \text{OPT}(t-1) \) ← need more sub-problems
Dynamic Programming Over Intervals

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

- **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
  - $\text{OPT}(i, j) = 1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \}$

  \[
  \text{take max over } t \text{ such that } i \leq t < j-4 \text{ and } b_t \text{ and } b_j \text{ are Watson-Crick complements}
  \]

**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.

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

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.
6.6 Sequence Alignment
String Similarity

How similar are two strings?

- occurrence
- occurrence

6 mismatches, 1 gap
1 mismatch, 1 gap
0 mismatches, 3 gaps
Edit Distance

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

- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.

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

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

**Goal:** Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find an 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 $j > j'$.

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

**Ex:** CTACCG vs. TACATG.

**Sol:** $M = x_2$-$y_1$, $x_3$-$y_2$, $x_4$-$y_3$, $x_5$-$y_4$, $x_6$-$y_6$. 
Sequence Alignment: Problem Structure

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

- **Case 1:** \( OPT \) matches \( x_i-y_j \).
  - pay mismatch for \( x_i-y_j \) + min cost of aligning two strings \( x_1 x_2 \ldots x_{i-1} \) and \( y_1 y_2 \ldots y_{j-1} \)

- **Case 2a:** \( OPT \) leaves \( x_i \) unmatched.
  - pay gap for \( x_i \) and min cost of aligning \( x_1 x_2 \ldots x_{i-1} \) and \( y_1 y_2 \ldots y_j \)

- **Case 2b:** \( OPT \) leaves \( y_j \) unmatched.
  - pay gap for \( y_j \) and min cost of aligning \( x_1 x_2 \ldots x_i \) and \( y_1 y_2 \ldots y_{j-1} \)

\[
OPT(i, j) = \begin{cases} 
  j\delta & \text{if } i = 0 \\
  \min \left\{ \begin{array}{l}
    \alpha_{x_i y_j} + OPT(i-1, j-1) \\
    \delta + OPT(i-1, j) \\
    \delta + OPT(i, j-1)
  \end{array} \right. & \text{otherwise} \\
  i\delta & \text{if } j = 0
\end{cases}
\]
Sequence Alignment: Algorithm

```plaintext
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n, δ, α) {
    for i = 0 to m
        M[i, 0] = iδ
    for j = 0 to n
        M[0, j] = jδ

    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(α[x_i, y_j] + M[i-1, j-1],
                           δ + M[i-1, j],
                           δ + M[i, j-1])

    return M[m, n]
}
```

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

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

Computational biology: \( m = n = 100,000 \). 10 billions ops OK, but 10GB array?
6.7 Sequence Alignment in Linear Space
Q. Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.
  ■ Compute $\text{OPT}(i, \cdot)$ from $\text{OPT}(i-1, \cdot)$.
  ■ 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.
Sequence Alignment: Linear Space

Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0, 0)$ to $(i, j)$.
- Observation: $f(i, j) = \text{OPT}(i, j)$. 

![Diagram showing edit distance graph with nodes $x_1$, $x_2$, $x_3$, $\varepsilon$, $y_1$, $y_2$, $y_3$, $y_4$, $y_5$, $y_6$, and $m-n$. Edges with weight $\alpha_{x_i y_j}$ and $\delta$.]
Sequence Alignment: Linear Space

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.
Sequence Alignment: Linear 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 $(m, n)$
Sequence Alignment: Linear Space

**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 $(q, n/2)$. 
**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.
Sequence Alignment: Running Time Analysis Warmup

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

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

**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

**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:

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

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2cmn$.

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