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; typically each subproblem is solved just once, and the solution is stored (i.e. cached).

The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup. The technique of storing solutions to subproblems instead of recomputing them is called memoization.
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 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.
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 $1, 2, ..., j-1$

$$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + OPT(p(j)), \, OPT(j-1) \} & \text{otherwise}
\end{cases}$$
Weighted Interval Scheduling: Brute Force

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 \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}

This algorithm can take exponential time in the worst-case, due to the potential redundancy in the recursive calls.
**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.

\[
p(1) = 0, \quad p(j) = j-2
\]
**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 \( p(1), p(2), \ldots, p(n) \)

```plaintext
for j = 1 to n
    M[j] = empty
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).
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 {
    \[ M[0] = 0 \]
    \[ \text{for } j = 1 \text{ to } n \]
    \[ \quad M[j] = \max(v_j + M[p(j)], M[j-1]) \]
}
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.

- 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?
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.
- \( OPT(j) = \) 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 = \( e(i, j) + c + OPT(i-1) \).

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

\textbf{INPUT:} n, p_1, …, p_N, c

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

Greedy: repeatedly add item with maximum ratio \( \frac{v_i}{w_i} \).

Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Dynamic Programming: False Start

**Def.** $OPT(i) =$ max profit subset of items $1, \ldots, i$.

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

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

- **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 \( \{ 1, 2, \ldots, i-1 \} \) 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-by-W array.

Input: \( n, W, w_1, \ldots, w_N, v_1, \ldots, v_N \)

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

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

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 poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]
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

<table>
<thead>
<tr>
<th></th>
<th>occurrence</th>
<th>occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>oc</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

6 mismatches, 1 gap

<table>
<thead>
<tr>
<th></th>
<th>occurrence</th>
<th>occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>oc</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

1 mismatch, 1 gap

<table>
<thead>
<tr>
<th></th>
<th>occurrence</th>
<th>occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>oc</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>e</td>
</tr>
</tbody>
</table>

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.

\[
\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA} \quad \text{vs.} \quad 2\delta + \alpha_{CA}
\]
**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 $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

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

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

- **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 $y_1 y_2 \ldots y_{j-1}$

$$OPT(i, j) = \begin{cases} 
  j\delta & \text{if } i = 0 \\
  \min \begin{cases} 
    \alpha_{x_i y_j} + OPT(i-1, j-1) \\
    \delta + OPT(i-1, j) \\
    \delta + OPT(i, j-1) \\
    i\delta & \text{if } j = 0 
  \end{cases} & \text{otherwise}
\end{cases}$$
Sequence Alignment: Algorithm

```c
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 $OPT(i, \cdot)$ 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.
Sequence Alignment: Linear Space

Edit distance graph.
- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Observation: $f(i, j) = \text{OPT}(i, j)$  (can prove this by induction on $i+j$)
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 $O(m)$ space.
*Note: we want to recover solution for best alignment in the end; this will require $O(m+n)$ 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 $O(m)$ space. (just like we did for $f()$).
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 $n$. $T(m, n) = O(mn \log n)$.

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

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

$$
T(m, 2) \leq cm \\
T(2, n) \leq cn \\
T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)
$$
#1: Let $G=(V,E)$ be an undirected graph with $n$ nodes. Recall that a subset of the nodes is called an independent set if no two of them are joined by an edge. Finding large independent sets is difficult in general (NP-hard); but here we’ll see that it can be done efficiently if the graph is “simple” enough.

Call a graph $G$ a path if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between $v_i$ and $v_j$ iff the indices differ by one. With each node $v_i$, we associate a positive integer weight $w_i$.

Goal: Find an independent set in a path $G$ whose total weight is as large as possible.
#1: **Goal:** Find an independent set in a path $G$ whose total weight is as large as possible.

(a) Give an example to show that the following algorithm doesn’t always find an independent set of max total weight.

**Algo:** “heaviest-first”

```plaintext
{ 
    S=empty set 
    while (some node remains in $G$) 
    { 
        Pick node with max weight and add it to $S$; delete this node and its neighbors 
    } 
}
```
#1: **Goal:** Find an independent set in a path $G$ whose total weight is as large as possible.

(c) Give an algorithm that takes an $n$-node path $G$ with weights and returns an independent set of maximum total weight. The running time should be polynomial in $n$, independent of the values of the weights.

(Use a DP structure; start with a recursion.)
#1: Goal: Find an independent set in a path \( G \) whose total weight is as large as possible.

(c) Give an algorithm that takes an \( n \)-node path \( G \) with weights and returns an independent set of maximum total weight. The running time should be polynomial in \( n \), independent of the values of the weights.

Let \( S_i \) denote an independent set on \( \{v_1, \ldots, v_i\} \), and let \( X_i \) denote its weight. Define \( X_0 = 0 \) and note that \( X_1 = w_1 \).

Now for \( i > 1 \), either \( v_i \) belong to \( S_i \) or it doesn't. What is a natural recursion?
#1: **Goal:** Find an independent set in a path $G$ whose total weight is as large as possible.

(c) **Give an algorithm** that takes an $n$-node path $G$ with weights and returns an independent set of maximum total weight. The running time should be polynomial in $n$, independent of the values of the weights.

Let $S_i$ denote an independent set on $\{v_1,..,v_i\}$, and let $X_i$ denote its weight. Define $X_0=0$ and note that $X_1=w_1$.

Now for $i>1$, either $v_i$ belongs to $S_i$ or it doesn’t. What is a natural recursion?

If $v_i$ belongs to $S_i$, then $v_{i-1}$ doesn’t (why?); so either $X_i=w_i+X_{i-2}$ or $X_i=X_{i-1}$ (why?).
#1: Goal: Find an independent set in a path $G$ whose total weight is as large as possible.

(c) Give an algorithm that takes an $n$-node path $G$ with weights and returns an independent set of maximum total weight. The running time should be polynomial in $n$, independent of the values of the weights.

Let $S_i$ denote an independent set on $\{v_1, \ldots, v_i\}$, and let $X_i$ denote its weight. Define $X_0=0$ and note that $X_1=w_1$.

Now for $i>1$, either $v_i$ belongs to $S_i$ or it doesn’t. What is a natural recursion?

If $v_i$ belongs to $S_i$, then $v_{i-1}$ doesn’t (why?); so either $X_i=w_i+X_{i-2}$ or $X_i=X_{i-1}$ (why?).

Hence, $X_i=\max(X_{i-1}, w_i+X_{i-2})$; what’s the run-time to compute $S_n$?
#3: Let $G=(V,E)$ be a directed graph with nodes $v_1,...,v_n$. We say that $G$ is an ordered graph if it has the following properties:

(i) Each edge goes from a node with a lower index to a node with a higher index; i.e., each directed edge has the form $(v_i,v_j)$ with $i<j$.

(ii) Each node except $v_n$ has at least one edge leaving it. That is, for every node $v_i$, $i=1,2,...,n-1$ there is at least one edge of the form $(v_i,v_j)$.

The length of a path is the number of edges it contains. The goal is to solve:

*Given an ordered graph $G$, find the length of the longest path that begins at $v_1$ and ends at $v_n$.***
#3: The length of a path is the number of edges it contains. The goal is to solve:

Given an ordered graph $G$, find the length of the longest path that begins at $v_1$ and ends at $v_n$.

(b) Give an efficient algorithm that takes an ordered graph $G$ and returns the length of the longest path that begins at $v_1$ and ends at $v_n$.

Idea: Use DP structure; consider subproblems $OPT[i]$, the length of the longest path from $v_1$ to $v_i$ in ordered graph, $G$. One caveat: not all nodes $v_i$ necessarily have a path from $v_1$ to $v_i$; let’s use the value “-inf” in this case.
#3: (b) Give an efficient algorithm that takes an ordered graph $G$ and returns the length of the longest path that begins at $v_1$ and end at $v_n$.

Idea: Use DP structure; consider subproblems $OPT[i]$, the length of the longest path from $v_1$ to $v_i$ in ordered graph, $G$. One caveat: not all nodes $v_i$ necessarily have a path from $v_1$ to $v_i$; let's use the value “-inf” in this case.

Define $OPT[1]=0$ (base case); use for loop:

$M[1]=0$

for $i=2,..,n$

{  
  $M=-\infty$
  
  for all edges $(j,i)$ in $G$

  {  
    if $M[j]=-\infty$ && $M<M[j]+1$
      
      then $M=M[j]+1$

      /endif

    } /end for

  $M[i]=M$

} /endfor

Kleinberg 6.3 (HW)
#3: (b) Give an efficient algorithm that takes an ordered graph $G$ and returns the length of the longest path that begins at $v_1$ and ends at $v_n$.

Idea: Use DP structure; consider subproblems $OPT[i]$, the length of the longest path from $v_1$ to $v_i$ in ordered graph, $G$. One caveat: not all nodes $v_i$ necessarily have a path from $v_1$ to $v_i$; let’s use the value “-inf” in this case.

Define $OPT[1]=0$ (base case); use for loop:

```plaintext
for i=2,...,n
    { 
        M=-inf
        for all edges (j,i) in G
            { 
                if $M=\text{-inf}$ && $M<$M[j]+1
                    then $M=M[j]+1$
                /endif
            } /end for
        M[i]=M
    } /endfor
```

What’s the run time?
#3:(b) Give an efficient algorithm that takes an ordered graph $G$ and returns the length of the longest path that begins at $v_1$ and end at $v_n$.

Idea: Use DP structure; consider subproblems $OPT[i]$, the length of the longest path from $v_1$ to $v_i$ in ordered graph, $G$. **One caveat:** not all nodes $v_i$ necessarily have a path from $v_1$ to $v_i$; let’s use the value “$-\infty$” in this case.

Define $OPT[1]=0$ (base case); use for loop:

```plaintext
for i=2,...,n
{
    M=-\infty
    for all edges (j,i) in G
    {
        if $M=-\infty$ && $M>M[j]+1$
            then $M=M[j]+1$
        /endif
    } /end for
    M[i]=M
} /end for
```

What’s the run time? $O(n^2)$
Kleinberg 6.6 (HW)

This problem is very similar in flavor to the segmented least squares problem. We observe that the last line ends with word \( w_n \) and has to start with some word \( w_j \); breaking off words \( w_j, \ldots, w_n \) we are left with a recursive sub-problem on \( w_1, \ldots, w_{j-1} \).

Thus, we define \( \text{OPT}[v] \) to be the value of the optimal solution on the set of words \( W_i = \{w_1, \ldots, w_i\} \). For any \( i \leq j \), let \( S_{i,j} \) denote the slack of a line containing the words \( w_i, \ldots, w_j \); as a notational device, we define \( S_{i,j} = \infty \) if these words exceed total length \( L \). For each fixed \( i \), we can compute all \( S_{i,j} \) in \( O(n) \) time by considering values of \( j \) in increasing order; thus, we can compute all \( S_{i,j} \) in \( O(n^2) \) time.

As noted above, the optimal solution must begin the last line somewhere (at word \( w_j \)), and solve the sub-problem on the earlier lines optimally. We thus have the recurrence

\[
\text{OPT}[n] = \min_{1 \leq j \leq n} S_{i,n}^2 + \text{OPT}[j - 1],
\]

and the line of words \( w_j, \ldots, w_n \) is used in an optimum solution if and only if the minimum is obtained using index \( j \).

Finally, we just need a loop to build up all these values:

Compute all values \( S_{i,j} \) as described above.
Set \( \text{OPT}[0] = 0 \)
For \( k = 1, \ldots, n \)
    \[
    \text{OPT}[k] = \min_{1 \leq j \leq k} \left( S_{j,k}^2 + \text{OPT}[j - 1] \right)
    \]
Endfor
Return \( \text{OPT}[n] \).

As noted above, it takes \( O(n^2) \) time to compute all values \( S_{i,j} \). Each iteration of the loop takes time \( O(n) \), and there are \( O(n) \) iterations. Thus the total running time is \( O(n^2) \).

By tracing back through the array \( \text{OPT} \), we can recover the optimal sequence of line breaks that achieve the value \( \text{OPT}[n] \) in \( O(n) \) additional time.
#11: Suppose you’re consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next \( n \) weeks, they have a projected supply \( s_i \) of equipment (measured in pounds), which has to be shipped by an air freight carrier.

Each week's supply can be carried by one of two air freight companies: A or B.

(*) Company A charges a fixed rate \( r \) per pound (so it costs \( r \times s_i \) to ship a week's supply \( s_i \)).

(*) Company B makes contracts for a fixed amount \( c \) per week, independent of the weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.
Kleinberg 6.11 (HW)

#11: Suppose you’re consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next n weeks, they have a projected supply $s_i$ of equipment (measured in pounds), which has to be shipped by an air freight carrier.

Each week’s supply can be carried by one of two air freight companies: A or B.

(*) Company A charges a fixed rate $r$ per pound (so it costs $r \cdot s_i$ to ship a week’s supply $s_i$).

(*) Company B makes contracts for a fixed amount $c$ per week, independent of the weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A schedule, for the PC company, is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks at a time. The cost of a schedule is the total amount paid to company A and B.
Kleinberg 6.11 (HW)

#11: Give a polynomial-time algorithm that takes a sequence of supply values \( s_1, s_2, \ldots, s_n \) and returns a schedule of minimum cost.

**Example.** Suppose \( r=1, c=10 \), and the sequence of values is:

11,9,9,12,12,12,12,9,9,11.

Then the optimal schedule would be to choose company A for first three weeks, then company B for a block of four consecutive weeks, and then company A for the final three weeks.

How to use DP structure to solve?
Kleinberg 6.11 (HW)

#11: Give a polynomial-time algorithm that takes a sequence of supply values $s_1, s_2, \ldots, s_n$ and returns a schedule of minimum cost.

How to use DP structure to solve?

Let $OPT[i]$ denote the minimum cost of a solution for weeks 1 through $i$. In an optimal solution, we either use company A or B for the $i$th week.

If we use company A, we pay $r \times s_i$ and behave optimally up through week $i-1$; else we use company B and pay $4c$ for this contract, and we behave optimally through week $i-4$. 
Kleinberg 6.11 (HW)

#11: Give a polynomial-time algorithm that takes a sequence of supply values $s_1, s_2, ..., s_n$ and returns a schedule of minimum cost.

How to use DP structure to solve?

Let $OPT[i]$ denote the minimum cost of a solution for weeks 1 through $i$. In an optimal solution, we either use company A or B for the $i$th week.

If we use company A, we pay $r * s_i$ and behave optimally up through week $i-1$; else we use company B and pay $4c$ for this contract, and we behave optimally through week $i-4$.

In summary, our recursion is as follows: $OPT[i] = \min(r * s_i + OPT(i-1), 4c + OPT(i-4))$.

What’s the runtime?