Administrative Stuff

Homeworks:
- HW 1 - out of 95 points (I can't add...)
- Average was approx 59 (± late day grades)
- Grades will be scaled!! Do not worry about the numerical value!!

Notes from TAs:
- Please type homeworks if possible. If not, please write legibly.
- WHENEVER you are presenting an algorithm, the grading guidelines will be used. This includes:
  - Proof of correctness
  - Runtime analysis

Other stuff:
- Textbook on reserve at Oldegaard.
- Pair Programming
**Algorithmic Paradigms**

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

*Dynamic programming.* Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

*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 History**

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

**Etymology.**
- Dynamic programming = planning over time.

"It's impossible to use dynamic in a pejorative sense"

### Dynamic Programming Applications

**Areas.**
- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.

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

- **Case 1:** \( \text{OPT} \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., \( j-1 \)

- **Case 2:** \( \text{OPT} \) 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, ..., \( p(j) \)

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

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

Observation. Recursive algorithm is exponential. (No better than brute force!)
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
  \( M[0] = 0 \)

\[ M\text{-Compute-Opt}(j) \{ \]
  \( \text{if} \) \( M[j] \text{ is empty} \)
  \( M[j] = \max( v_j + M\text{-Compute-Opt}(p(j)), \]
  \( M\text{-Compute-Opt}(j-1) ) \)
  \( \text{return} \ M[j] \)
\[ \}

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.

- \( M\text{-Compute-Opt}(j) \): 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 \( \implies \) at most \( 2n \) recursive calls.

- Overall running time of \( M\text{-Compute-Opt}(n) \) is \( O(n) \).
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 $n$
    $M[j] = \max(v_j + M[p(j)], M[j-1])$
}

We will always use bottom-up dynamic programming
- Easier to analyze runtime
- Easier to prove correctness (strong induction!)

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($n$) or Iterative-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 $\leq n \Rightarrow O(n)$. 

Weighted Interval Scheduling: Finding a Solution

A. Build it into your program.

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

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n

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

Iterative-Compute-Opt2 {
    M[0] = 0
    for j = 1 to n
        if (v_j + M[p(j)]) > M[j-1])
            M[j] = v_j + M[p(j)]
            Opt-Traceback[j] = p(j)
        else
            M[j] = M[j-1]
            Opt-Traceback[j] = j-1
    }
}

A. Must still do some post-processing.

Run Iterative-Compute-Opt2(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (M[j] ≠ M[Opt-Traceback(j)])
        print j
        Find-Solution(Opt-Traceback(j))
    }

- # of recursive calls ≤ n ⇒ O(n).
- This method often much cleaner.
Dynamic Programming Template

We use dynamic programming when:
- The overall solution can be easily (and quickly) computed from subproblems.
- There are a polynomial number of subproblems.
- There is a natural ordering (e.g. “smallest” to “largest”) of subproblems.

Key Design Problem: “chicken-and-egg” issue
- finding useful subproblems
- finding a recurrence that connects them

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 $\ldots$.

Q. What's a reasonable balance of accuracy and parsimony?
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:
    - 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 = $e(i, j) + c + OPT(i-1)$.

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

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

**Segmented-Least-Squares()**

```plaintext
M[0] = 0
for j = 1 to n
    M[j] = \min_{i \leq i \leq j} (e(i,j) + 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.
6.5 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: GUCGUUGAGCGAUGUAAACGGUCCUACGGCGAGA*

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.

*approximate by number of base pairs*

**Goal.** Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure $S$ that maximizes the number of base pairs.

---

**RNA Secondary Structure: Examples**

**Examples.**

```
Example 1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
</tr>
<tr>
<td>A</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>A</td>
</tr>
</tbody>
</table>

(base pair)
```

```
Example 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
</tr>
<tr>
<td>A</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>A</td>
</tr>
</tbody>
</table>

```

```
Example 3

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
</tr>
<tr>
<td>A</td>
<td>G</td>
</tr>
<tr>
<td>U</td>
<td>G</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
</tr>
<tr>
<td>A</td>
<td>G</td>
</tr>
<tr>
<td>U</td>
<td>G</td>
</tr>
</tbody>
</table>
```

*ok*

```
Example 4

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

```

*sharp turn*

```
Example 5

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

```

*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: $b_{t+1}b_{t+2}...b_{n-1}$.

Dynamic Programming Over Intervals

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

  take max over $t$ such that $i \leq t < j-4$ and $b_t$ and $b_j$ are Watson-Crick complements
**Bottom Up Dynamic Programming Over Intervals**

**Q.** What order to solve the sub-problems?

**A.** Do shortest intervals first.

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

---

**Bottom Up Dynamic Programming Over Intervals**

**Q.** What order to solve the sub-problems?

**A.** Do shortest intervals first.

```java
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]
}
```
Bottom Up Dynamic Programming Over Intervals

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

```latex
RNA(b_1,...,b_n) \{ 
  \text{for } k = 5, 6, \ldots, n-1 
  \text{for } i = 1, 2, \ldots, n-k 
  \quad j = i + k 
  \quad M[i, j] = \max\{ \text{OPT}(i, j-1), 
                    1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \} \} 
\}
```

Running time. $O(n^3)$.

RNA Secondary Structure: Finding a Solution

A. Must still do some post-processing.

```latex
\text{Run Iterative-Compute-Opt2(n)} 
\text{Run Find-Solution(n)}

\text{Find-Solution(j) \{ 
  \text{if } (j = 0) 
  \quad \text{output nothing} 
  \text{else if } (M[j] \neq M[\text{Opt-Traceback(j)}]) 
  \quad \text{print } j 
  \quad \text{Find-Solution(\text{Opt-Traceback(j)})} 
\}}
```

- # of recursive calls $\leq n \Rightarrow O(n)$.
- This method often much cleaner.
More Complex RNA Structure

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. \( OPT(i) = \) 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 $1, \ldots, i$ 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.

**Input**: $n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$

**for** $w = 0$ to $W$
  **M**[0, $w$] = 0

**for** $i = 1$ to $n$
  **for** $w = 1$ to $W$
    **if** ($w_i > w$)
      **M**[$i$, $w$] = **M**[$i-1$, $w$]
    **else**
      **M**[$i$, $w$] = max ($M[i-1, w], v_i + M[i-1, w-w_i]$)

**return** **M**[$n$, $W$]
Knapsack Algorithm

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.
- Dynamic programming over intervals: RNA secondary structure.
- Adding a new variable: knapsack.

**Top-down vs. bottom-up:** different people have different intuitions.