Midterm 2!

- Monday, November 21
- In class
- Covers Greedy Algorithms
- Closed book

Dynamic Programming “Recipe”

- Solve subproblems, remember the solutions in an array
- Gradually build up solution to bigger problems based on subproblem solutions
Weighted Interval Scheduling

- 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.

Optimal Substructure

- $\text{OPT}(j) =$ value of optimal solution to the problem consisting of job requests 1, 2, ..., $j$.
- Case 1: $\text{OPT}$ selects job $j$.
- Case 2: $\text{OPT}$ does not select job $j$.

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

Straightforward Recursive 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 $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$
}
Iterative Solution

Bottom-up dynamic programming. Unwind recursion.

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

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

Iterative-Compute-Opt {
  \( M[0] = 0 \)
  for \( j = 1 \) to \( n \)
  \( M[j] = \max(v_j + M[p(j)], M[j-1]) \)
}

Dynamic Programming Formula

- Divide a problem into a polynomial number of smaller subproblems
- We often think recursively to identify the subproblems
- Solve subproblem, recording its answer
- Build up answer to bigger problem by using stored answers of smaller problems
- We develop algorithm that builds up the answer iteratively

Iterative Solution

Bottom-up dynamic programming. Unwind recursion.

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

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

Iterative-Compute-Opt {
  \( M[0] = 0 \)
  for \( j = 1 \) to \( n \)
  \( M[j] = \max(v_j + M[p(j)], M[j-1]) \)
}
Iterative Solution
Bottom-up dynamic programming. Unwind recursion.

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 \( n \)
    \[ M[j] = \max(v_j + M[p(j)], M[j-1]) \]
}

Traveling Salesman (sic) Problem

http://imgs.xkcd.com/comics/travelling_salesman_problem.png
Least Squares
- Foundational problem in statistics 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:
  \[ \text{SSE} = \sum_{i=1}^{n} (y_i - ax_i - b)^2 \]

Least Squares Solution
- Result from calculus, least squares achieved when:
  \[
  a = \frac{\sum x_i y_i - \sum x_i \sum y_i}{\sum x_i^2 - \left(\sum x_i\right)^2}, \quad b = \frac{\sum y_i - a \sum x_i}{n}
  \]

Least Squares
- Sometimes a single line does not work very well.
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 fits well.

Segmented Least Squares

- What are the subproblems?
- What are the cases?
- What is the solution for each case?
- How do you find the optimal solution from the cases?

Segmented least squares for points $i..j$
Segmented Least Squares

What are the subproblems?

Segmented least squares for points i...j

What are the cases?

Different values for the starting point of the line segment that ends at point j

What is the solution for each case?

\[ e(i, j) + c + \text{OPT}(i-1) \]

How do you find the optimal solution from the cases?

\[ \min_{i} \text{ over all values of } i \text{ from } 1 \text{ to } j-1 \]
Calculating Line Segment Error

First calculate slope and y-intercept for least squares line:

\[
x = \frac{\sum x_i y_i - (\sum x_i)(\sum y_i)}{\sum x_i^2 - (\sum x_i)^2}
\]

Then calculate the error associated with that line:

\[
SSE = \sum (y_i - ax_i - b)^2
\]

The Price is Right!
(or shopping with somebody else's money)

Spend as much money as possible without going over $100. You can buy at most 1 of each.

<table>
<thead>
<tr>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>$11</td>
</tr>
<tr>
<td>Jeans</td>
<td>$33</td>
</tr>
<tr>
<td>DVD</td>
<td>$20</td>
</tr>
<tr>
<td>Dinner</td>
<td>$15</td>
</tr>
<tr>
<td>Book</td>
<td>$13</td>
</tr>
<tr>
<td>Ice cream</td>
<td>$5</td>
</tr>
<tr>
<td>Shoes</td>
<td>$70</td>
</tr>
<tr>
<td>Pizza</td>
<td>$7</td>
</tr>
</tbody>
</table>

Subset sum

What are the subproblems?

What are the cases?

What is the solution for each case?

How do you find the optimal solution from the cases?
Subset sum

- What are the subproblems?
  - The most we can spend considering items 1..j

- What are the cases?
  - Case 1: Do not buy item j
  - Case 2: Buy item j

- What is the solution for each case?
  - Case 1: OPT(j-1)
  - Case 2: ???

- How do you find the optimal solution from the cases?
Subset sum

- What are the subproblems?
- The most we can spend considering items 1..j

- What are the cases?
  - Case 1: Do not buy item j
  - Case 2: Buy item j

- What is the solution for each case?
  - Case 1: OPT(j-1)
  - Case 2: ???

- How do you find the optimal solution from the cases?
  max(...)

Dynamic Programming Formula

- Divide a problem into a polynomial number of smaller subproblems
  - We often think recursively to identify the subproblems

- Solve subproblem, recording its answer

- Build up answer to bigger problem by using stored answers of smaller problems
  - We develop algorithm that builds up the answer iteratively

Knapsack Problem

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