Knapsack Problem

- Items have weights and values
- The problem is to maximize total value subject to a bound on weight
- Items \( \{I_1, I_2, \ldots, I_n\} \)
  - Weights \( \{w_1, w_2, \ldots, w_n\} \)
  - Values \( \{v_1, v_2, \ldots, v_n\} \)
  - Bound \( K \)
- Find set \( S \) of indices to:
  - Maximize \( \sum_{i \in S} v_i \) such that \( \sum_{i \in S} w_i \leq K \)

Midterm Problem

- \( w_i = 1 \) or \( w_i = 2 \)
- Idea one:
  - sort items by \( v_i/w_i \)
  - greedy packing

  \[
  \begin{array}{cccc}
  7 & 12 & 11 & 1 \\
  \end{array}
  \]
  \( K = 4 \)

- Idea two:
  - pair up items of weight 1
  - greedy packing

  \[
  \begin{array}{ccccc}
  7 & 12 & 6 & 6 & 11 & 3 \\
  \end{array}
  \]
  \( K = 6 \)

Subset Sum Problem

- Let \( w_1, \ldots, w_n = \{6, 8, 9, 11, 13, 16, 18, 24\} \)
- Find a subset that has as large a sum as possible, without exceeding 50

Adding a variable for Weight

- \( \text{Opt}[j, K] \) the largest subset of \( \{w_1, \ldots, w_j\} \) that sums to at most \( K \)
- \( \{2, 4, 7, 10\} \)
  - \( \text{Opt}[2, 7] = \)
  - \( \text{Opt}[3, 7] = \)
  - \( \text{Opt}[3,12] = \)
  - \( \text{Opt}[4,12] = \)
Subset Sum Recurrence

- $\text{Opt}[j, K]$ the largest subset of $\{w_1, \ldots, w_j\}$ that sums to at most $K$

Subset Sum Grid

$\text{Opt}[j, k] = \max(\text{Opt}[j-1, k], \text{Opt}[j-1, k-w_j] + w_j)$

Knapsack Problem

- Items have weights and values
- The problem is to maximize total value subject to a bound on weight
- Items $\{I_1, I_2, \ldots, I_n\}$
  - Weights $\{w_1, w_2, \ldots, w_n\}$
  - Values $\{v_1, v_2, \ldots, v_n\}$
  - Bound $K$
- Find set $S$ of indices to:
  - Maximize $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq K$

Knapsack Recurrence

Subset Sum Recurrence:
$\text{Opt}[j, K] = \max(\text{Opt}[j-1, K], \text{Opt}[j-1, K-w_j] + w_j)$

Knapsack Recurrence:
$\text{Opt}[j, K] = \max(\text{Opt}[j-1, K], \text{Opt}[j-1, K-w_j] + v_j)$

Knapsack Grid

Weights $\{2, 4, 7, 10\}$ Values: $\{3, 5, 9, 16\}$
Dynamic Programming Examples

- Examples
  - Optimal Billboard Placement
    - Text, Solved Exercise, Pg 307
  - Linebreaking with hyphenation
    - Compare with HW problem 6, Pg 317
  - String approximation
    - Text, Solved Exercise, Page 309

Billboard Placement

- Maximize income in placing billboards
  - $b_i = (p_i, v_i)$; value of placing billboard at position $p_i$
- Constraint:
  - At most one billboard every five miles
- Example
  - $\{(6,5), (8,6), (12, 5), (14, 1)\}$

Design a Dynamic Programming Algorithm for Billboard Placement

- Compute $\text{Opt}[1], \text{Opt}[2], \ldots, \text{Opt}[n]$
- What is $\text{Opt}[k]$?

Input $b_1, \ldots, b_n$ where $b_i = (p_i, v_i)$, position and value of billboard $i$

$\text{Opt}[k] = \text{fun}(\text{Opt}[0], \ldots, \text{Opt}[k-1])$

- How is the solution determined from sub problems?

Input $b_1, \ldots, b_n$ where $b_i = (p_i, v_i)$, position and value of billboard $i$

Solution

\[
\begin{align*}
    j &= 0; & \text{if } j \text{ is five miles behind the current position} \\
    \text{for } k &= 1 \text{ to } n \\
    & \text{if the last valid location for a billboard, if one placed at } P[k] \\
    & \text{while } |P[j]| < |P[k] - 5| \\
    & j = j + 1; \\
    & j = j - 1; \\
    & \text{Opt}[k] = \text{Max}(\text{Opt}[k-1], v[k] + \text{Opt}[j]);
\end{align*}
\]

Optimal line breaking and hyphenation

- Problem: break lines and insert hyphens to make lines as balanced as possible
- Typographical considerations:
  - Avoid excessive white space
  - Limit number of hyphens
  - Avoid widows and orphans
  - Etc.
Penalty Function

- Pen(i, j) – penalty of starting a line at position i, and ending at position j

Optimal line breaking and hyphenation is computed with dynamic programming.

- Key technical idea
  - Number the breaks between words/syllables

String approximation

- Given a string S, and a library of strings B = \{b_1, …, b_m\}, construct an approximation of the string S by using copies of strings in B.

  B = \{abab, bbbbaaa, ccbb, ccaacc\}

  S = abacbabbaabbcccbccaabab

Formal Model

- Strings from B assigned to non-overlapping positions of S
- Strings from B may be used multiple times
- Cost of δ for unmatched character in S
- Cost of γ for mismatched character in S
  - MisMatch(i, j) – number of mismatched characters of b_j when aligned starting with position i in S.

Design a Dynamic Programming Algorithm for String Approximation

- Compute Opt[1], Opt[2], …, Opt[n]
- What is Opt[k]?

Solution

```plaintext
for i := 1 to n
    Opt[i] = Opt[i-1] + δ;
for j := 1 to |B|
    p = i - len(b_j);
    Opt[i] = min(Opt[i], Opt[p-1] + γ MisMatch(p, j));
```

Target string S = s_1 s_2 … s_n
Library of strings B = \{b_1, …, b_m\}
MisMatch(i, j) = number of mismatched characters with b_j when aligned starting at position i of S.