CSE 417 Algorithms

Lecture 20, Winter 2023
Dynamic Programming
Subset Sum etc.
Announcements

• Homework 8: Available now
• Dynamic Programming Reading:
  – 6.1-6.2, Weighted Interval Scheduling
  – Path Counting, Paragraphing
  – 6.4 Knapsack and Subset Sum
  – 6.6 String Alignment
    • 6.7* String Alignment in linear space
  – 6.8 Shortest Paths (again)
  – 6.9 Negative cost cycles
    • How to make an infinite amount of money
What is the largest sum you can make of the following integers that is \( \leq 20 \)

\[ \{4, 5, 8, 10, 13, 14, 17, 18, 21, 23, 28, 31, 37\} \]
What is the largest sum you can make of the following integers that is ≤ 2000

\{78, 101, 122, 133, 137, 158, 189, 201, 220, 222, 267, 271, 281, 289, 296, 297, 301, 311, 315, 321, 322, 341, 349, 353, 361, 385, 396 \}
Subset Sum Problem

- Given integers \( \{w_1, \ldots, w_n\} \) and an integer \( K \)
- Find a subset that is as large as possible that does not exceed \( K \)

- Dynamic Programming: Express as an optimization over sub-problems.

- New idea: Represent at a sub problems depending on \( K \) and \( n \)
  - Two dimensional grid
Subset Sum Optimization

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

{2, 4, 7, 10}
Subset Sum Grid

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

\[
\begin{array}{cccccccccccccccc}
4 & 0 & 2 & 2 & 4 & 4 & 6 & 7 & 7 & 9 & 10 & 11 & 12 & 13 & 14 & 14 & 16 & 17 \\
3 & 0 & 2 & 2 & 4 & 4 & 6 & 7 & 7 & 9 & 9 & 11 & 11 & 13 & 13 & 13 & 13 \\
2 & 0 & 2 & 2 & 4 & 4 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
1 & 0 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\{2, 4, 7, 10\}
Subset Sum Code

for $j = 1$ to $n$
    for $k = 1$ to $W$
        $\text{Opt}[j, k] = \max(\text{Opt}[j-1, k], \text{Opt}[j-1, k-w_j] + w_j)$
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:
Knapsack Grid

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

Weights \{2, 4, 7, 10\}  Values: \{3, 5, 9, 16\}
Knapsack Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + v_j)

Weights \{2, 4, 7, 10\}  Values: \{3, 5, 9, 16\}
Alternate approach for Subset Sum

• Alternate formulation of Subset Sum dynamic programming algorithm
  – \( \text{Sum}[i, K] = \text{true} \) if there is a subset of \( \{w_1, \ldots, w_i\} \) that sums to exactly \( K \), false otherwise
  – \( \text{Sum} [i, K] = \text{Sum} [i - 1, K] \text{ OR} \text{Sum}[i - 1, K - w_i] \)
  – \( \text{Sum} [0, 0] = \text{true}; \text{Sum}[i, 0] = \text{false} \) for \( i \neq 0 \)

• To allow for negative numbers, we need to fill in the array between \( K_{\text{min}} \) and \( K_{\text{max}} \)
Run time for Subset Sum

- With $n$ items and target sum $K$, the run time is $O(nK)$
- If $K$ is $1,000,000,000,000,000,000,000,000,000$, this is very slow
- Alternate brute force algorithm: examine all subsets: $O(n2^n)$
- Point of confusion: Subset sum is NP Complete
Two dimensional dynamic programming

Subset sum and knapsack

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

\( \text{Opt}[j, K] = \max(\text{Opt}[j-1, K], \text{Opt}[j-1, K-w_j] + v_j) \)
Reducing dimensions

- Computing values in the array only requires the previous row
  - Easy to reduce this to just tracking two rows
  - And sometimes can be implemented in a single row
- Space savings is significant in practice
- Reconstructing values is harder
Longest Common Subsequence

- C=c₁…c₉ is a subsequence of A=a₁…aₘ if C can be obtained by removing elements from A (but retaining order)
- LCS(A, B): A maximum length sequence that is a subsequence of both A and B

ocurrancem
occurrence

attacggcct

tacgacca
Determine the LCS of the following strings

BARTHOLEMEWSIMPSON

KRUSTYTHECLOWN
String Alignment Problem

- Align sequences with gaps

\[
\begin{align*}
&\text{CAT TGA AT} \\
&\text{CAGAT AGGA}
\end{align*}
\]

- Charge \( \delta_x \) if character \( x \) is unmatched
- Charge \( \gamma_{xy} \) if character \( x \) is matched to character \( y \)

Note: the problem is often expressed as a minimization problem, with \( \gamma_{xx} = 0 \) and \( \delta_x > 0 \)
LCS Optimization

- $A = a_1a_2...a_m$
- $B = b_1b_2...b_n$

- $\text{Opt}[j, k]$ is the length of $\text{LCS}(a_1a_2...a_j, b_1b_2...b_k)$
Optimization recurrence

If \( a_j = b_k \), \( \text{Opt}[ j,k ] = 1 + \text{Opt}[ j-1, k-1 ] \)

If \( a_j \neq b_k \), \( \text{Opt}[ j,k] = \max(\text{Opt}[ j-1,k], \text{Opt}[ j,k-1]) \)