CSE 421, Introduction to Algorithms
Lecture 14, Winter 2024
Dynamic Programming
Subset Sum, Longest Common Subsequence

Announcements
- Dynamic Programming Reading:
  - 6.1-6.2, Weighted Interval Scheduling
  - 6.3 Segmented Least Squares
  - 6.4 Knapsack and Subset Sum
  - 6.6 String Alignment
  - 6.8 Shortest Paths (Bellman-Ford)
- Midterm, Friday, Feb 9
  - Material through 6.3 and HW 5
  - Feb 8 Section will be Midterm review

What is the largest sum you can make of the following integers that is ≤ 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
\[
\text{Opt}[j, K] \text{ the largest subset of } \{w_1, \ldots, w_j\} \text{ that sums to at most } K
\]
\[
\text{Opt}[j, K] = \max(\text{Opt}[j−1, K], \text{Opt}[j−1, K−w_j] + w_j)
\]
Subset Sum Grid

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

\[
\begin{array}{cccccccccccccccc}
4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\{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:

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

Knapsack Grid

\[
\begin{array}{cccccccccccccccc}
4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

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


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

Alternate approach for Subset Sum

• Alternate formulation of Subset Sum dynamic programming algorithm
  – Sum[i, K] = true if there is a subset of {wij,…,wik} that
    sums to exactly K,  false otherwise
  – Sum [i, K] = Sum [i-1, K] OR Sum[i-1, K - wij]
  – Sum [0, 0] = true; Sum[i, 0] = false for i != 0

• To allow for negative numbers, we need to fill in the array between Kmin and Kmax

Run time for Subset Sum

• With n items and target sum K, the run time is O(nK)
• If K is 1,000,000,000,000,000,000,000,000,000,000,000 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

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…cg 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
Determine the LCS of the following strings

BARTHOLEMEWSIMPSON

KRUSTYTHECLOWN

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

Code to compute $\text{Opt}[n, m]$

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        if (A[i] == B[j])
            Opt[i,j] = 1 + Opt[i-1,j-1];
            Best[i,j] := Diag;
        else if (Opt[i-1,j] >= Opt[i,j-1])
            Opt[i,j] := Opt[i-1,j], Best[i,j] := Left;
        else
            Opt[i,j] := Opt[i,j-1], Best[i,j] := Down;
```

Storing the path information

```
for i := 1 to m     Opt[i,0] := 0;
    for j := 1 to n     Opt[0,j] := 0;
    Opt[0,0] := 0;
    for i := 1 to m
        for j := 1 to n
            if A[i] = B[j]  
                Opt[i,j] := 1 + Opt[i-1,j-1];
                Best[i,j] := Diag;
            else if Opt[i-1,j] >= Opt[i,j-1]
                Opt[i,j] := Opt[i-1,j], Best[i,j] := Left;
            else
                Opt[i,j] := Opt[i,j-1], Best[i,j] := Down;
```

Reconstructing Path from Distances
How good is this algorithm?

• Is it feasible to compute the LCS of two strings of length 300,000 on a standard desktop PC? Why or why not.

Implementation 1

```java
public int ComputeLCS() {
    int n = str1.Length;
    int m = str2.Length;
    int[,] opt = new int[n + 1, m + 1];
    for (int i = 0; i <= n; i++)
        opt[i, 0] = 0;
    for (int j = 0; j <= m; j++)
        opt[0, j] = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (str1[i - 1] == str2[j - 1])
                opt[i, j] = opt[i - 1, j - 1] + 1;
            else
                if (opt[i - 1, j] >= opt[i, j - 1])
                    opt[i, j] = opt[i - 1, j];
                else
                    opt[i, j] = opt[i, j - 1];
    return opt[n, m];
}
```

N = 170000

Runtime should be about 5 seconds*

* Personal PC, 10 years old

Observations about the Algorithm

• The computation can be done in O(m+n) space if we only need one column of the Opt values or Best Values

• The computation requires O(nm) space if we store all of the string information