CSE 421, Introduction to Algorithms

Lecture 14, Winter 2024
Dynamic Programming
Subset Sum, Lognest 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

$\text{Opt}[j, K]$ the largest subset of \{w_1, \ldots, w_j\} 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

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

\[
\{2, 4, 7, 10\}
\]
Subset Sum Grid

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>7</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>14</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\{2, 4, 7, 10\}
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

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

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

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

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

• 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

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

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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=c_1…c_g is a subsequence of A=a_1…a_m 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

occuranec  attacggct

occurrence  tacgacca
Determine the LCS of the following strings

BARTHOLEMEWSIMPSON

KRUSTYTHECLOWN
LCS Optimization

• \( A = a_1a_2\ldots a_m \)
• \( B = b_1b_2\ldots b_n \)

• \( \text{Opt}[j, k] \) is the length of 
  \( \text{LCS}(a_1a_2\ldots a_j, b_1b_2\ldots b_k) \)
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 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 ] = 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 ];
Storing the path information

\[ A[1..m], B[1..n] \]

for \( i := 1 \) to \( m \) \quad Opt[i, 0] := 0;
for \( j := 1 \) to \( n \) \quad Opt[0,j] := 0;
Opt[0,0] := 0;
for \( i := 1 \) to \( m \)
  for \( j := 1 \) to \( n \)
    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.
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 = 17000

Runtime should be about 5 seconds*

* Personal PC, 10 years old

---

Manufacturer: Dell
Model: Optiplex 990
Processor: Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz 3.10 GHz
Installed memory (RAM): 8.00 GB (7.88 GB usable)
System type: 64-bit Operating System, x64-based processor
public int SpaceEfficientLCS() {
    int n = str1.Length;
    int m = str2.Length;
    int[] prevRow = new int[m + 1];
    int[] currRow = new int[m + 1];

    for (int j = 0; j <= m; j++)
        prevRow[j] = 0;

    for (int i = 1; i <= n; i++) {
        currRow[0] = 0;
        for (int j = 1; j <= m; j++) {
            if (str1[i - 1] == str2[j - 1])
                currRow[j] = prevRow[j - 1] + 1;
            else if (prevRow[j] >= currRow[j - 1])
                currRow[j] = prevRow[j];
            else
                currRow[j] = currRow[j - 1];
        }
        for (int j = 1; j <= m; j++)
            prevRow[j] = currRow[j];
    }

    return currRow[m];
}
N = 300000

N: 10000 Base 2 Length: 8096   Gamma: 0.8096   Runtime:00:00:01.86
N: 20000 Base 2 Length: 16231  Gamma: 0.81155  Runtime:00:00:07.45
N: 30000 Base 2 Length: 24317  Gamma: 0.8105667 Runtime:00:00:16.82
N: 40000 Base 2 Length: 32510  Gamma: 0.81275   Runtime:00:00:29.84
N: 50000 Base 2 Length: 40563  Gamma: 0.81126   Runtime:00:00:46.78
N: 60000 Base 2 Length: 48700  Gamma: 0.8116667 Runtime:00:01:08.06
N: 70000 Base 2 Length: 56824  Gamma: 0.8117715 Runtime:00:01:33.36

N: 300000 Base 2 Length: 243605 Gamma: 0.8120167
Runtime:00:28:07.32
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