Lecture 27:
• Quiz 6
• Knapsack Problem
• Sequence Alignment

Announcements
• PS-7 out
• HW-7 due Fri

0-1 Knapsack Problem

“Classic” description:
• a thief robbing a store finds \( n \) items
• the \( i \)-th item is worth \( v_i \) dollars and weighs \( w_i \) pounds
• the thief can carry at most \( W \) pounds in his knapsack
• which items should the thief take to get as valuable a load as possible?

Check In: What items should be taken if the knapsack can hold 60 pounds?

<table>
<thead>
<tr>
<th>item</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>30</td>
</tr>
</tbody>
</table>
0-1 Knapsack Problem

The problem:

Given items 1, 2, . . . , n with

• positive integer values $v_1, v_2, \ldots, v_n$
• and positive integer weights $w_1, w_2, \ldots, w_n$

Find a subset $S \subseteq \{1, 2, \ldots, n\}$ of items with

• the max possible sum $\sum_{i \in S} v_i$
• such that $\sum_{i \in S} w_i$ is at most $W$

Other Variants

• Fractional Knapsack Problem: can use fractions of an item
• Bounded Knapsack Problem: each item can be in knapsack 0–c times
• Unbounded Knapsack Problem: each item can be in knapsack $\geq 0$ times

Optimal Substructure

Assume $S$ is an optimal solution with value $V$ and max weight $W$ ...

• then $S$ either contains the last item $n$ or it doesn’t

If $n \not\in S$ then

• $S$ is an optimal solution for the first $n - 1$ items

If $n \in S$ then

• $S - \{n\}$ is optimal for first $n - 1$ items and knapsack capacity $W - w_n$
• where the total value of $S - \{n\}$ is $V - v_n$
0-1 Knapsack Problem

Recurrence Relation:

\[ V_{i,w} = \begin{cases} V_{i-1,w} & \text{if } w_i > w \\ \max\{V_{i-1,w}, V_{i-1,w-w_i} + v_i\} & \text{if } w_i \leq w \end{cases} \]

- where \( w \) is updated max capacity \( W \) (since it decreases in the second case)
- note \( w_i > w \) is a special case since an item itself may exceed capacity

Dynamic programming solution

- build \( V \) bottom up using the recurrence
- have to iterate over both items and weight capacities \( V_{i,w} \)

Algorithm: 0-1 Knapsack Algorithm

Input: positive item values \( v_1, \ldots, v_n \), weights \( w_1, \ldots, w_n \), and capacity \( W \)

Result: Subset \( S \subseteq \{1, 2, \ldots, n\} \) with max total value \( \sum_{i \in S} w_i \leq W \) of items

1. begin
2. let \( A \) be an \((n + 1) \times (W + 1)\) 2-D array
3. for \( w = 0 \) to \( W \) do // base cases
4. \hspace{1em} \( A[0][w] \leftarrow 0 \)
5. for \( i = 1 \) to \( n \) do // iteratively solve subproblems
6. \hspace{1em} for \( w = 0 \) to \( W \) do
7. \hspace{2em} if \( w_i > w \) then
8. \hspace{3em} \( A[i][w] \leftarrow A[i-1][w] \)
9. \hspace{2em} else
10. \hspace{3em} \( A[i][w] \leftarrow \max\{A[i-1][w], A[i-1][w-w_i] + v_i\} \)
11. return \( A[n][W] \) // return solution to largest subproblem

Note: can determine \( S \) from input and \( A \)
Analysis

• cost: $O(nW)$
• referred to as “psuedo polynomial” since $W$ is an integer ...
• because the algorithm relies on $W$, its length becomes a factor

However, the length of $W$ in binary is exponential

• assume $m$ bits required to represent $W$ ... length of $W$ is $m$
• since $m = \lceil \log_2 W \rceil$, we have that $W = 2^m$

This means the problem is exponential in the size of $W$

• in fact, the 0-1 Knapsack problem is NP-Complete
• because it is “psuedo polynomial”, considered “weakly” NP-Complete

Sequence Alignment (String Matching)

Basic Idea

• given two DNA sequences ... with characters from $A, C, T, G$
• find the “best” alignment ... fewest gaps and mismatches
• useful in many bioinformatics applications (e.g., phylogenetics)
• we’ll look at one algorithm (Needleman-Wunsch) among many

Simple Example: (where "-" denotes a “gap”)

<table>
<thead>
<tr>
<th>Alignment 1:</th>
<th>Alignment 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A G T A C - G</td>
<td>A G - T A C G</td>
</tr>
<tr>
<td>A C - A T A G</td>
<td>A C A T A - G</td>
</tr>
</tbody>
</table>

• each possible alignment has some form of “score”
• NW uses sum of the penalty costs $P$ ... gaps + mismatches
• e.g., if gap costs 1 and mismatch 2, then $P = 6$ and $P = 4$ above
Sequence Alignment (String Matching)

The basic problem:

Given strings $X, Y$ over an alphabet $\Sigma = \{A, C, G, T\}$ with penalties:

- $\alpha_{xy}$ for each mismatched $x, y \in \Sigma$, and
- $\alpha_{\text{gap}} \geq 0$ for each gap

Find an alignment of $X$ and $Y$ with the minimum possible total penalty

Optimal Substructure

Assume $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$

- as before, we’ll assume we have an optimal alignment of $X$ and $Y$
- then define a recurrence and dynamic programming approach

Check In: In an alignment with penalty $P$, what are cases for $x_i$ and $y_j$?

(1) $x_i$ and $y_j$ are matched with penalty $\alpha_{x_i y_j}$ ... 0 if $x_i = y_j$

(2) $x_i$ is matched to a gap ... contributing 1 to $P$

(3) $y_j$ is matched to a gap ... contributing 1 to $P$

Assume:

- $X$ and $Y$ are optimally aligned with penalty $P$
- $X'$ contains the first $i - 1$ characters of $X$ (i.e., character $x_i$ is removed)
- $Y'$ contains the first $j - 1$ characters of $Y$ (i.e., character $y_j$ is removed)

Case 1: if $x_i$ and $y_j$ are matched in the alignment with penalty $\alpha_{x_i y_j}$

- then the alignment is optimal for $X'$ and $Y'$ with $P' = P - \alpha_{x_i y_j}$
Sequence Alignment (String Matching)

Case 1 proof by contradiction:
• assume a better alignment exists for \(X'\) and \(Y'\) with penalty \(P'^* < P'\)
• then \(P'^* + \alpha_{x_i y_j} < P' + \alpha_{x_i y_j}\), which means \(P'^* + \alpha_{x_i y_j} < P\)
• implying \(P\) isn’t optimal (a contradiction)

Case 2: if \(x_i\) is matched with a gap
• then the alignment is optimal for \(X'\) and \(Y\) with \(P' = P - \alpha_{\text{gap}}\)
• same rationale as for Case 1

Case 3: if \(y_j\) is matched with a gap
• then the alignment is optimal for \(X\) and \(Y'\) with \(P' = P - \alpha_{\text{gap}}\)
• same rationale as for Cases 1 and 2

Recurrence Relation

\[
P_{i,j} = \min\{P_{i-1,j-1} + \alpha_{x_i y_j}, P_{i-1,j} + \alpha_{\text{gap}}, P_{i,j-1} + \alpha_{\text{gap}}\}
\]

Base Cases: What’s the penalty if a string is empty (i.e., \(i = 0\) or \(j = 0\))?
• The alignment is filled with gaps … one per non-empty string character
• For example, if \(Y\) is empty, the cost is \(i \times \alpha_{\text{gap}}\)
Algorithm: Needleman-Wunsch (Sequence Alignment) Algorithm

Input: $X = x_1, \ldots, x_m$, $Y = y_1, \ldots, y_n$, and penalties $\alpha_{xy}$ and $\alpha_{\text{gap}}$

Result: Minimum alignment score

1 begin
2 let $A$ be an $(m + 1) \times (n + 1)$ 2-D array
3 // base cases
4 for $i = 0$ to $m$ do
5 $A[i][0] \leftarrow i \times \alpha_{\text{gap}}$
6 for $j = 0$ to $n$ do
7 $A[0][j] \leftarrow j \times \alpha_{\text{gap}}$
8 for $i = 1$ to $m$ do // iteratively solve subproblems
9 for $j = 1$ to $n$ do
10 $A[i][j] = \min\{A[i-1][j-1] + \alpha_{x_i,y_j},$ // case 1
11 $A[i-1][j] + \alpha_{\text{gap}},$ // case 2
12 $A[i][j-1] + \alpha_{\text{gap}}\}$ // case 3
13 return $A[m][n]$ // return solution to largest subproblem

Note: can determine alignment from input and $A$ ... cost: $O(mn)$