Lecture 9:
• Dynamic Programming:
  • Tug of War / Subset Sum / Knapsacks
  • Edit Distance / Alignments

Feb 6, 2018
Tug of War
Subset-Sum
Knapsack
Tug-of-War

- We have $n$ students with weights $w_1, \ldots, w_n \in \mathbb{N}$, need to split as evenly as possible into two teams
  - e.g. $\{21, 42, 33, 52\}$

\[
A_1 = \{21, 42\} \quad 63 \\
A_2 = \{33, 52\} \quad 85 \\
\Delta = 22
\]

\[
T = 148
\]

Goal: Lighter team should have as close to $\frac{T}{2}$ as possible weight over.

\[
T = \sum_{i=1}^{n} w_i; \\
the \ lighter \ team \ has \ wt \ at \ most \ \frac{T}{2}.
\]
Tug-of-War

- **Input:** weights $w_1, \ldots, w_n \in \mathbb{N}$ for $n$ students
  - Define $T = \sum_i w_i$
- **Output:** a subset of students $S$ with weight as large as possible but not more than $T/2$
  - $S \subseteq \{1, \ldots, n\}, W_S = \sum_{i \in S} w_i$
Subset Sum (generalization of ToW)

- **Input**: weights $w_1, \ldots, w_n \in \mathbb{N}$ for $n$ items, and a maximum weight $T > 0$
- **Output**: a subset of students $S$ with weight as large as possible but not more than $T$
  - $S \subseteq \{1, \ldots, n\}$, $W_S = \sum_{i \in S} w_i$
Tug-of-War

$w_1, \ldots, w_n \in \mathbb{N}$ \quad $T > 0$ is the max ut.

(not necessarily sorted)

- Let $O$ be the **optimal** subset
- $O$ either contains $n$ or it doesn't.
- Case I ($n \notin O$): $O$ is the optimal set among $\{w_1, \ldots, w_{n-1}, T\}$
- Case II ($n \in O$): $O$ is the [optimal set among $\{w_1, \ldots, w_{n-1}, T-w_n\}\}$ + $w_n$
- $OPT(i, t)$ the optimal sol'n w/ items $1, \ldots, i$, max ut at $t$.
- $OPT(n, T) = \max \{ OPT(n-1, T), w_n + OPT(n-1, T-w_n)\}$
Ask the Audience

• **Input:** $T = 7$, $w_1 = 1$, $w_2 = 3$, $w_3 = 5$

• **Fill the dynamic programming table**

$$\text{OPT}(i, t) \text{ is the value of OPT for } w_i \text{ and } w_1 \text{ to } w_i \text{ and } w_1 \text{ up to } t.$$  

$$\text{OPT}(n, T) \text{ is max } \{ \text{OPT}(n-1), v_n + \text{OPT}(\text{OPT}(n-1)) \}$$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<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>

- **Weights**

- **Items**

$\text{OPT}(0, t) = 0$

$\text{OPT}(i, 0) = 0$

$\text{OPT}(i, -\infty) = -\infty$

$\text{OPT}(s) = n \cdot 3^2$

$w_1 \text{ up to } w_6.$
Tug-of-War (Bottom-Up)

ToW\((w_1, \ldots, w_n, T)\):

Let \(M[0:n, 0:T]\) be an array to store the solutions
\(M[0, t] \leftarrow 0\) for all \(0 \leq t \leq T\)
For \(i = 1, \ldots, n\):
  For \(t = 0, 1, \ldots, T\):
    if \(w_i \leq t\) \(M[i, t] \leftarrow \max\{M[i-1, t], w_i + M[i-1, t-w_i]\}\)
    if \(w_i > t\) \(M[i, t] \leftarrow M[i-1, t]\)

Return \(M[n, T]\)

After running ToW, need another alg to trace through \(M\)
finding the optimal set.
Knapsack

- **Input:** weights and values \((w_1, v_1), \ldots, (w_n, v_n)\) for \(n\) items, and a maximum weight \(T > 0\)
- **Output:** a subset of students \(S\) with value as large as possible but weight at most \(T\)

\[
S \subseteq \{1, \ldots, n\}, \quad W_S = \sum_{i \in S} w_i, \quad V_S = \sum_{i \in S} v_i \leq T
\]

\(\text{thing you're trying to optimize}\)

\(\text{OPT}(n, T)\) is the value of the opt soln

\[
\text{OPT}(n, T) = \max \left\{ \text{OPT}(n-1, T), \ v_n + \text{OPT}(n-1, T-w_n) \right\}
\]
Ask the Audience

• Let $OPT(i, t)$ be the optimal solution for items $1, \ldots, i$ with weight at most $t$.

• $OPT(i, t) = ???$
Summary

• Can also solve subset sum and knapsack problems in $\Theta(nT)$ time where $T$ is the maximum weight
  • Note, dependence on $T$ is rather undesirable, what if $M = 2^{168}$?

  • Tradeoff betw precision of sol'n and running time
Edit Distance Alignments
Distance Between Strings

- Autocorrect is (usually) pretty good

- *occurance* and *occurrence* seem similar, but only if we define similarity carefully

```
occurance
did you mean: occurrence
```

```
ocurrence
```

```
7 apart
```

```
2 apart
```
Edit Distance / Alignments

• Given two strings $x \in \Sigma^n, y \in \Sigma^m$, the edit distance is the number of insertions, deletions, and swaps required to turn $x$ into $y$.

• Given an alignment, the cost is the number of positions where the two strings don’t agree.

```
\begin{array}{cccc}
& x & o c c u r r a n c e & y \\
\hline
y & o c c u r r e n c e & & \\
\end{array}
```

edit distance is 2.

edit distance $d \iff$ alignment of cost $d$. 

For today $\Sigma' = \{a, \ldots, z\}$ replace $a \leftrightarrow b$. 
Ask the Audience

- What is the minimum cost alignment of the strings moon and money

```

  moon

money

  money

moon

money

cost 3

cost 3
```
Edit Distance / Alignments

- **Input:** Two strings $x \in \Sigma^n$, $y \in \Sigma^m$
- **Output:** The minimum cost alignment of $x$ and $y$

$O(1)$ time to check if $x_i = y_j$
Dynamic Programming

- As always, consider the **optimal** alignment...
- Three choices for the final column
  - Case I: only use $x$ ($x_n, -$)
  - Case II: only use $y$ ($-, y_m$)
  - Case III: use one symbol from each ($x_n, y_m$)
**Recurrence:**

Let \( \text{OPT}(i,j) \) be the edit dist/cost of opt. alignment of the prefixes \( X_1 \ldots X_i \) and \( Y_1 \ldots Y_j \).

\[
\text{OPT}(i,j) = \begin{cases} 
\min \left\{ \text{OPT}(i-1,j) + 1, \text{OPT}(i,j-1) + 1, \text{OPT}(i-1,j-1) \right\} & \text{if } X_i = Y_j \\
| + \text{min} \left\{ \text{OPT}(i-1,j), \text{OPT}(i,j-1), \text{OPT}(i-1,j-1) \right\} & \text{if } X_i \neq Y_j 
\end{cases}
\]

\( \text{OPT}(i,0) = i \) \quad \text{Base cases where one string is empty}

\( \text{OPT}(0,j) = j \)

\( \text{OPT}(0,0) = 0 \)
Dynamic Programming

• As always, consider the **optimal** alignment...

• Three choices for the final column
  
  • Case I: only use $x$ ($x_n, -$)
  
  • Case II: only use $y$ ($-, y_m$)
  
  • Case III: use one symbol from each ($x_n, y_m$)
Example

\[x = \text{peas}\]
\[y = \text{east}\]

\[
\begin{array}{c|ccccc}
\text{-} & \text{e} & \text{a} & \text{s} & \text{t} \\
\hline
\text{-} & 0 & 1 & 2 & 3 & 4 \\
\text{p} & 1 \quad \text{①} & 2 & 3 & 4 \\
\text{e} & 2 & 1 \quad \text{①} & 3 & 4 \\
\text{a} & 3 & 2 & 1 \quad \text{①} & 2 \\
\text{s} & 4 & 3 & 2 & 1 \quad \text{②} \\
\end{array}
\]
Ask the Audience

• Suppose adding a space costs $\delta > 0$ and aligning $a, b$ costs $c_{a,b} > 0$. Write a recurrence for the minimum cost alignment.
Summary

• Can compute the edit distance, or minimum cost alignment between two strings in time $O(nm)$
  • Works for any alphabet, assuming we can decide if two symbols are equal in $O(1)$ time

• There was nothing special about our notion of alignment cost
  • In general: cost $\delta$ for using a space (cases I/II) and costs $c_{a,b}$ for aligning $a, b \in \Sigma$ (case III)
  • Can still solve in $O(nm)$ time

• Uses $O(nm)$ space, more prohibitive in practice