CS3000: Algorithms & Data
Jonathan Ullman

Lecture 7:

- Dynamic Programming: Knapsacks, Edit Distance

Jan 29, 2020

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}

54 94

{21, 52} {42, 33}

73 75


The Knapsack Problem

- **Input:** n items for your knapsack
  - value v_i and a weight $w_i \in \mathbb{N}$ for n items
  - capacity of your knapsack $T \in \mathbb{N}$
- **Output:** the most valuable subset of items that fits in the knapsack
  - Subset $S \subseteq \{1, \dots, n\}$
  - Value $V_S = \sum_{i \in S} v_i$ as large as possible
  - Weight $W_S = \sum_{i \in S} w_i$ at most T

- **SubsetSum:** $v_i = w_i$

Tug-of-War: $v_i = w_i$, $T = \frac{1}{2}\sum_{i=1}^{n} w_i$

Item 1: $w_i = \frac{T}{2} + 1$ $v_i = 3$

Item 2: $w_i = \frac{T}{2}$ $v_i = 2$

Item 3: $w_i = \frac{T}{2}$ $v_i = 2$

(1) Interval Scheduling

- Given n items
- Wanted to find the "best" subset $S \subseteq \{1, \dots, n\}$
- "Is the n^{th} item in the optimal set?"

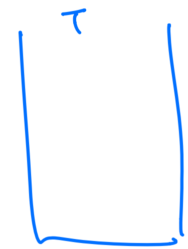
(2) Segmented Least Squares

- Given n points in order

- Find the "best" partition of the n points
- "What is the last segment in the optimal partition?"

Dynamic Programming

- Let $O \subseteq \{1, \ldots, n\}$ be the **optimal** subset of items
- Should item n go in the optimal solution?



- Case 1 ($n \notin O$) "n not in O"
  - Then O is the optimal solution for items $1, \ldots, n-1$ and capacity T
- Case 2 ($n \in O$) "n is in O"
  - Then O is $\{n\}$ + the optimal solution for items $1, \ldots, n-1$ and capacity $T - w_n$

Dynamic Programming

$0 \le i \le n \qquad 0 \le S \le T$

- Let $\mathbf{OPT}(\boldsymbol{i}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \dots, i\}$ in a knapsack of size S
- **Case 1:** $i \notin O_{i,S}$ $\quad OPT(i, S) = OPT(i-1, S)$
- **Case 2:** $i \in O_{i,S}$ $\quad OPT(i, S) = v_i + OPT(i-1, S-w_i)$

$$OPT(i, S) = \begin{cases} \max\{OPT(i-1, S), v_i + OPT(i-1, S-w_i)\} & \text{if } w_i \le S \\ OPT(i-1, S) & \text{if } w_i > S \end{cases}$$

Dynamic Programming

- Let $\mathbf{OPT}(\boldsymbol{i}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \ldots, i\}$ in a knapsack of size S
- **Case 1:** $i \notin O_{i,S}$
  - Use opt. solution for items 1 to $i - 1$ and size S
- **Case 2:** $i \in O_{i,S}$
  - Use i + opt. solution for items 1 to $i - 1$ and size $S - w_j$

Recurrence:

$$\mathrm{OPT}(i, S) = \begin{cases} \max\{\mathrm{OPT}(i-1, S), v_i + \mathrm{OPT}(i-1, S-w_i)\} & \text{if } w_i \leq S \\ \mathrm{OPT}(i-1, S) & \text{if } w_i > S \end{cases}$$

Base Cases:

$$\mathrm{OPT}(i, 0) = \mathrm{OPT}(0, S) = 0$$

Ask the Audience

Recurrence:

$$\mathrm{OPT}(i,S) = \begin{cases} \max\{\mathrm{OPT}(i-1,S), v_i + \mathrm{OPT}(i-1,S-w_i)\} & \text{if } w_i \le S \\ \mathrm{OPT}(i-1,S) & \text{if } w_i > S \end{cases}$$

Base Cases:

$$\mathrm{OPT}(i,0) = \mathrm{OPT}(0,S) = 0$$

- Input: $T = 8, n = 3$
  - $w_1 = 1\,, v_1 = 4$
  - $w_2 = 3\,, v_2 = 5$
  - $w_3 = 5\,, v_3 = 8$

$O_{3,8} = \{3\} + O_{2,3}$

$= \{3\} + \{2\} + O_{1,0}$

$= \{2,3\}$

items									
3	0	4 N	4 N	5 N	9 N	9 N	12 Y	12 Y	13 Y
2	0	4 N	4 N	5 Y	9 Y	9 Y	9 Y	9 Y	9 Y
1	0	4 Y	4 Y	4 Y	4 Y	4 Y	4 Y	4 Y	4 Y
0	0	0	0	0	0	0	0	0	0
-	0	1	2	3	4	5	6	7	8

capacities

Knapsack ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n,T):
  M[0,S] ← 0, M[i,0] ← 0

  for (S = 1,…,T):
    for (i = 1,…,n):
      if (wi > S): M[i,S] ← M[i-1,S]
      else: M[i] ← max{M[i-1,S],vi + M[i-1,S-wi]}

  return M[n,T]
```

Filling the Knapsack

```
// All inputs are global vars
// M[0:n,0:T] contains solutions to subproblems
FindSol(M,n,T):
  if (n = 0 or T = 0): return ∅
  else:
    if (w_n > T): return FindSol(M,n-1,T)
    else:
      if (M[n-1,T] > v_n + M[n-1,T-w_n]):
        return FindSol(M,n-1,T)
      else:
        return {n} + FindSol(M,n-1,T-w_n)
```

Knapsack Wrapup

- Can solve **knapsack** in time/space $O(nT)$
  - Brute force algorithms runs in time $O(2^n)$

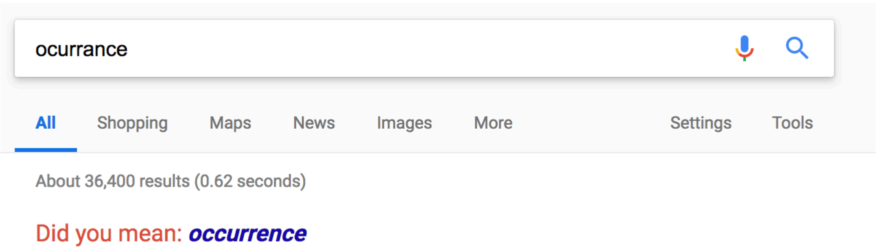
- Dynamic Programming:
  - Decide whether the n^{th} item goes in the knapsack

- Solve **subset-sum** and **tug-of-war** as special cases

Edit Distance
Alignments

Distance Between Strings

- Autocorrect works by finding similar strings



- `ocurrance` and `occurrence` seem similar, but only if we define similarity carefully

```
ocurrance          oc urrance
occurrence         occurrence
```

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.

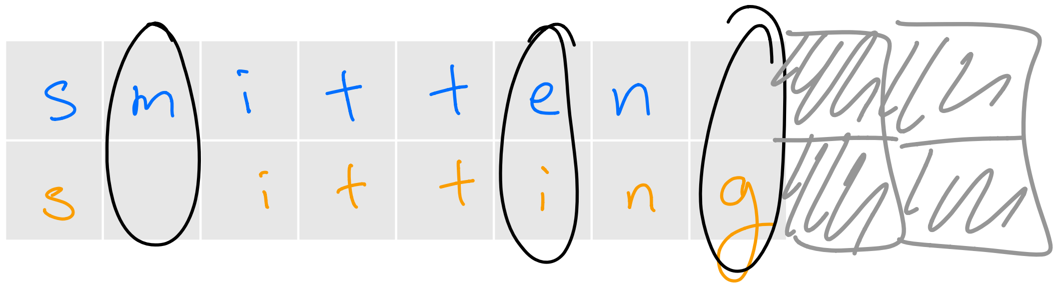
- Given an alignment, the cost is the number of positions where the two strings don't agree

o	c		u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

The edit dist btw x, y is the cost of the min. cost alignment

Ask the Audience

- What is the minimum cost alignment of the strings `smitten` and `sitting`

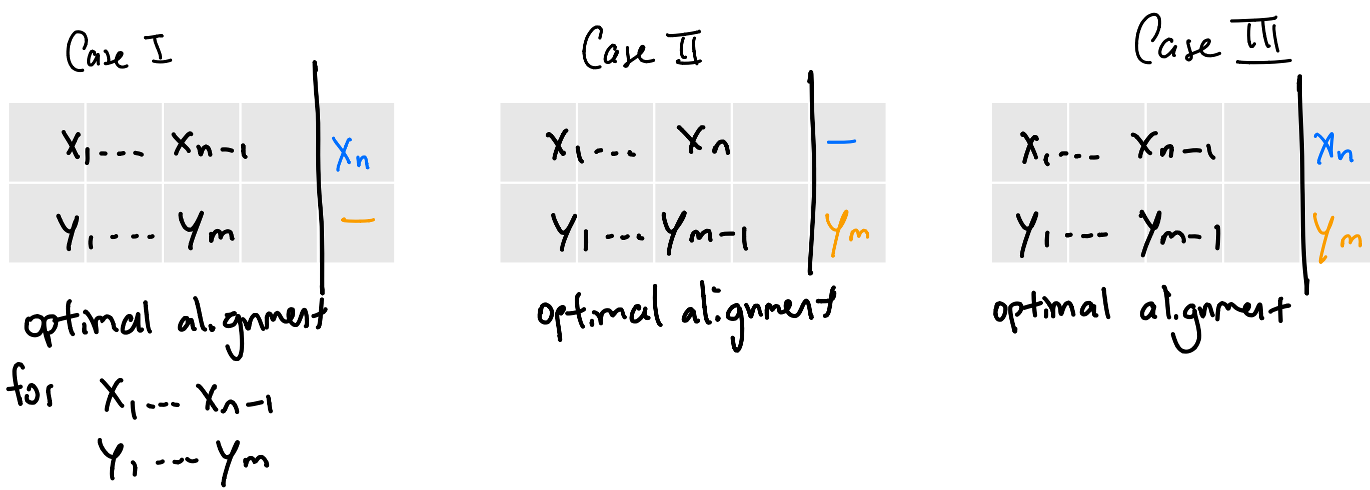


Edit Distance / Alignments

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

Dynamic Programming

- Consider the **optimal** alignment of x, y
- 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)



Dynamic Programming

- Consider the **optimal** alignment of x, y
- **Case I:** only use x ($x_n, -$)
  - deletion + optimal alignment of $x_{1:n-1}, y_{1:m}$
- **Case II:** only use y ($-, y_m$)
  - insertion + optimal alignment of $x_{1:n}, y_{1:m-1}$
- **Case III:** use one symbol from each (x_n, y_m)
  - If $x_n = y_m$: optimal alignment of $x_{1:n-1}, y_{1:m-1}$
  - If $x_n \neq y_m$: mismatch + opt. alignment of $x_{1:n-1}, y_{1:m-1}$

Dynamic Programming

- $\mathbf{OPT}(\boldsymbol{i}, \boldsymbol{j})$ = cost of opt. alignment of $x_{1:i}$ and $y_{1:j}$
- **Case I:** only use x ($x_i, -$)
- **Case II:** only use y ($-, y_j$)
- **Case III:** use one symbol from each (x_i, y_j)

$$OPT(i,j) = \begin{cases} \text{if } x_i = y_j & \min\{OPT(i-1,j-1),\ 1+OPT(i,j-1),\ 1+OPT(i-1,j)\} \\ \text{if } x_i \neq y_j & 1+\min\{OPT(i-1,j-1),\ OPT(i,j-1),\ OPT(i-1,j)\} \end{cases}$$

Dynamic Programming

- $\mathbf{OPT}(\boldsymbol{i}, \boldsymbol{j})$ = cost of opt. alignment of $x_{1:i}$ and $y_{1:j}$
- **Case I:** only use x ($x_i, -$)
- **Case II:** only use y ($-, y_j$)
- **Case III:** use one symbol from each (x_i, y_j)

Recurrence:

$$\mathrm{OPT}(i,j) = \begin{cases} 1 + \min\{\mathrm{OPT}(i-1,j), \mathrm{OPT}(i,j-1), \mathrm{OPT}(i-1,j-1)\} & \text{if } x_i \neq y_j \\ \min\{1 + OPT(i-1,j), 1 + \mathrm{OPT}(i,j-1), \mathrm{OPT}(i-1,j-1)\} & \text{if } x_i = y_j \end{cases}$$

Base Cases:

$\mathrm{OPT}(i,0) = i$, $\mathrm{OPT}(0,j) = j$

Example

x = pert

y = beast

	-	b	e	a	s	t
-						
p			OPT(1,1)	OPT(1,2)		
e			OPT(2,1)	OPT(2,2)		
r						
t						

Finding the Alignment

- $\mathbf{OPT}(\boldsymbol{i}, \boldsymbol{j})$ = cost of opt. alignment of $x_{1:i}$ and $y_{1:j}$
- **Case I:** only use x ($x_i, -$)
- **Case II:** only use y ($-, y_j$)
- **Case III:** use one symbol from each (x_i, y_j)

Knapsack ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n,m):
  M[0,j]←j, M[i,0]←i

  for (i= 1,…,n):
    for (j = 1,…,m):
      if (xi = yj):
        M[i,j] = min{1+M[i-1,j],1+M[i,j-1],M[i-1,j-1]
      elseif (xi != yj):
        M[i,j] = 1+min{M[i-1,j],M[i,j-1],M[i-1,j-1]}

  return M[n,m]
```

Ask the Audience

- Suppose **inserting/deleting costs $\delta > 0$** and **swapping $a \leftrightarrow b$ costs $c_{a,b} > 0$**
- Write a recurrence for the min-cost alignment

Summary

- Compute the **edit distance**, or **min-cost alignment** between two strings in time/space $O(nm)$

- Dynamic Programming:
  - Decide the final pair of symbols in the alignment
- Space can be prohibitive in practice
  - Compute edit distance in space $O(\min\{n, m\})$

  - Can also find alignment in small space!

Saving Space

- Input: Two strings $x \in \Sigma^n, y \in \Sigma^m$
- Output: The **edit distance between** x and y

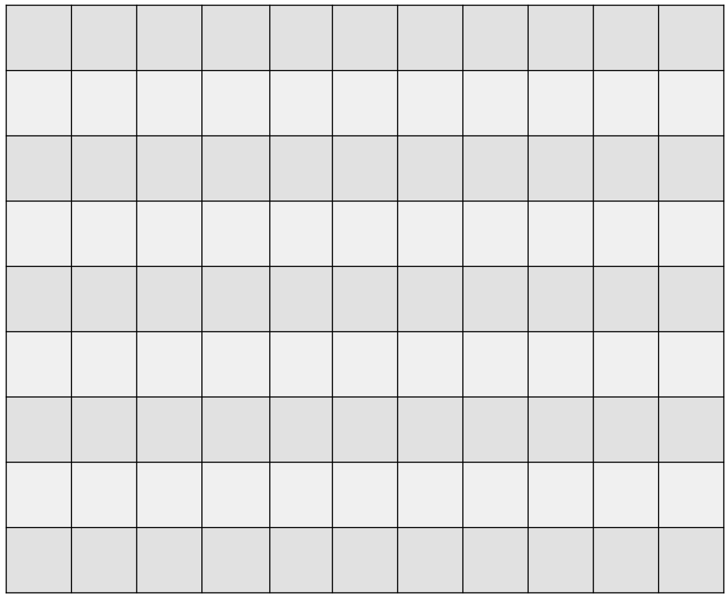
- Can compute $EDIT(x, y)$ with $O(n + m)$ space.

Saving Space

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

- Can we still use $O(n + m)$ space?

Saving Space



Saving Space

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

Summary

- Can compute the edit distance, or minimum cost alignment between two strings in **time** $O(nm)$ and **space** $O(n + m)$