CS4800: Algorithms & Data
Jonathan Ullman

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

$T = 148$

integer weight

Goal: Lighter team should have wt as close to $\frac{T}{2}$ as possible w/o going over.

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

$T = \sum_{i=1}^{n} w_i$

$A_1 = \{21, 42\}$ 63

$A_2 = \{33, 52\}$ 85

$\Delta = 22$

$A_1 = \{42, 33\}$ 75

$A_2 = \{21, 52\}$ 73

$\Delta = 2$

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 $T/2$
  - $S \subseteq \{1, \ldots, n\}$, $\boxed{W_S = \sum_{i \in S} w_i}$


Subset Sum (generalization of ToW)

in ToW $T = \frac{1}{2}\sum_{i=1}^{n} w_i$

- 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

$w_1, \dots, w_n \in \mathbb{N}$ $T > 0$ is the max wt.
(not necessarily sorted)

- Let O be the **optimal** subset (wt as close to T as possible w/o going over)
- O either contains n or it doesn't.
- Case I ($n \notin O$): O is the optimal set among $(w_1, \dots, w_{n-1}, T)$
- Case II ($n \in O$): O is the [optimal set among $(w_1, \dots, w_{n-1}, T - w_n)$] $+ w_n$
- $OPT(i, t)$ the weight of the optimal sol'n w/ items $1, \dots, i$, max wt t.

$$\boxed{OPT(n, T) = \max\{OPT(n-1, T),\ w_n + OPT(n-1, T - w_n)\}}$$

Ask the Audience

OPT(i,t) : value of OPT for $w_1, \ldots, w_i$ and wt t.

$$\text{OPT}(n,T) = \max\{\text{OPT}(n-1,T),\ w_n + \text{OPT}(n-1, T-w_n)\}$$

- Input: $T = 7, w_1 = 1, w_2 = 3, w_3 = 5$
- Fill the dynamic programming table

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

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

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

items

3	0	1	1	3	4	5	6	6
2	0	1	1	3	4	4	4	4
1	0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
-	0	1	2	3	4	5	6	7

weights

OPT sdn $\Rightarrow \{1, 3\}$ w/ wt 6.

Tug-of-War (Bottom-Up)

ToW($w_1, \dots, w_n, T$):

Let $M[0{:}\, n, 0{:}\, T]$ be an array to store the solutions

$M[0, t] \leftarrow 0$ for all $0 \leq t \leq T$ $M[i,0]=0$ $1 \leq i \leq n$

For $i = 1, \dots, n$:

For $t = \cancel{0}, 1, \dots, T$:

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 ToW is $[\ w_i = v_i\ ]$

- Input: **weights** and **values** $(w_1, v_1), \ldots, (w_n, v_n)$ for n items, and a maximum weight $\mathbf{T} > 0$
- Output: a subset of students S with **value** as large as possible but **weight** at most $\mathbf{T}$
  - $S \subseteq \{1, \ldots, n\}$, $\boxed{W_S = \sum_{i \in S} w_i}$, $\boxed{V_S = \sum_{i \in S} v_i}$

$\leq T$

thing you're trying to optimize

OPT(n, T) is the value of the opt soln

$$OPT(n,T) = \max\{\, OPT(n-1, T),\ v_n + OPT(n-1, T-w_n) \,\}$$

Ask the Audience

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

- $OPT(i, t) = ???$

Summary

table has size $(n+1)(T+1)$

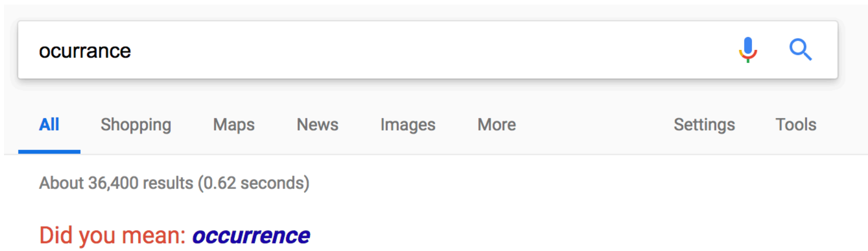
- Can also solve subset sum and knapsack problems in $\Theta(nT)$ time where T is the maximum weight
  - Note, dependence on T is rather undesirable, what if $M = 2^{168}$?

  - Tradeoff btw precision of sol'n and running time

Edit Distance Alignments

Distance Between Strings

- Autocorrect is (usually) pretty good



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

```
ocurrance              oc urrance
occurrence             occurrence
```

7 apart

2 apart

Edit Distance / Alignments

minimum

for today $\Sigma = \{a, \dots, z\}$

- 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.

replace a w/ b.

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

x	o	c		u	r	r	a	n	c	e
y	o	c	c	u	r	r	e	n	c	e

edit distance is 2.

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

Ask the Audience

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

m	o	o	n		
m		o	n	e	y

cost 3

m	o	o	n	
m	o	n	e	y

cost 3

Edit Distance / Alignments

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

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

edit distance

Dynamic Programming

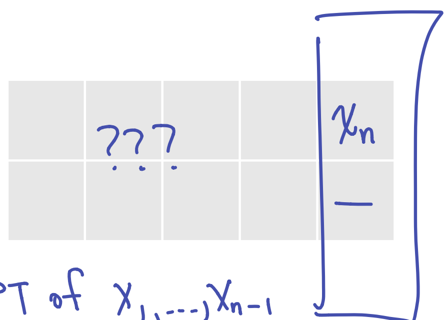
$x_1 \dots x_n$

$y_1 \dots y_m$

- 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)

Do not consider $(-,-)$

case I

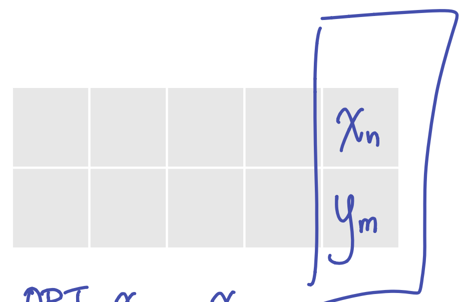


OPT of $x_1, \dots, x_{n-1}$ $y_1, \dots, y_m$ +1 cost

case II

OPT $x_1, \dots, x_n$ $y_1, \dots, y_{m-1}$ +1

case III



OPT $x_1, \dots, x_{n-1}$ $y_1, \dots, y_{m-1}$ +1 if $x_n \neq y_m$ +0 o.w.

Recurrence:

Let OPT(i,j) be the edit dist / cost of opt. alignment of the prefixes $x_1, \ldots, x_i$
$y_1, \ldots, y_j$

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

$$\left.\begin{aligned} OPT(i,0) &= i \\ OPT(0,j) &= j \\ OPT(0,0) &= 0 \end{aligned}\right\}$$ Base cases where at least one string is empty

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 = peas

y = east

p	e	a	s	
	e	a	s	t

	-	e	a	s	t
-	0	1	2	3	4
p	1	1	2	3	4
e	2	1	2	3	4
a	3	2	1	2	3
s	4	3	2	1	2

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 δ 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