Dynamic Programming in 2D

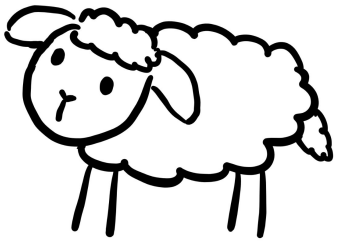
DNA Comparison: Longest Common Subsequence (LCS)

The length of the longest common subsequence (LCS) between two genomes is a measure of similarity.

S1 ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S2 GTCGTTCGGAATGCCGTTGCTCTGTAA

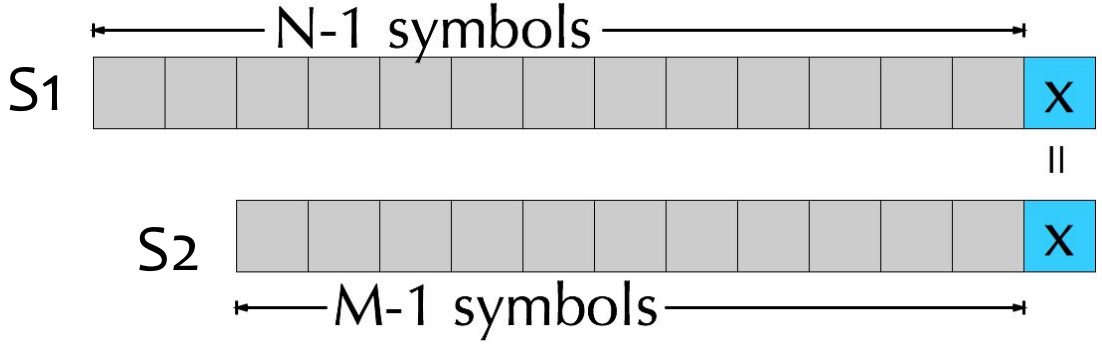
What is an LCS of "Go Blue" and "Ghoul"?



LCS Recurrence

Case 1: Suppose the last character of S1 and S2 are the same i.e. S1[N] = S2[M]

Claim. There exists an optimal solution that matches S1[N] and S2[M].
Proof.

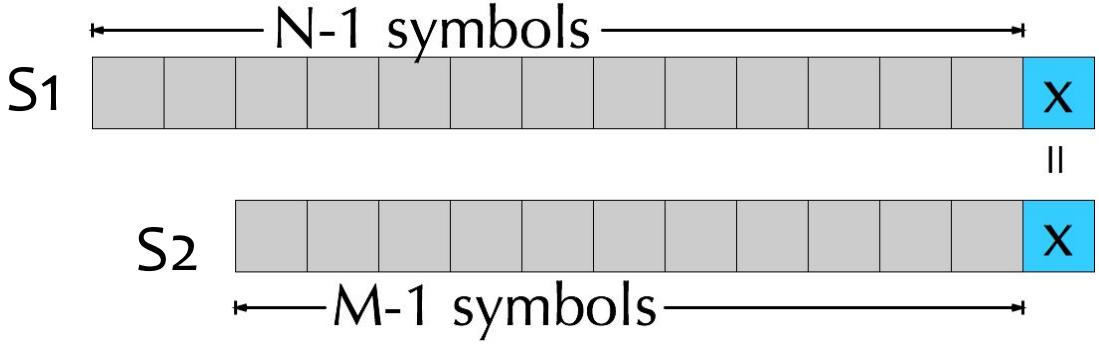


LCS Recurrence

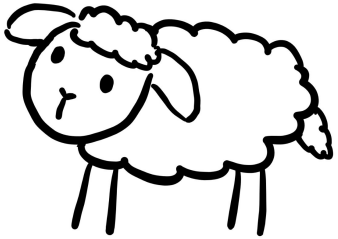
Case 1: Suppose the last character of S1 and S2 are the same
i.e. S1[N] = S2[M]

Claim. There exists an optimal solution that matches S1[N] and S2[M].

LCS(S1[1..N] , S2[1..M]) =



Since there's an optimal solution matching S1[N] and S2[M], we can **safely** add that match to our solution!

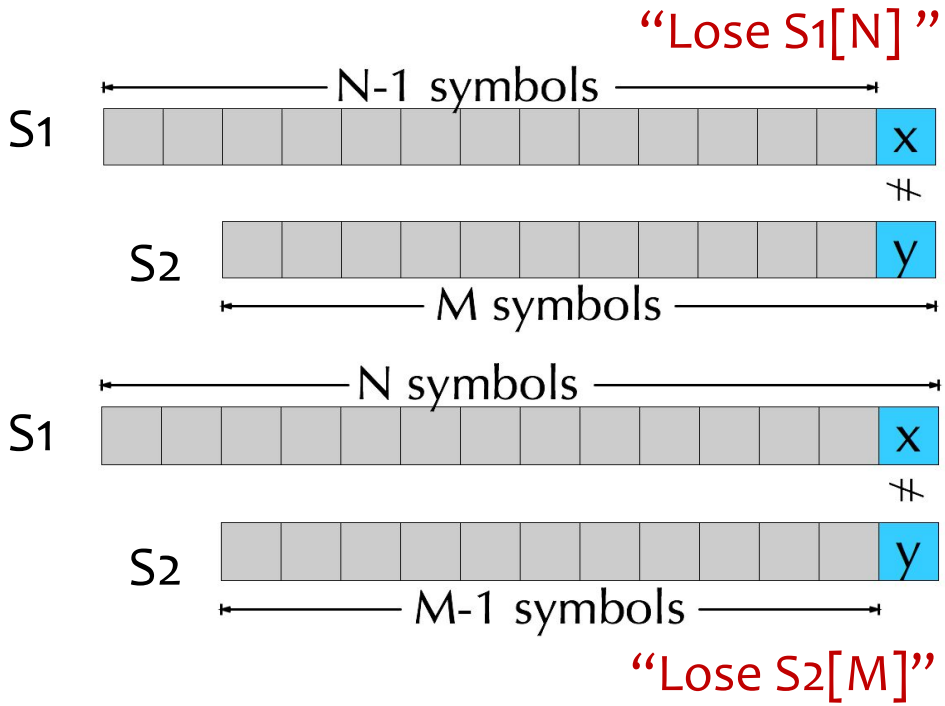


LCS Recurrence

Case 2: The last character of S1 and S2 are **not** the same

OPT doesn't have at least one of S1[N] and S2[M] **("lose it or lose it!")**

LCS(S1[1..N] , S2[1..M]) =



Full Recurrence for LCS

LCS(S1[1..N] , S2[1..M]) =

{

if S1[N] = S2[M]

otherwise

Base cases:

The DP Recipe

1. Write recurrence ← usually the trickiest part
2. Size of table: How many dimensions? Range of each dimension?
3. What are the base cases?
4. To fill in a cell, which other cells do I look at? In which order do I fill the table?
5. Which cell(s) contain the final answer?
6. Running time = (size of table)·(time to fill each entry)
7. To reconstruct the solution (instead of just its size) follow arrows from final answer to base case

Let's Follow the DP Recipe

S_1=GAC; S_2=AGCAT						
	Ø	A	G	C	A	T
Ø						
G						
A						
C						

Pseudocode

```
Algorithm LCS( S1[1..N] , S2[1..M] )
    table := 2D-array indexed from 0 to N, and 0 to M
    for i = 0 to N:
        table(i, 0) = 0
    for i = 0 to M:
        table(0, i) = 0
    for i = 0 to N:
        for j = 0 to M:
            if S1[i] = S2[j]:
                table(i,j) = 1 + table(i-1, j-1)
            else:
                table(i,j) = max{table(i-1, j), table(i, j-1)}
    Return table(N,M)
```

A Visualization of LCS DP

https://www.cs.usfca.edu/~galles/visualization/DPLCS.html

Can we do better than O(MN) time?

Under a hypothesis called the "strong exponential time hypothesis", there is no algorithm even in time $O(MN^{0.99})$!

If the hypothesis is true, you've seen the (essentially) optimal algorithm.

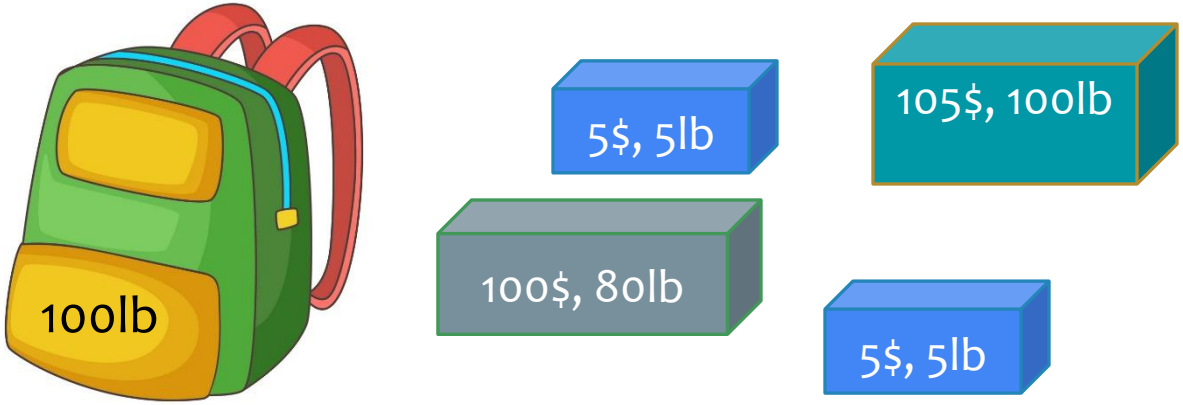
But... |human genome| ≈ 3bil, |chimp genome| ≈ 2.8bil

- **Current research**: faster approximation algorithms

Knapsack Problem

Input: A set of **n** items $t_1,\ldots,t_n$, each with an integer value v_i and weight w_i, along with an integer **W** (representing the total capacity of the knapsack).

Output: A set of items whose total weight is at most **W** and whose total value is maximized.



Question: What is the input size in terms of **n** and **W**?

(Suppose each individual v_i and w_i is small enough that we can disregard its contribution to the input size)

Knapsack Recurrence

Consider item **tn**.

Knapsack({**t1,...,tn**}, **W**) = $\begin{cases} \\ \\ \end{cases}$

(use it!)

(lose it!)

Base cases:

Let's follow the DP Recipe

remaining weight

remaining items	∅	1	2	3
∅				
t1: 4$, 2lb				
t2: 8$, 1lb				
t3: 10$, 2lb				
t4: 5$, 5lb				

Pseudocode

Algorithm Knapsack($\{t_1,\ldots,t_n\}$, W)

table := 2D-array indexed from 0 to n, and 0 to W

for i = 0 to n:

table(i, 0) = 0

for j = 0 to W:

table(0, j) = 0

for i = 0 to n:

for j = 0 to W:

if $w_i \leq j$: useit = v_i + **table**(i-1, j - w_i)

else: useit = $-\infty$

table(i,j) = **max**{ useit, **table**(i-1, j) }

Return table(n,W)