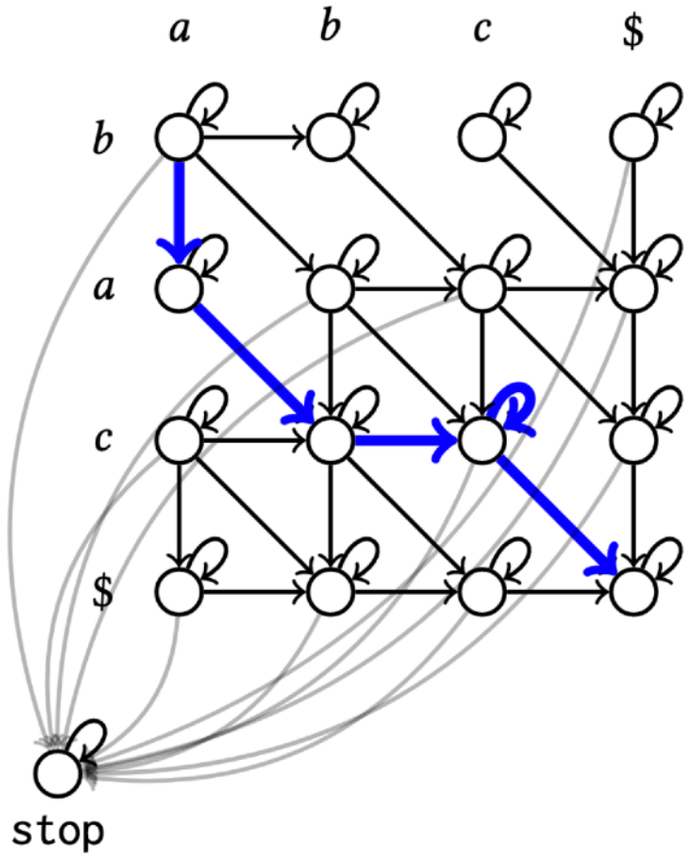
DP: Edit Distance and Knapsack



Admin

- Midterm review Tuesday Oct 27 at 7PM
  - Bring questions!!!
- No office hours next week (but some tomorrow!)
- No class on Wednesday next week (midterm instead)
- Reminder: email me if you're interested in a study group. I'll be sending out the groups tonight
- Assignment 4 back tonight hopefully, Assignment 5 over the weekend if possible

Edit Distance

- **Problem**. Given two strings find the minimum number of edits (letter insertions, deletions and substitutions) that transform one string into the other
- Measure of similarity between strings
- For example, the edit distance between FOOD and MONEY is at most four:

- Not hard to see that 3 edits don't work
- Edit distance = 4 in this case

Visualizing Alignment

- Visualize editing process by aligning source string above final string
- **Gaps:** represent insertions and deletions (insertions in the top string, deletion in bottom)
- **Mismatches**: columns with two different characters correspond to substitutions
- Cost of an alignment: number of gaps + mismatches

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Cost = 6 (three gaps + three mismatches)

Recursive Structure

- Before we develop a dynamic program, we need to figure out the recursive structure of the problem
- Our alignment representation has an optimal substructure
  - Suppose we have the mismatch/gap representation of the shortest edit sequence of two strings
    - If we remove the last column, the remaining columns must represent the shortest edit sequence of the remaining prefixes!

Recursive Structure

- Before we develop a dynamic program, we need to figure out the recursive structure of the problem
- For any prefix of our input strings $A[1,\ldots i]$ and $B[1,\ldots j]$, $1 \leq i \leq m$ and $1 \leq j \leq n$, the edit distance problem can be recursively formulated by using subproblem

- **Subproblem**.
    - Edit(i, j): edit distance between the strings $A[1,\ldots i]$ and $B[1,\ldots, j]$
- **Final answer**.
    - Edit(m, n)

Recurrence

- Three possibilities for the last column in the optimal alignment of $A[1,\ldots,i]$ and $B[1,\ldots,j]$, $i,j>0$:
- **Insertion**: Last entry in the top row is empty. In this case, Edit$(i,j)=$ Edit$(i,j-1)+1$

- **Deletion**: Last entry in bottom row is empty. In this case Edit$(i,j)=$ Edit$(i-1,\ j)+1$

- **Substitution**: Both rows have characters,
  if same:
  Edit$(i,j)=$ Edit$(i-1,\ j-1)$,
  else:
  Edit$(i,j)=$ Edit$(i-1,\ j-1)+1$

What About the Base Cases?

- Base cases occur when $i = 0$ or $j = 0$
- But these are easy to deal with
  - Edit$(0, j) = j$: Transforming an empty string to a string of length j, takes min j insertions
  - Edit$(i, 0) = i$: Transforming a string of length i to a string of length 0, takes min i deletions
- **Sanity check**, does our base case to compute the edit distance between two empty strings?
  - Yes, gives us 0.

Final Recurrence

- We have everything we need for our final recurrence

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

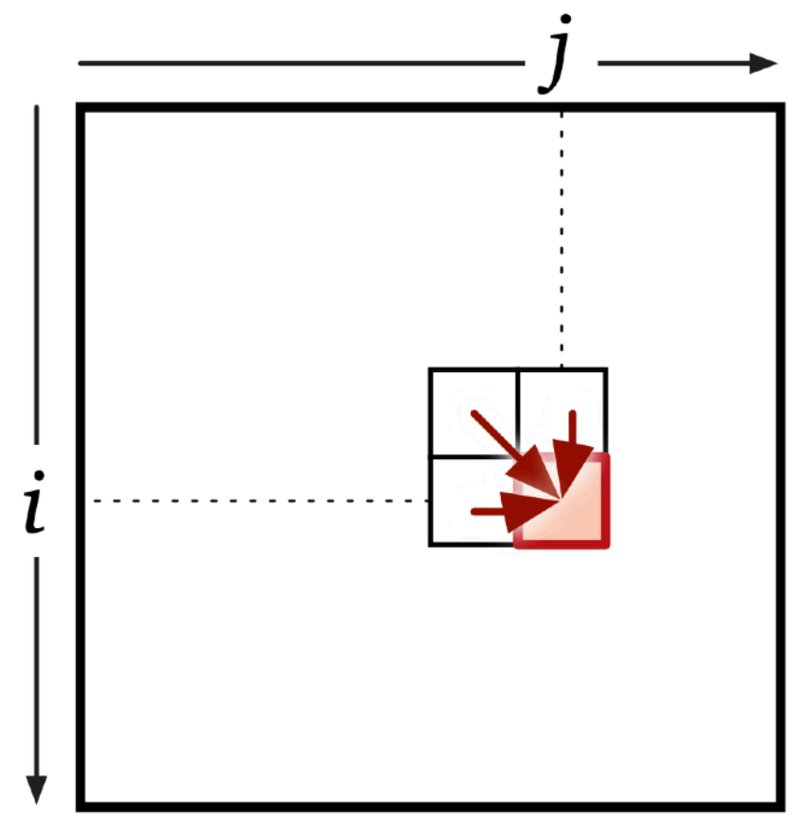
- Uses the shorthand: $[A[i] \neq B[j]]$ which is 1 if it is true (and they mismatch), and zero otherwise

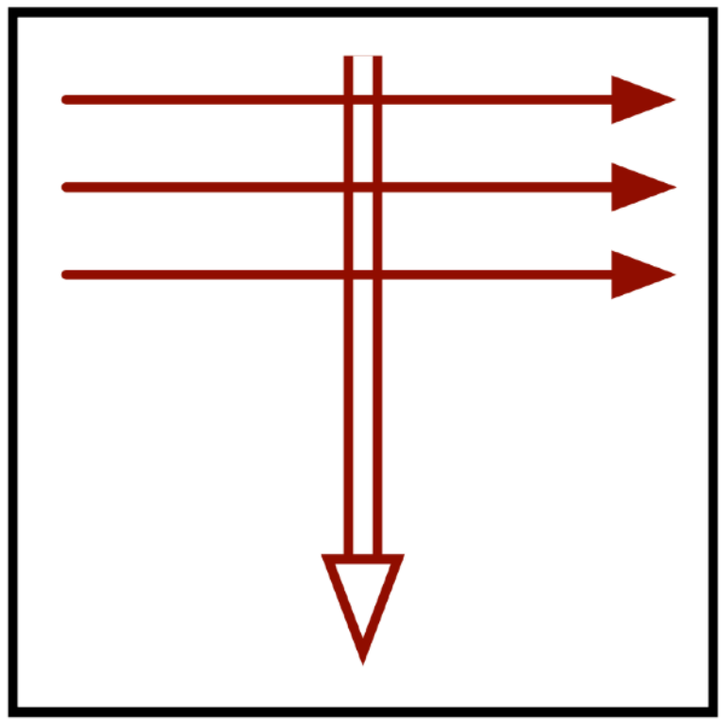
From Recurrence to DP

- We can now transform it into a dynamic program
- **Subproblems:** Each recursive subproblem Edit$[i, j]$ is defined by two indices $1 \leq i \leq m$ and $1 \leq j \leq n$
- **Memoization Structure:** We can memoize all possible values of Edit$[i, j]$ in a table/ two-dimensional array
- **Dependencies:** Each entry Edit$[i, j]$ depends on three neighboring entries: Edit$[i-1, j]$, Edit$[i, j-1]$ and Edit$[i-1, j-1]$
- **Evaluation order?**

From Recurrence to DP

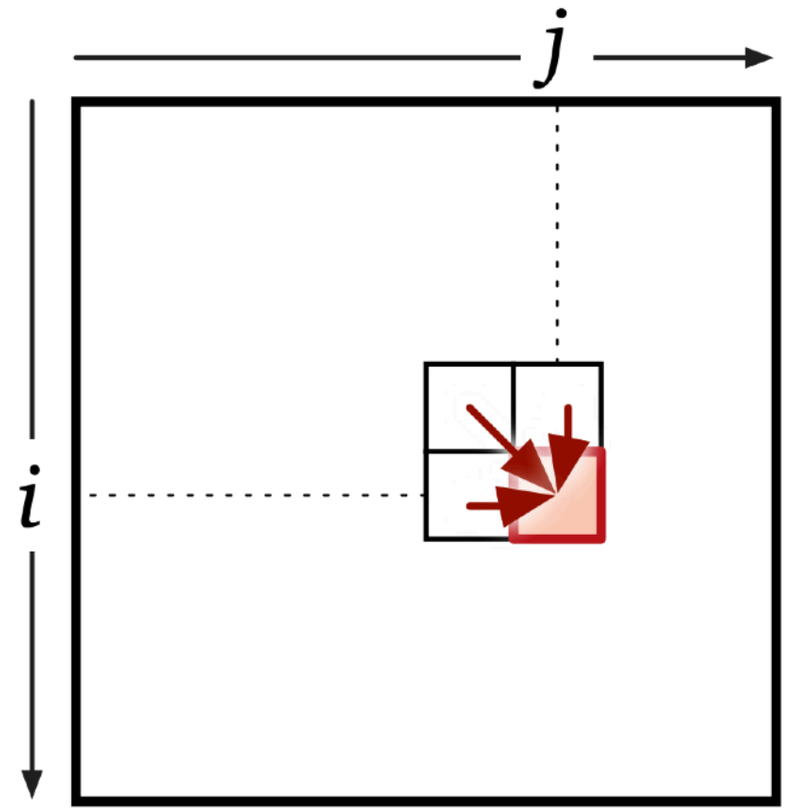
- **Evaluation order**
  - We can fill in *row major order*, which is row by row from top down, each row from left to right: when we reach an entry in the table, it depends only on filled-in entries

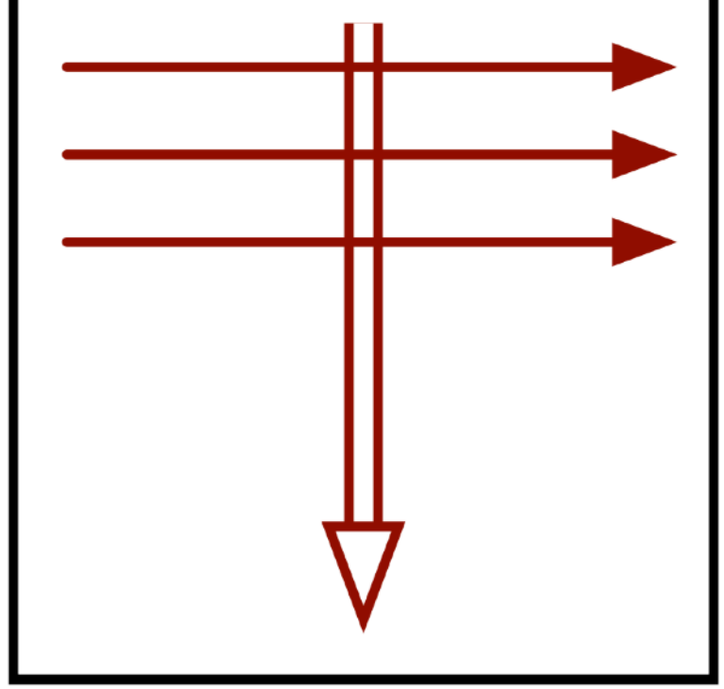




Space and Time

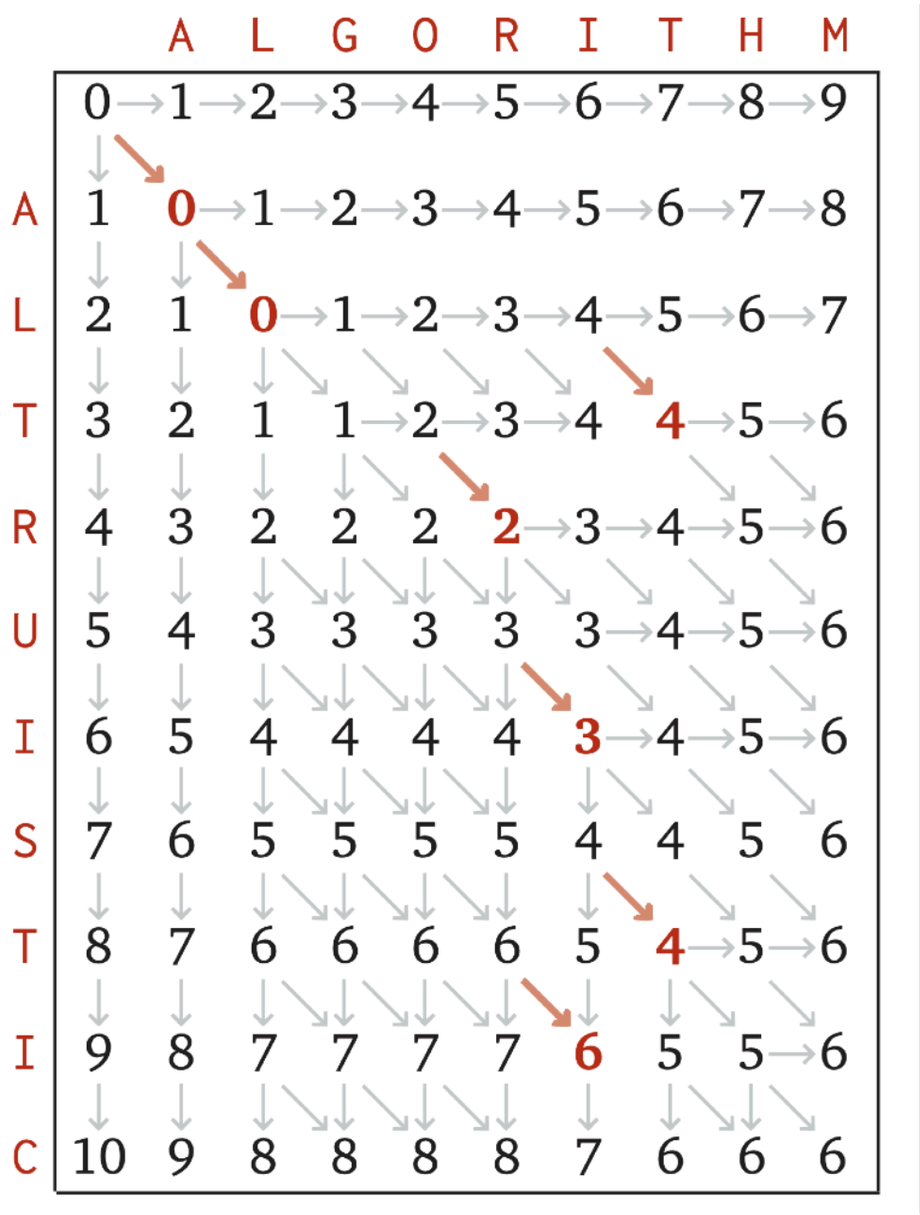
- The memoization uses $O(nm)$ space
- We can compute each Edit$[i, j]$in $O(1)$ time
- Overall running time: $O(nm)$





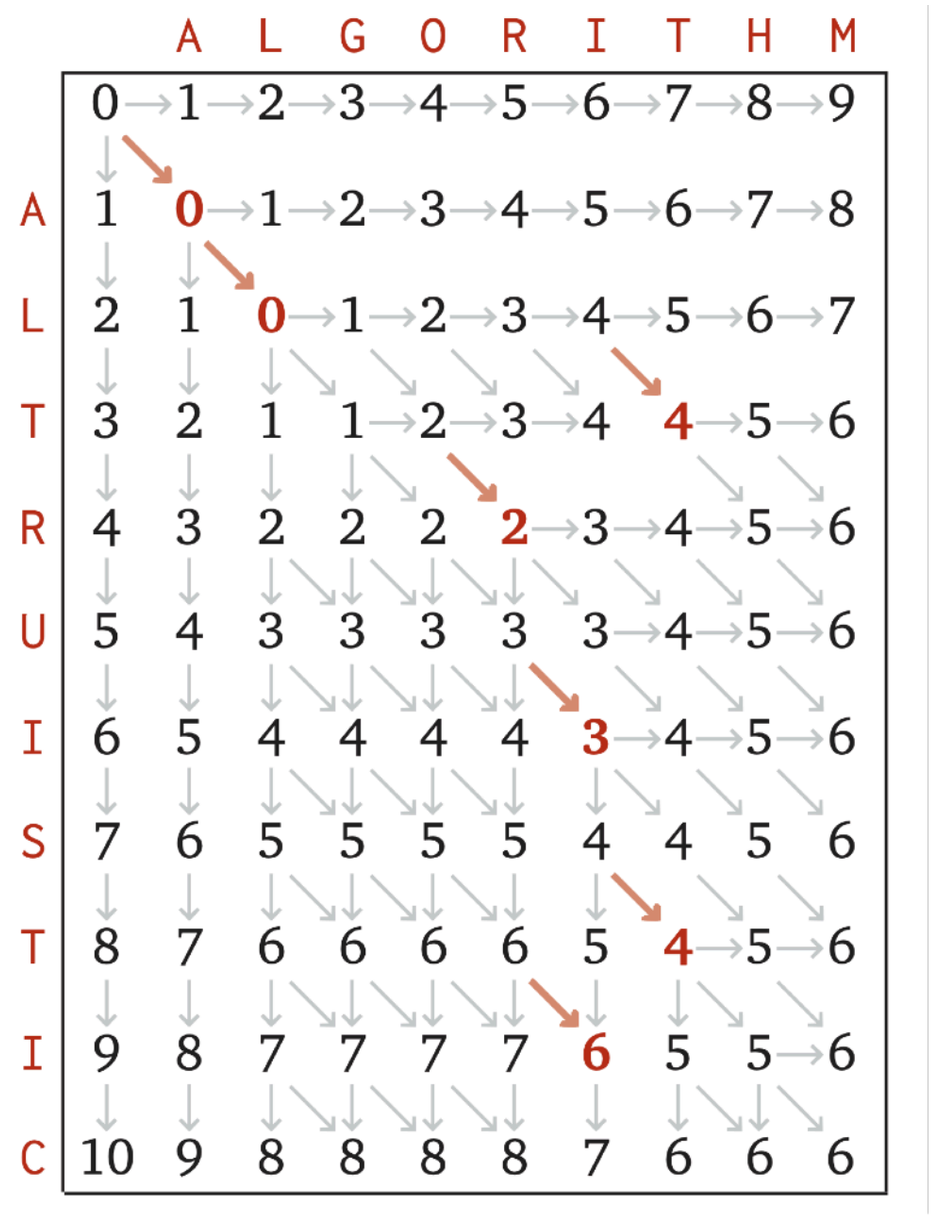
Memoization Table: Example

- Memoization table for **ALGORITHM** and **ALTRUISTIC**
- Bold numbers indicate where characters are same
- Horizontal arrow: deletion
- Vertical arrow: insertion
- Diagonal: substitution
- Bold red: free substitution
- Only draw an arrow if used in DP
- Any directed path of arrows from top left to bottom right represents an optimal edit distance sequence



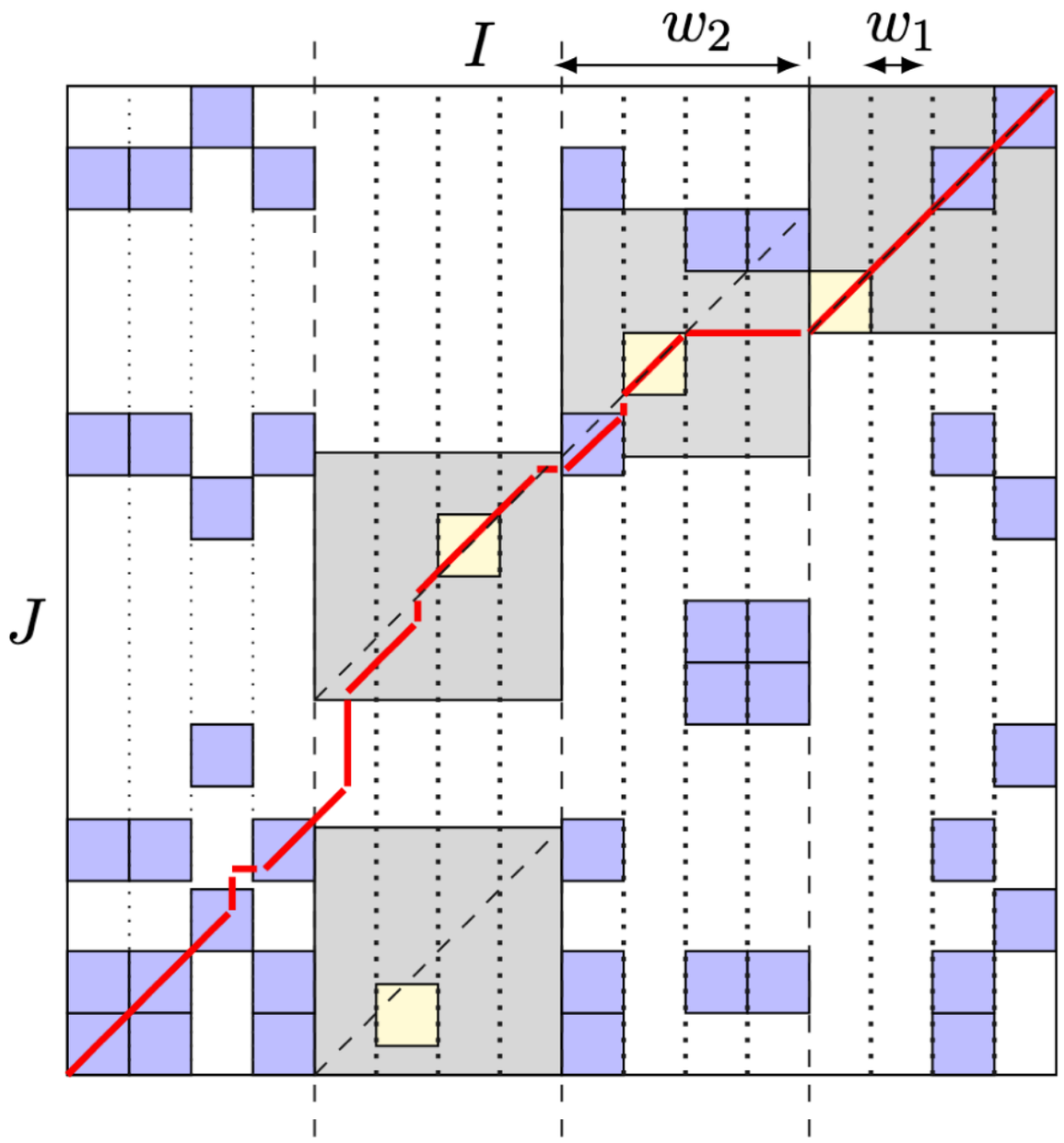
Reconstructing the Edits

- We don't need to store the arrow!
- Can be reconstructed on the fly in $O(1)$ time using the numerical values
- Once the table is built, we can construct the shortest edit distance sequence in $O(n+m)$ time
- Think at home: can you reconstruct the solution for the other dynamic programs we've seen in the same way?



Edit Distance Fun Facts

- Can we do better than $O(n^2)$ if $n = m$?
- Yes; can get $O(n^2/\log^2 n)$ [Masek Paterson '80] (uses "bit packing" trick called "Four Russians Technique")
- (Probably) cannot get $O(n^{2-\epsilon})$ for any constant $\epsilon > 0$ [Bakurs Indyk '15].
  - (In fact, some evidence that we can't get too many more log factors [AHWW16])
- Can approximate to any $1 + \epsilon$ factor in $O(n)$ time! [Andoni Nosatski '20]



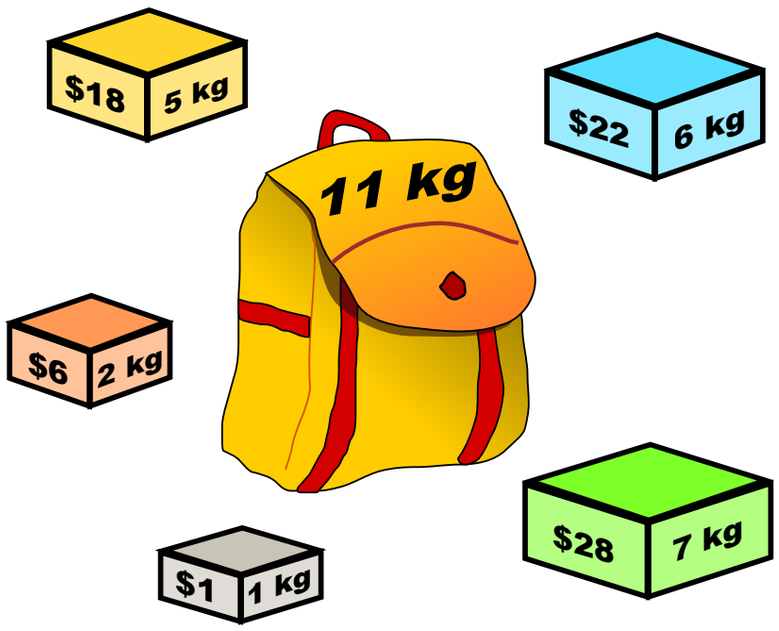
A figure from [CDGKS'18], the first approximation algorithm for edit distance. The idea: rule out large portions of the dynamic programming table

Knapsack Problem

- **Problem**. Pack a knapsack to maximize total value
- There are n items, each with weight w_i and value v_i, where $v_i, w_i > 0$. Weights must be integers!
- Knapsack has total capacity C
- Output: subset S of items fit in the knapsack, that is, $\sum_{i \in S} w_i \leq C$ and maximizes the total value $\sum_{i \in S} v_i$
- **Assumption**. All values are integral

Knapsack Problem

- Example (Knapsack capacity C = 11)
    - {1, 2, 5} has value $35 (and weight 10)
    - {3, 4} has value $40 (and weight 11)

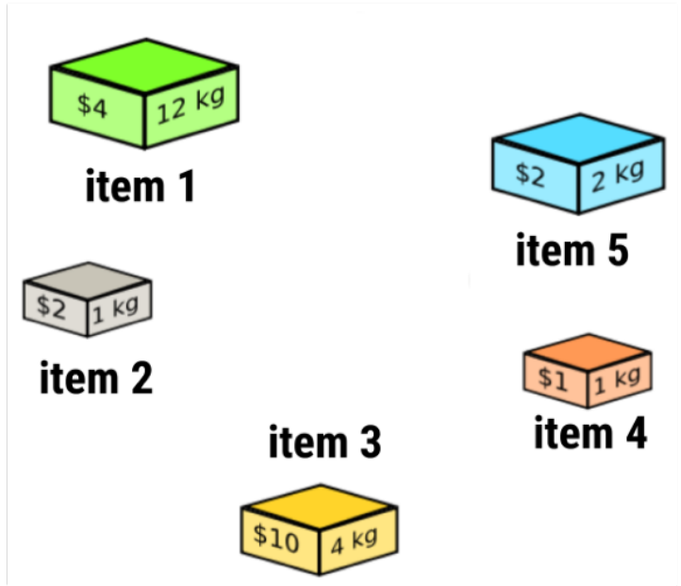


i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

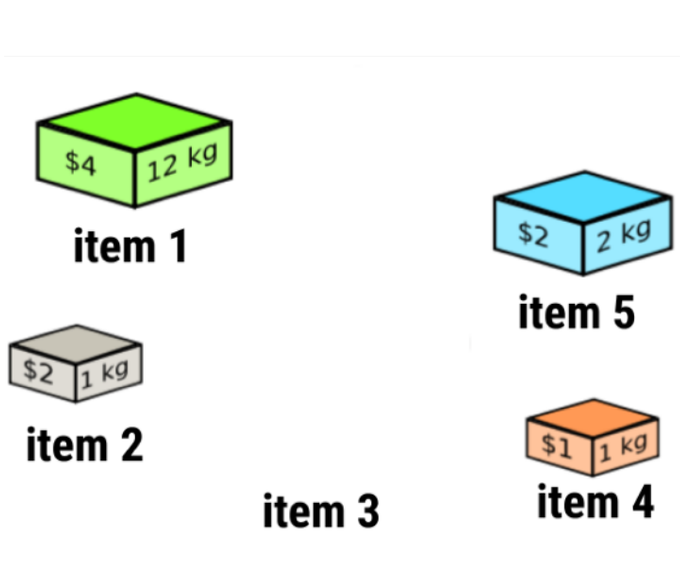
knapsack instance
(weight limit W = 11)

Subproblems and Optimality

- When items are selected we need to fill the remaining capacity optimally
- Subproblem associated with a given remaining capacity can be solved in different ways



Partial Selection #1



Partial Selection #2

- In both cases, remaining capacity: 13 but items left are different

Idea #1: Capacity Table

- Let's create a table T where $T[c]$ contains the optimal solution using capacity $\leq c$.
- Optimal solution: $T[C]$
- How do come up with a recurrence?
- Not obvious with just capacities

capacity	items	value
c = 0		$0
c = 1	$2/1kg	$2
c = 2	$2/1kg $1/1kg	$3
c = 3	$2/1kg $2/2kg	$4
c = 4	$10/4kg	$10
c = 5	$2/1kg $10/4kg	$12
c = 6	$2/1kg $10/4kg $1/1kg	$13
c = 7	[activity]	
...	...	...

Table for the item set

$4/12kg $2/1kg $10/4kg $1/1kg $2/2kg

DP: Right Recurrence

- What else can we keep track of to get a recurrence with an optimal substructure?

- Let $T[j, c]$ be the optimal solution using items $[1,\ldots j]$ with total capacity $\leq c$

- What are our two cases?
- Case 1. If item j is not in the optimal solution
    - $T[j, c] = T[j-1, c]$
- Case 2. If item j is in the optimal solution then
    - $T[j, c] = v_j + T[j-1, c-w_j]$

Recurrence & Memoization

- **Base case**.
  - $T[j, c] = 0$ if $j = 0$, or $c = 0$
- For $j, c > 0$
  - $T[j, c] = \max\{T[j-1,\ c],\ v_j + T[j-1,\ c - w_j]\}$
- Now that we have the recurrence, we can memoize and figure out the evaluation order
- We will store $T[j, c]$ for $1 \le j \le n, \quad 1 \le c \le C$
- **Evaluation order?**
  - Row by row (i.e. item by item: for each item fill in each capacity one by one)
- **Final answer?** $T[n, c]$

Running Time

- Takes $O(1)$ to fill out a cell, $O(nC)$ total cells
- Is this polynomial? By which I mean polynomial in the *size of the input*
- How large is the input to knapsack?
  - Store n items, plus need to store C
  - $O(n + \log C)$
- Is $O(nC)$ polynomial?
  - No!
  - “Pseudopolynomial” - polynomial in the *value* of the input
- To think about: does this work if the weights are not integers?

Acknowledgments

- Some of the material in these slides are taken from
    - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
    - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)