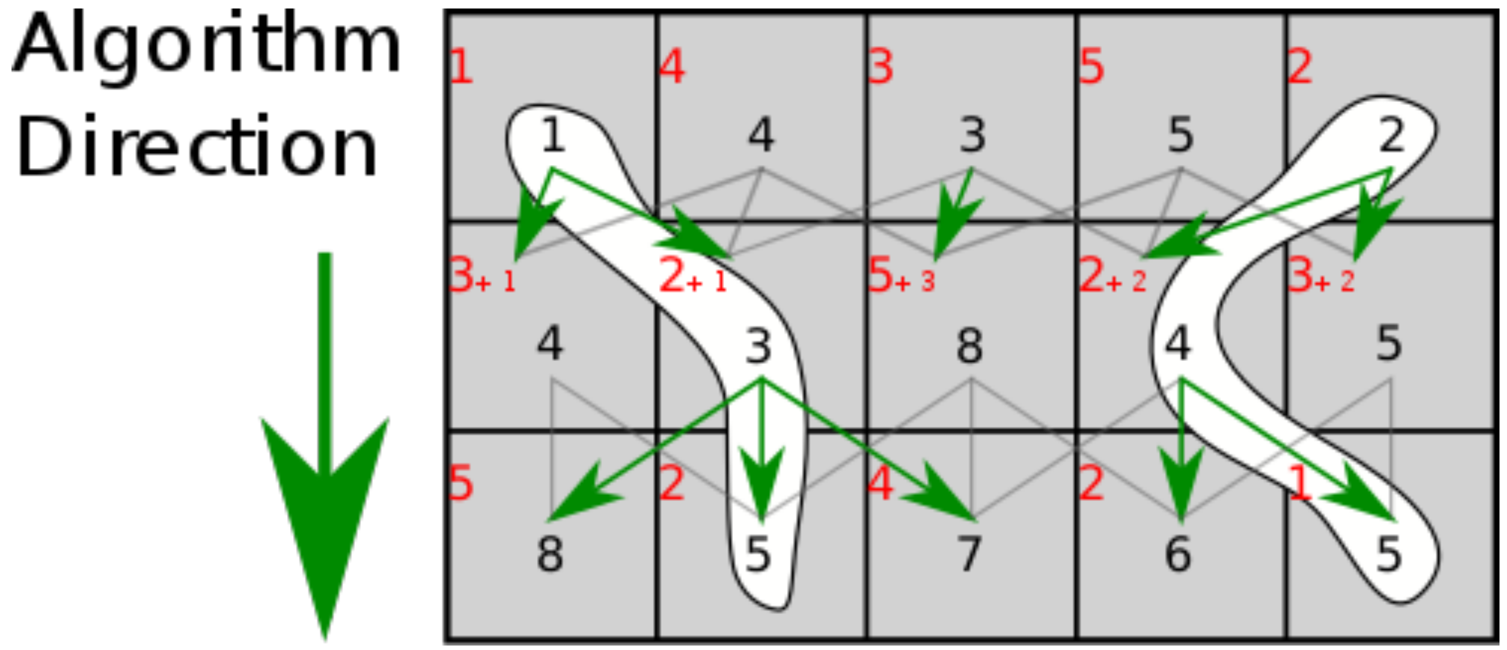
Dynamic Programming II: Edit Distance



Admin

- Assignment 5 is due a day early
- **Midterm** next Friday **(April 2);** no class
    - Will be released 10.00 am Friday; can be taken in any 24 hour period between **10.00 am Friday to 10.00 am Sunday**

Longest Increasing Subsequence

Reading: Chapter 1, Erickson

Longest Increasing Subsequence

- Given a sequence of integers as an array $A[1,\ldots n]$, find the longest subsequence whose elements are in increasing order
- Find the longest possible sequence of indices $1 \leq i_1 < i_2 < \ldots < i_\ell \leq n$ such that $A[i_k] < A[i_{k+1}]$

$$\boxed{\mathbf{1}\ \ \mathbf{2}\ \ 10\ \ \mathbf{3}\ \ 7\ \ 6\ \ \mathbf{4}\ \ \mathbf{8}\ \ \mathbf{11}}$$

- Length of the longest increasing subsequence above is 6
- To simplify, we will only compute **length of the LIS**

Base Case & Final Answer

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

Base Case. $L[1] = 1$

Final answer. $\max_{1 \leq i \leq n} L[i]$

LIS: Recurrence

$$\boxed{\begin{array}{l} L[j] = 1 + \max\{L[i] \mid i < j \text{ and } A[i] < A[j]\} \\ \text{Assuming } \max \varnothing = 0 \end{array}}$$

Recursion → DP

- If we used recursion (without memoization) we'll be inefficient—we'll do a lot of repeated work
- Once you have your recurrence, the remaining pieces of the dynamic programming algorithm are
    - **Evaluation order.** In what order should I evaluate my subproblems so that everything I need is available to evaluate a new subproblem?
        - For LIS we just left-to-right on array indices
    - **Memoization structure.** Need a table (array or multi-dimensional array) to store computed values
        - For LIS, we just need a one dimensional array
        - For others, we may need a table (two-dimensional array)

LIS Analysis

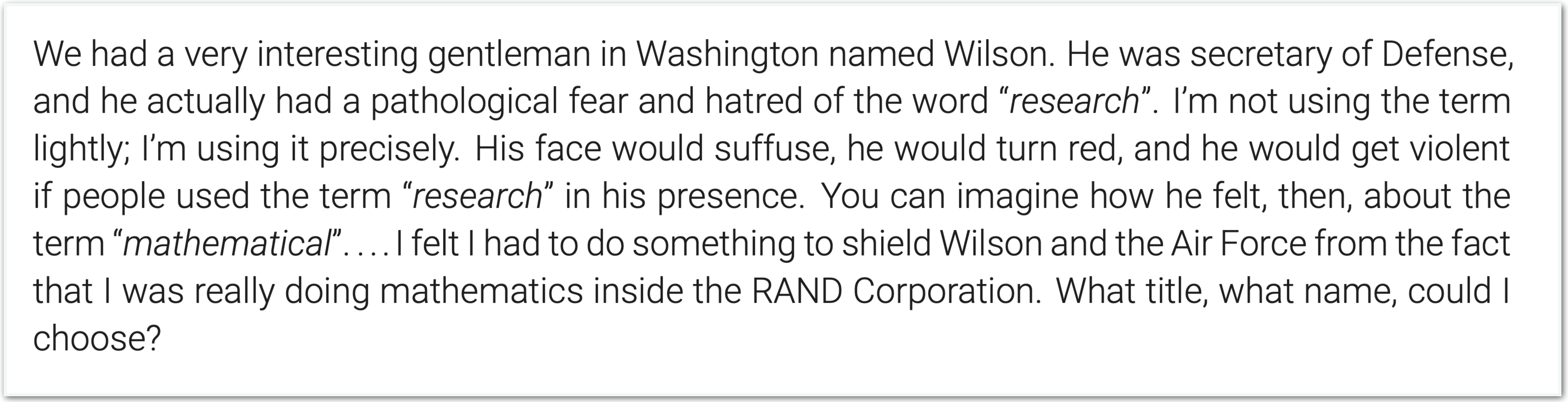
- Correctness
    - Follows from the recurrence using induction
- Running time?
    - Solve $O(n)$ subproblems
    - Each one requires $O(n)$ time to take the min
    - $O(n^2)$
    - An Improved DP solution takes $O(n \log n)$
- Space?
    - $O(n)$ to store array $L[]$

Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure
- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem
- **State the base case(s)**. The subproblem thats so small we know the answer to it!
- **State the final answer.** (In terms of the subproblem)
- **Choose a memoization data structure.** Where are you going to store already computed results? (Usually a table)
- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which ones? Using these dependencies, identify an evaluation order
- **Analyze space and running time.** As always!

Name: Dynamic Programming

- Formalized by Richard Bellman in the 1950s



We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word "*research*". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term "*research*" in his presence. You can imagine how he felt, then, about the term "*mathematical*". . . . I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?

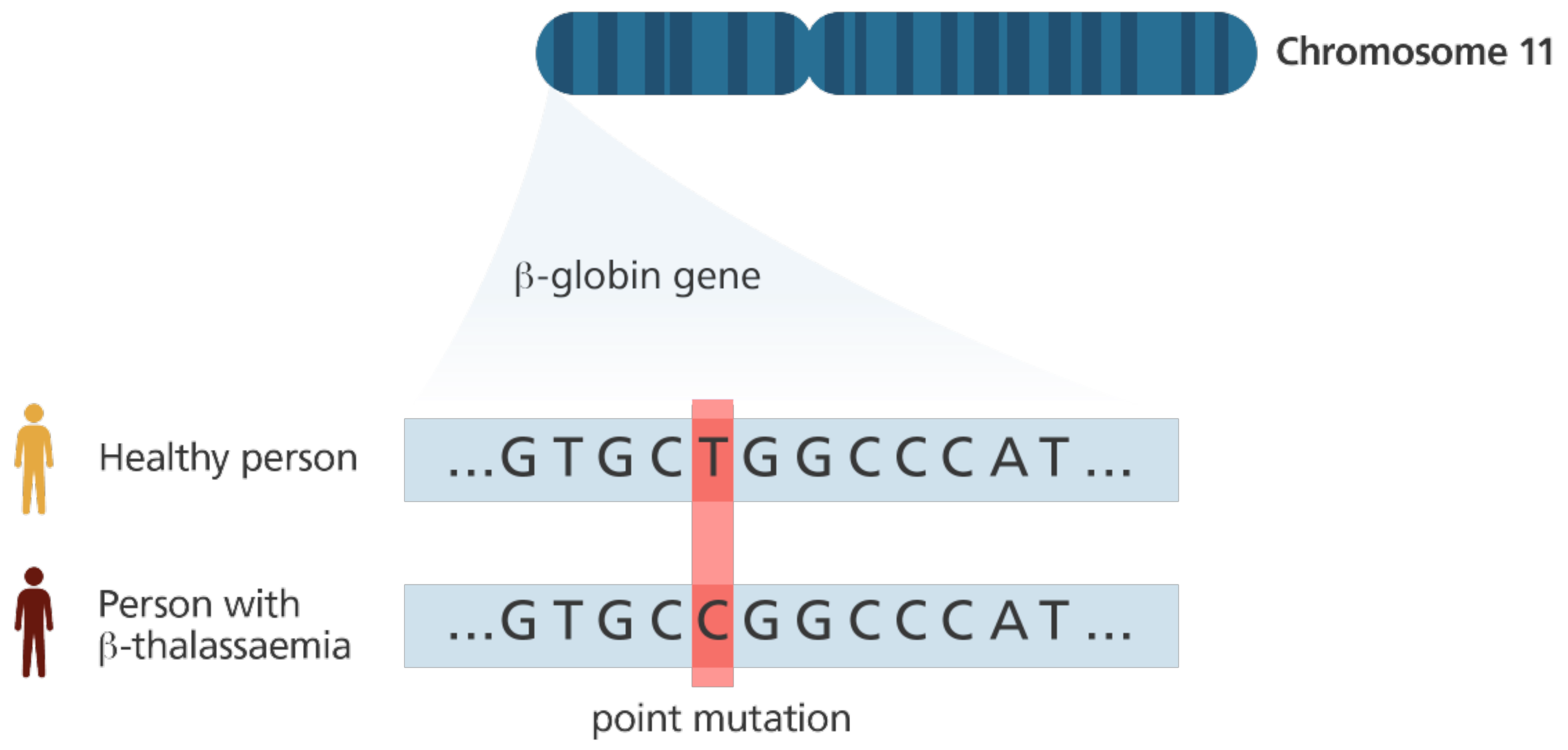
- Chose the name **"dynamic programming"** to hide the mathematical nature of the work from military bosses

Edit Distance

Reading: Chapter 1, Erickson

Motivation

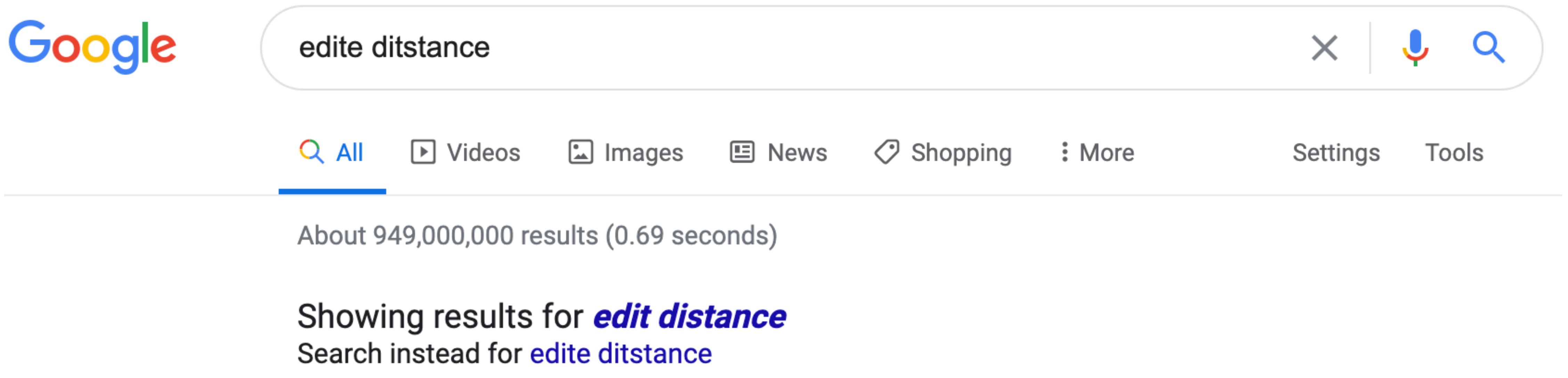
- **Edit distance**: is a metric that captures the similarity between two strings


Chromosome 11
β-globin gene
Healthy person
...G T G C T G G C C C A T ...
Person with β-thalassaemia
...G T G C C G G C C C A T ...
point mutation

DNA sequencing: finding similarities between two genome sequences

Motivation

- **Edit distance**: is a metric that captures the similarity between two strings


edite ditstance

About 949,000,000 results (0.69 seconds)

Showing results for ***edit distance***
Search instead for edite ditstance

Text processing: finding similar strings and NLP

Problem Defintion

- **Problem**. Given two strings $A = a_1 \cdot a_2 \cdots a_n$ and $B = b_1 \cdot b_2 \cdots b_m$ find the edit distance between them
- Edit distance between A and B is the smallest number of the following operations that are needed to transform A into B
    - Replace a character (substitution)
    - Delete a character
    - Insert a character

$$\begin{tikzcd}
\text{riddle} \arrow[r, "\text{delete}"] & \text{ridle} \arrow[r, "\text{substitute}"] & \text{riple} \arrow[r, "\text{insert}"] & \text{triple}
\end{tikzcd}$$

Edit distance(riddle, triple): 4

Structure of the Problem

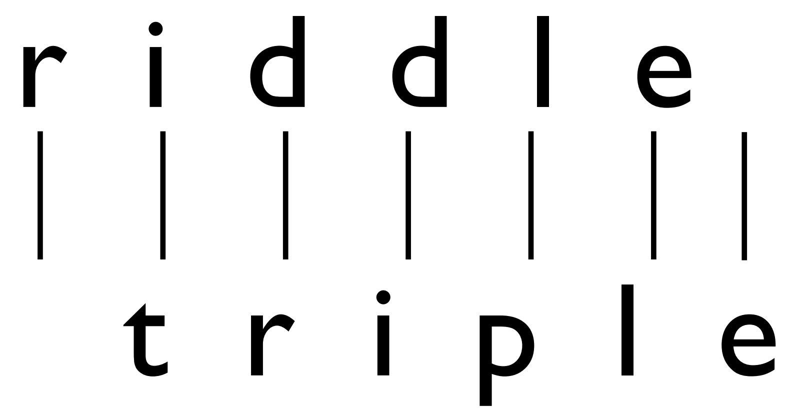
- **Problem**. Given two strings $A = a_1 \cdot a_2 \cdots a_n$ and $B = b_1 \cdot b_2 \cdots b_m$ find the edit distance between them
- Notice that the process of getting from string A to string B by doing substitutions, inserts and deletes is reversible
- **Inserts** in one string correspond to **deletes** in another

$$\begin{tikzcd}
\text{riddle} \arrow[r, "\text{delete}"] & \text{ridle} \arrow[r, "\text{substitute}"] & \text{riple} \arrow[r, "\text{insert}"] & \text{triple} \\
\text{riddle} & \text{ridle} \arrow[l, "\text{insert}"'] & \text{riple} \arrow[l, "\text{substitute}"'] & \text{triple} \arrow[l, "\text{delete}"']
\end{tikzcd}$$

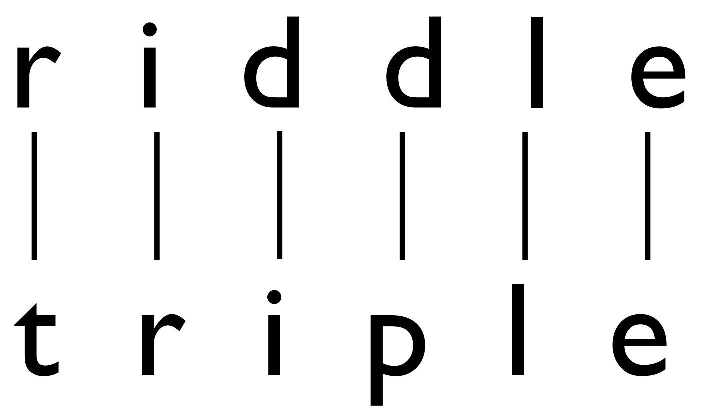
Edit distance(riddle, triple): 4

Sequence Alignment

- We can visualize the problem of finding the edit distance as an the problem of finding the best alignment between two strings
- **Gaps** in alignment represent inserts/deletes
- **Mismatches** in alignment represent substitutes
- Cost of an alignment = **number of gaps + mismatches**
- Edit distance: minimum cost alignment



cost = 7



cost = 4

Sequence Alignment

```
prin-ciple
|||| |||xx
prinncipal
(1 gap, 2 mm)
```

```
prin-cip-le
|||| ||| |
prinncipal-
(3 gaps, 0 mm)
```

```
misspell
||| ||||
mis-pell
(1 gap)
```

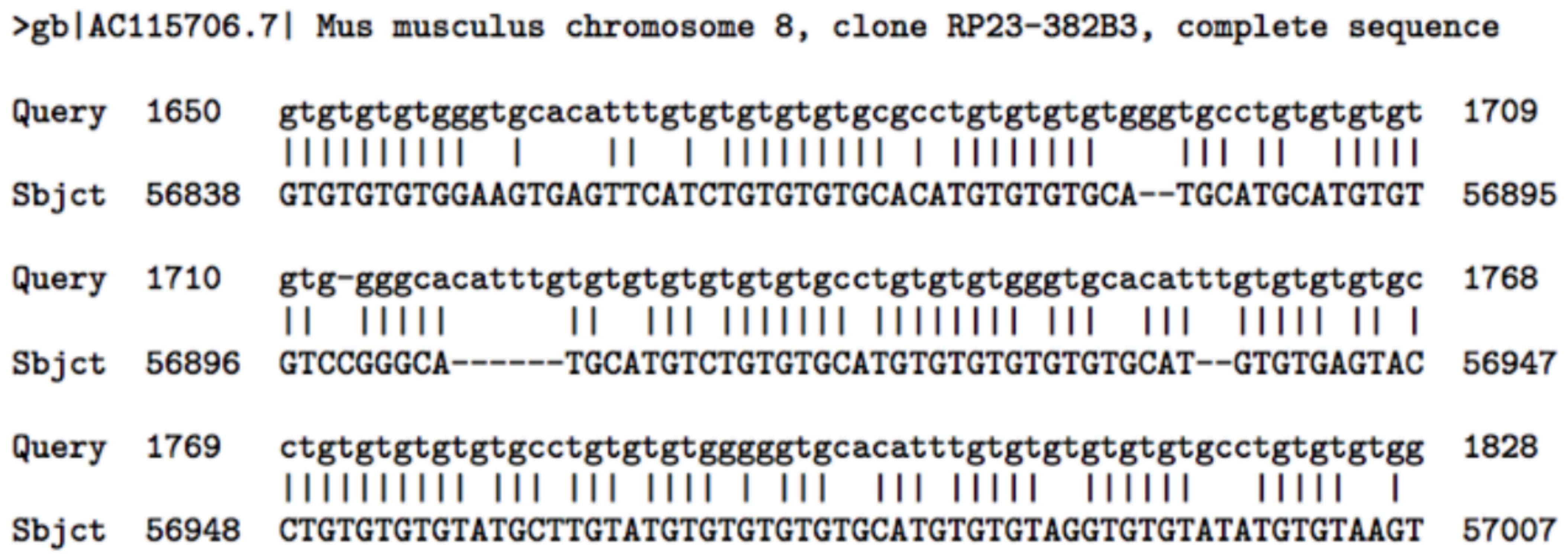
```
prehistoric
   ||||||||
---historic
(3 gaps)
```

```
aa-bb-ccaabb
|x || | | |
ababbbc-a-b-
(5 gaps, 1 mm)
```

```
al-go-rithm-
|| xx ||x |
alKhwariz-mi
(4 gaps, 3 mm)
```

https://www.cs.cmu.edu/~ckingsf/class/02713-s13

Sequence Alignment

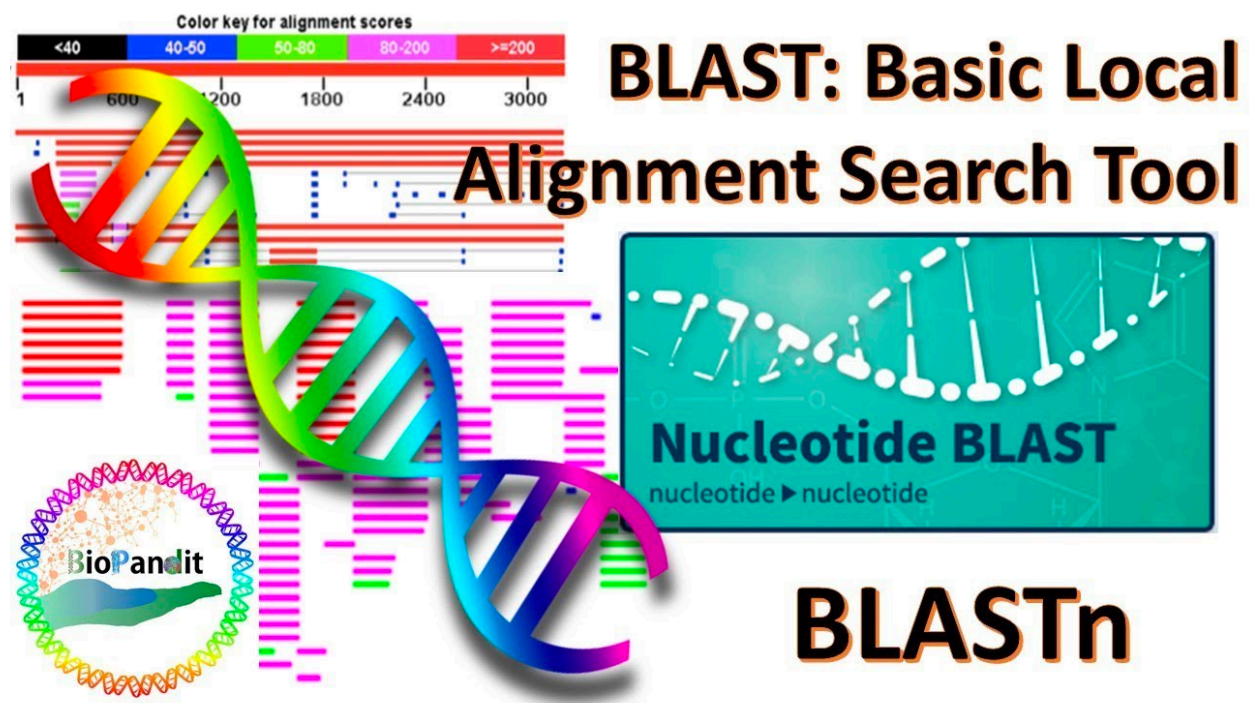


```
>gb|AC115706.7| Mus musculus chromosome 8, clone RP23-382B3, complete sequence

Query  1650   gtgtgtgtgggtgcacatttgtgtgtgtgtgcgcctgtgtgtgtgggtgcctgtgtgtgt  1709
              ||||||||||  |    ||  | ||||||||| | ||||||||    ||| ||  |||||
Sbjct  56838  GTGTGTGTGGAAGTGAGTTCATCTGTGTGTGCACATGTGTGTGCA--TGCATGCATGTGT  56895

Query  1710   gtg-gggcacatttgtgtgtgtgtgtgtgcctgtgtgtgggtgcacatttgtgtgtgtgc  1768
              ||  |||||      ||  ||| ||||||| |||||||| |||  |||  ||||| || |
Sbjct  56896  GTCCGGGCA------TGCATGTCTGTGTGCATGTGTGTGTGTGTGCAT--GTGTGAGTAC  56947

Query  1769   ctgtgtgtgtgtgcctgtgtgtgggggtgcacatttgtgtgtgtgtgtgcctgtgtgtgg  1828
              |||||||||| ||| ||| |||| | |||  ||| |||||  ||||||   |||||  |
Sbjct  56948  CTGTGTGTGTATGCTTGTATGTGTGTGTGTGCATGTGTGTAGGTGTGTATATGTGTAAGT  57007
```

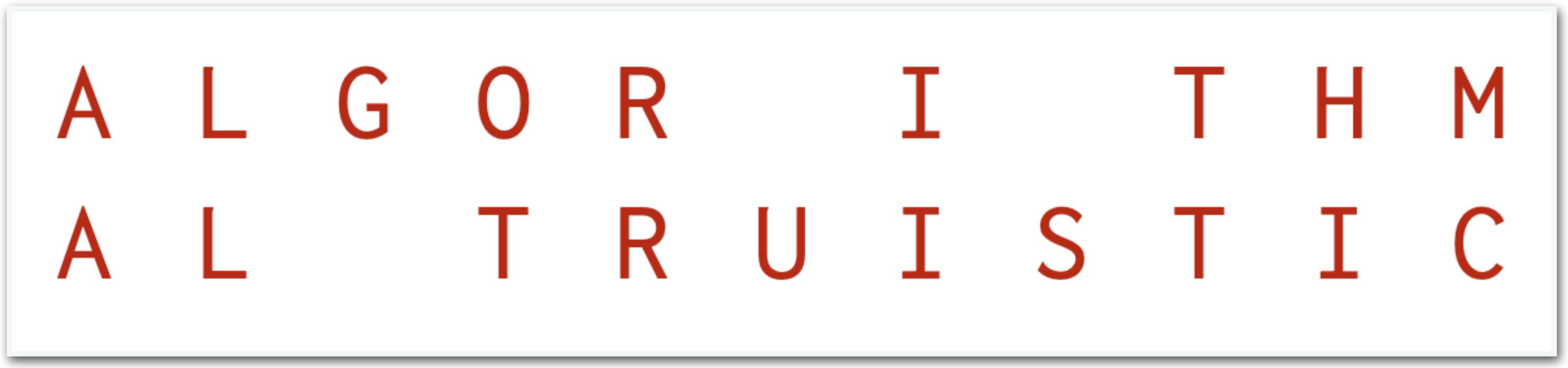


Sequence Alignment Problem

- Find an alignment of the two strings A, B where
  - each character a_i in A is matched to a string b_j in B or unmatched
  - each character b_j in A is matched to a string a_i in A or unmatched
- cost(a_i, b_j) $= 0$ if $a_i = b_j$, else cost(a_i, b_j) $= 1$
- cost of an unmatched letter (gap) $= 1$
- Total cost = # unmatched (gaps) $+ \sum_{a_i, b_j}$ cost(a_i, b_j)
- **Goal**. Compute edit distance by finding an alignment of the minimum total cost

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!



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

Subproblem

- **Subproblem**

$$\boxed{\text{Edit}(i, j)\text{: edit distance between the strings } a_1 \cdot a_2 \cdots a_i \text{ and } b_1 \cdot b_2 \cdots b_j, \text{ where } 0 \le i \le m \text{ and } 0 \le j \le n}$$

- **Final answer**

$$\boxed{\text{Edit}(n, m)}$$

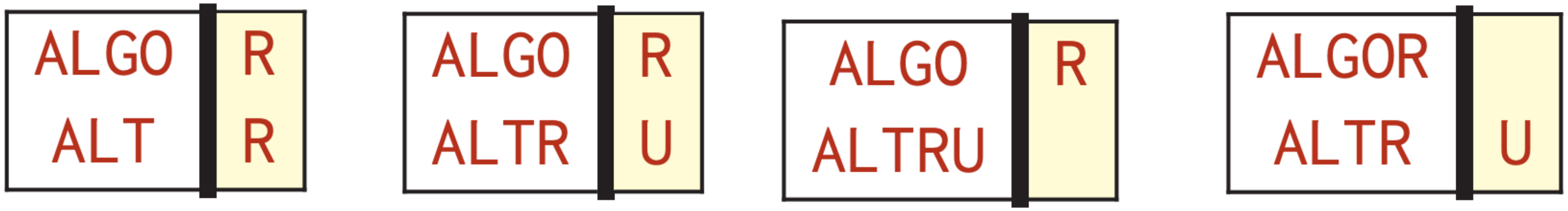
Base Cases

- We have to fill out a two-dimensional array to memoize our recursive dynamic program
- Let us think about which rows/columns can we fill initially
- Edit$(i, 0)$: Min number of edits to transform a string of length i to an empty string

$$\boxed{\begin{array}{l} \text{Edit}(i,\ 0) = i \ \text{ for } 0 \le i \le n \\ \text{Edit}(0,\ j) = j \ \text{ for } 0 \le j \le m \end{array}}$$

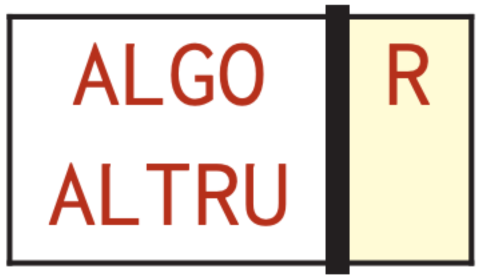
Recurrence

- Imagine the optimal alignment between the two strings
- What are the possibilities for the last column?
  - It could be that both letters match: cost 0
  - It could be that both letters do not match: cost 1
  - It could be that there an unmatched character (gap): either from A or from B: cost 1

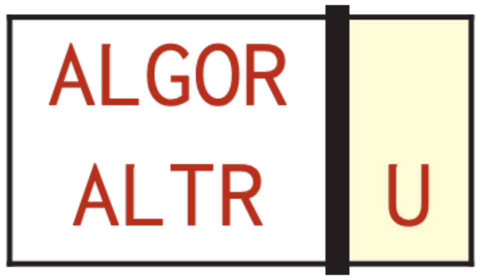


Recurrence

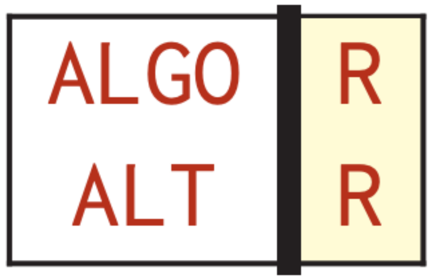
- Three possibilities for the last column in the optimal alignment of $a_1 \cdot a_2 \cdots a_i$ and $b_1 \cdot b_2 \cdots b_j$:
- **Case 1.** Only one row has a character:
  - Case 1a. Letter a_i is unmatched
    Edit$(i, j) =$ Edit$(i, j - 1) + 1$



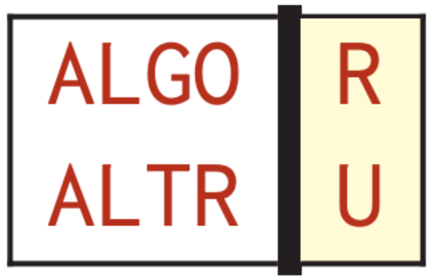
  - Case 1b. Letter b_j is unmatched
    Edit$(i, j) =$ Edit$(i, j - 1) + 1$



- **Case 2**: Both rows have characters:
  - Case 2a. Same characters:
    Edit$(i, j) =$ Edit$(i - 1, j - 1)$



  - Case 2b. Different characters:
    Edit$(i, j) =$ Edit$(i - 1, j - 1) + 1$



Final Recurrence

- For $1 \leq i \leq n$ and $1 \leq j \leq m$, we have:

$$\mathrm{Edit}(i,j) = \min \begin{cases} \mathrm{Edit}(i, j-1) + 1 \\ \mathrm{Edit}(i-1, j) + 1 \\ \mathrm{Edit}(i-1, j-1) + (a_i \neq b_j) \end{cases}$$

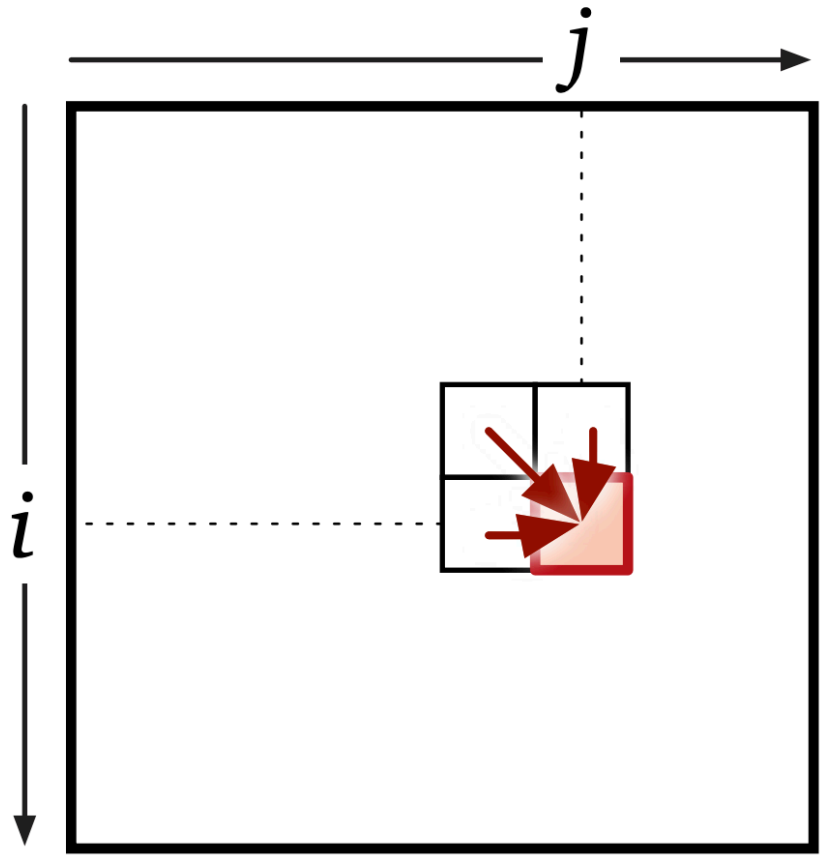
- Uses the shorthand: $(a_i \neq b_j)$ which is 1 if it is true (and they mismatch), and zero otherwise

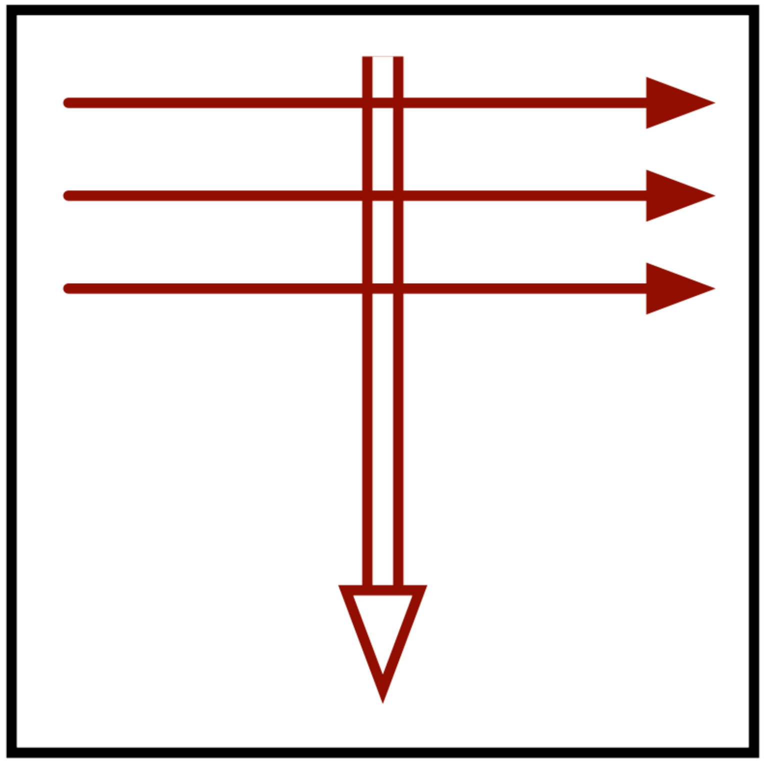
From Recurrence to DP

- We can now transform it into a dynamic program
- **Memoization Structure**: We can memoize all possible values of Edit(i, j) in a table/ two-dimensional array of size $O(nm)$:
    - Store Edit$[i, j]$ in a 2D array; $0 \leq i \leq n$ and $0 \leq j \leq m$
- **Evaluation order**:
    - Is interesting for a 2D problem
    - Based on dependencies between subproblems
    - We want values required to be already computed

From Recurrence to DP

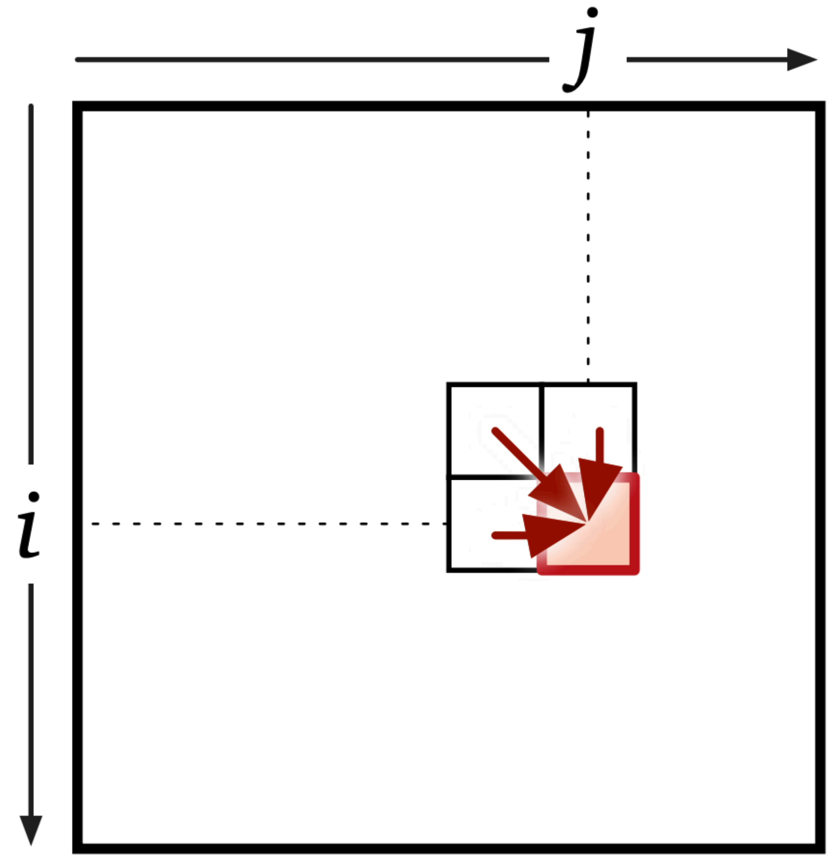
- **Evaluation order**
  - We can fill in **row major order**, which is row by row from top down, each row from left to right: when we reach an entry in the table, it depends only on filled-in entries

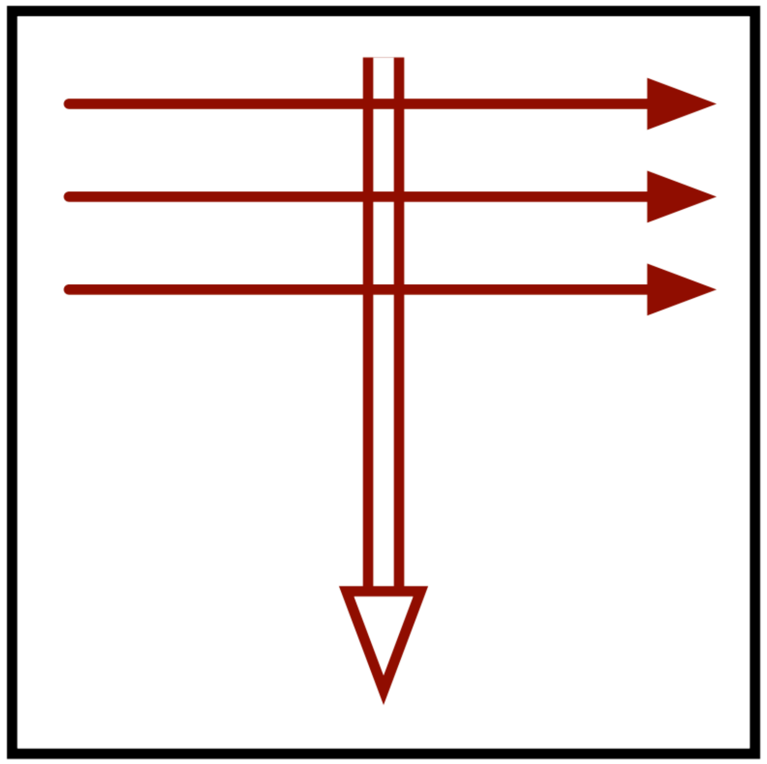




Space and Time

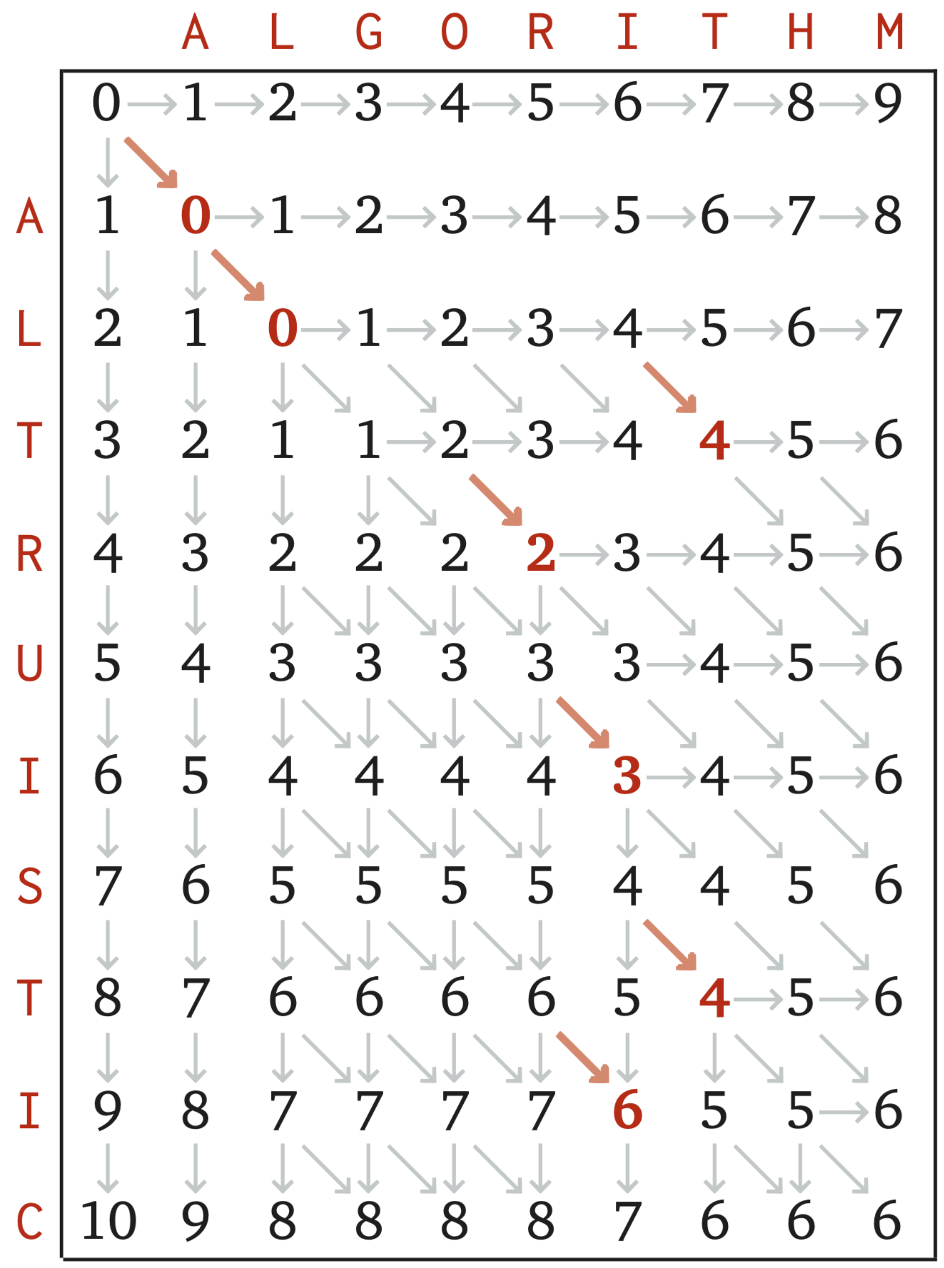
- The memoization uses $O(nm)$ space
- We can compute each Edit$[i, j]$in $O(1)$ time
- Overall running time: $O(nm)$





Memoization Table: Example

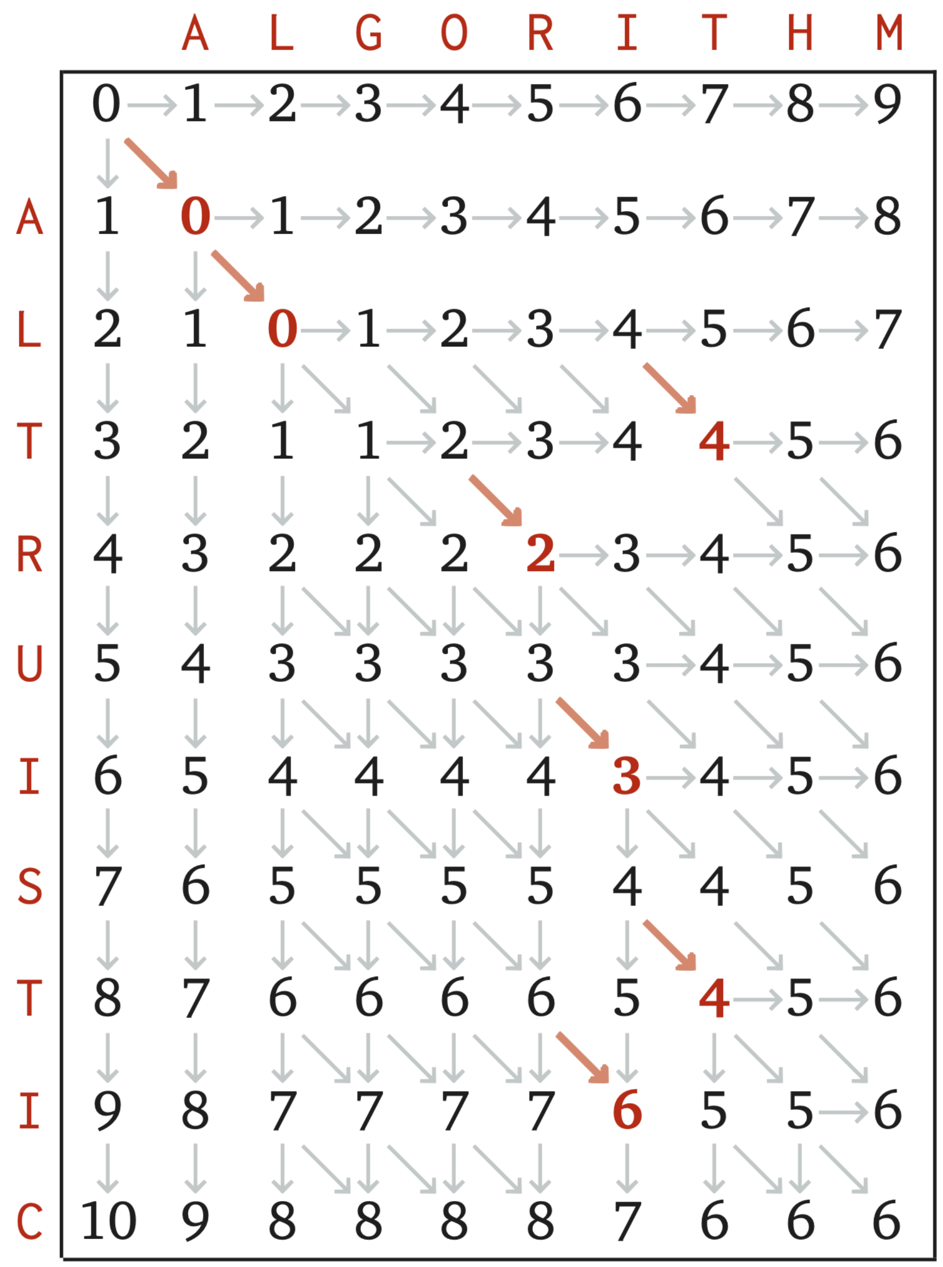
- Memoization table for **ALGORITHM** and **ALTRUISTIC**
- Bold numbers indicate where characters are same
- Horizontal arrow: deletion in A
- Vertical arrow: insertion in A
- Diagonal: substitution
- Bold red: free substitution
- Only draw an arrow if used in DP
- Any directed path of arrows from top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



		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

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?



		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

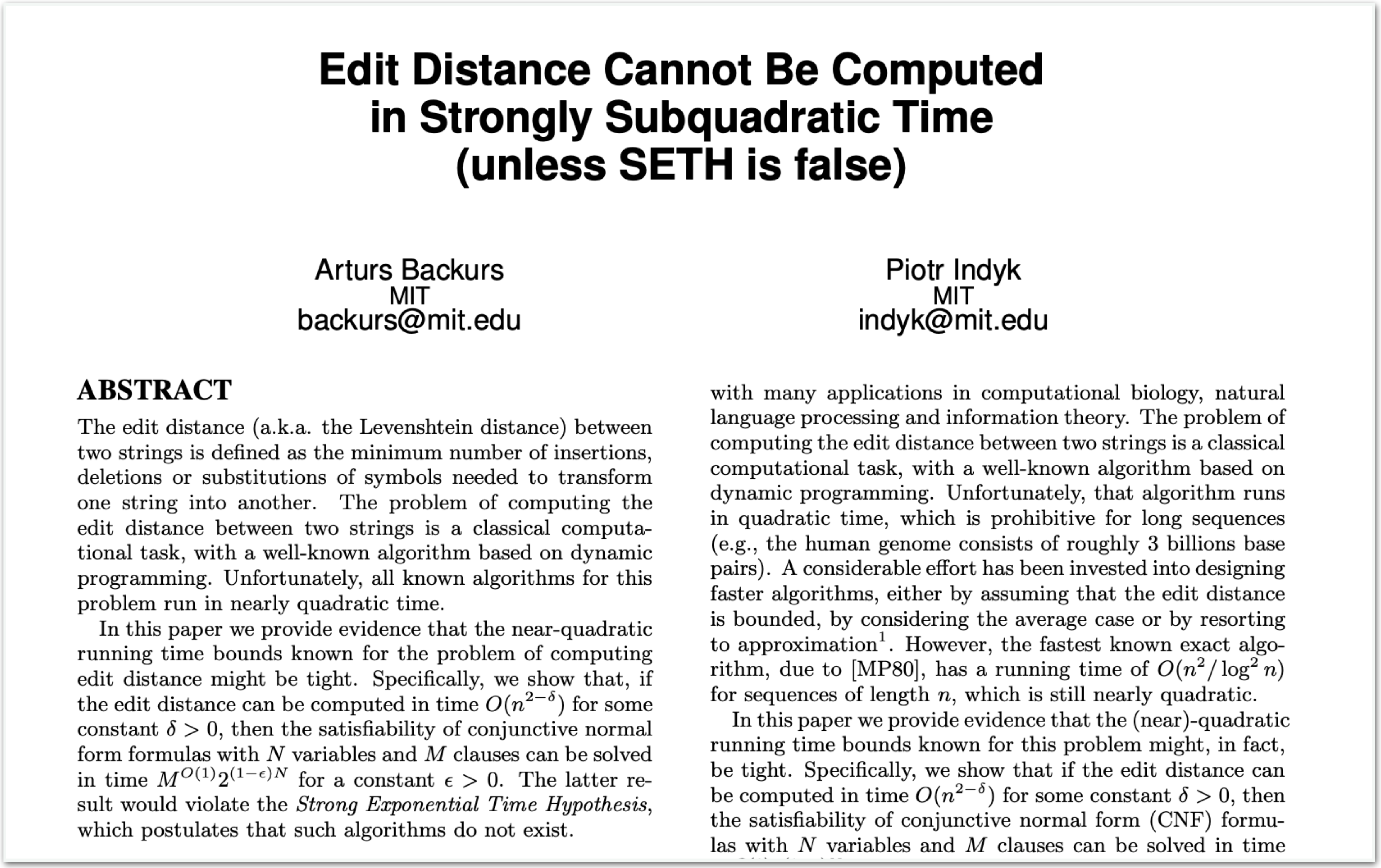
Edit Distance Fun Facts

- Can we compute edit distance using less space?
  - $O(nm)$ is huge for large genomic sequences
  - If we only care cost cost, we only need $O(n+m)$ space (just keep row above current)
  - But this doesn't let us recreate the path
- **Hirschberg's algorithm:** Can compute the actual path (edits) in $O(nm)$ time using $O(n+m)$ space
  - Neat divide-and-conquer trick to save space

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")
- Can we get an algorithm for edit distance with runtime $O(n^{2-\epsilon})$, e.g. $O(n^{1.9})$?
  - Probably not (unless a well-known conjecture breaks)

Edit Distance Fun Facts



Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false)

Arturs Backurs
MIT
backurs@mit.edu

Piotr Indyk
MIT
indyk@mit.edu

ABSTRACT

The edit distance (a.k.a. the Levenshtein distance) between two strings is defined as the minimum number of insertions, deletions or substitutions of symbols needed to transform one string into another. The problem of computing the edit distance between two strings is a classical computational task, with a well-known algorithm based on dynamic programming. Unfortunately, all known algorithms for this problem run in nearly quadratic time.

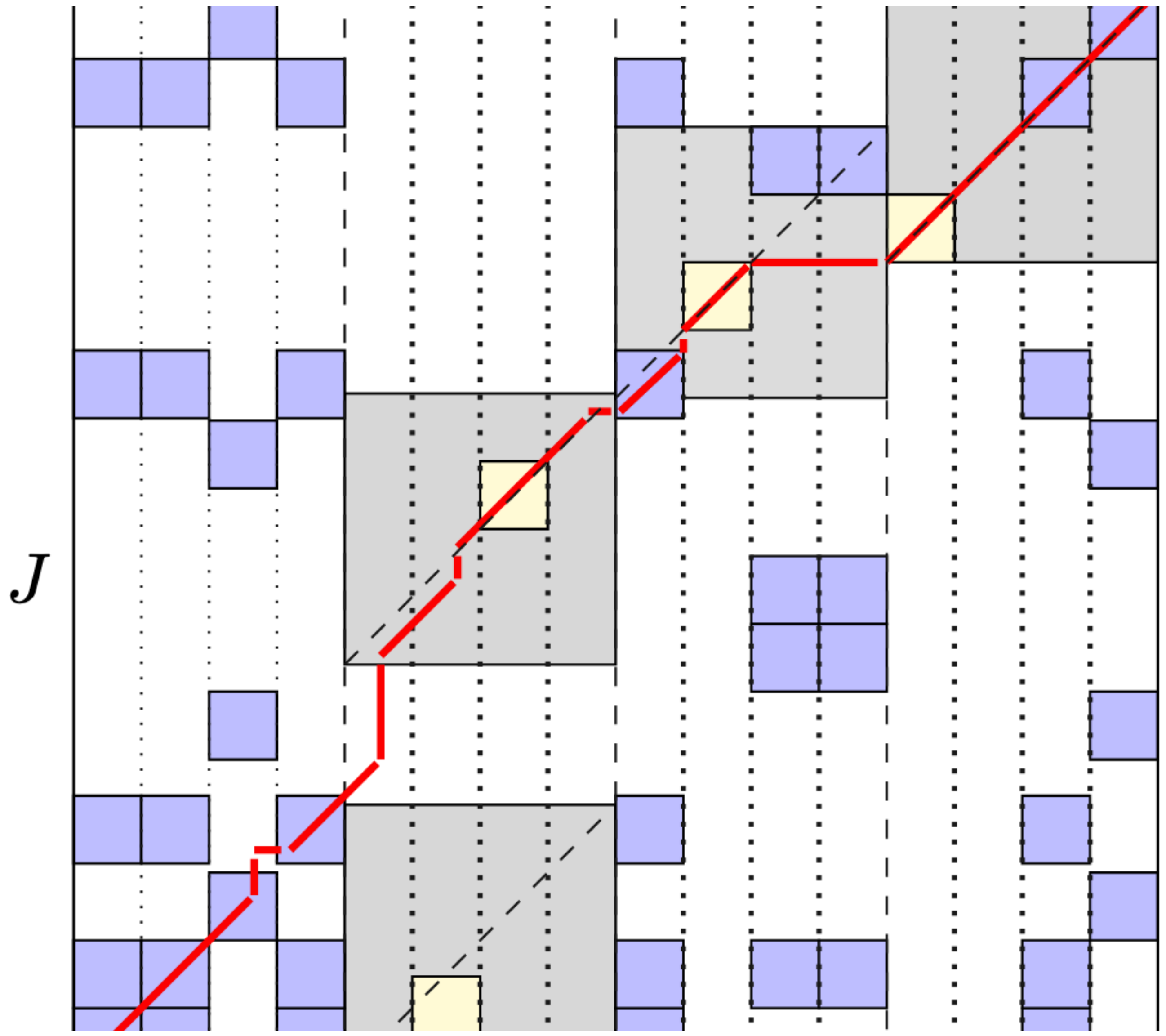
In this paper we provide evidence that the near-quadratic running time bounds known for the problem of computing edit distance might be tight. Specifically, we show that, if the edit distance can be computed in time $O(n^{2-\delta})$ for some constant $\delta > 0$, then the satisfiability of conjunctive normal form formulas with N variables and M clauses can be solved in time $M^{O(1)}2^{(1-\epsilon)N}$ for a constant $\epsilon > 0$. The latter result would violate the *Strong Exponential Time Hypothesis*, which postulates that such algorithms do not exist.

with many applications in computational biology, natural language processing and information theory. The problem of computing the edit distance between two strings is a classical computational task, with a well-known algorithm based on dynamic programming. Unfortunately, that algorithm runs in quadratic time, which is prohibitive for long sequences (e.g., the human genome consists of roughly 3 billions base pairs). A considerable effort has been invested into designing faster algorithms, either by assuming that the edit distance is bounded, by considering the average case or by resorting to approximation[1]. However, the fastest known exact algorithm, due to [MP80], has a running time of $O(n^2/\log^2 n)$ for sequences of length n, which is still nearly quadratic.

In this paper we provide evidence that the (near)-quadratic running time bounds known for this problem might, in fact, be tight. Specifically, we show that if the edit distance can be computed in time $O(n^{2-\delta})$ for some constant $\delta > 0$, then the satisfiability of conjunctive normal form (CNF) formulas with N variables and M clauses can be solved in time

Edit Distance Fun Facts

- 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

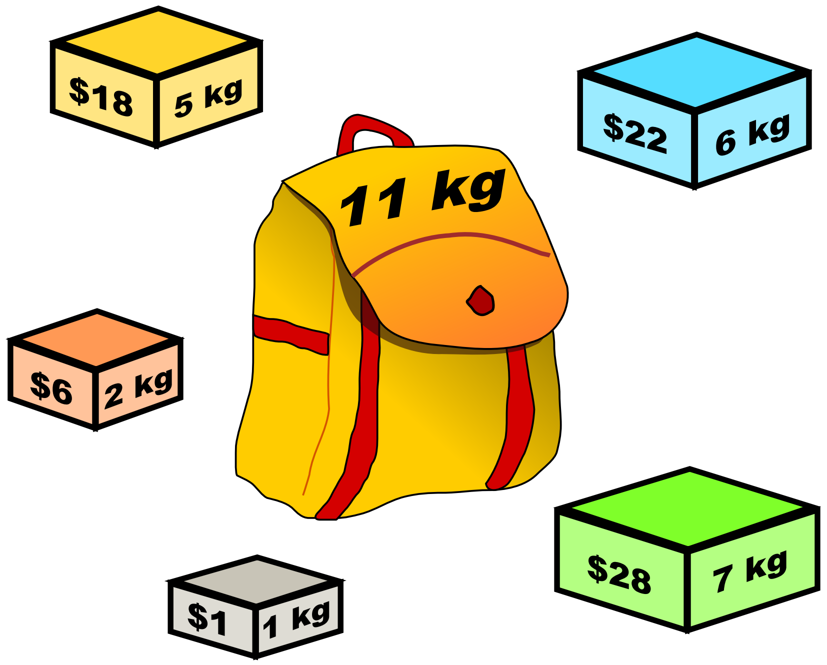
Reading: Chapter 6.4, KT

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)

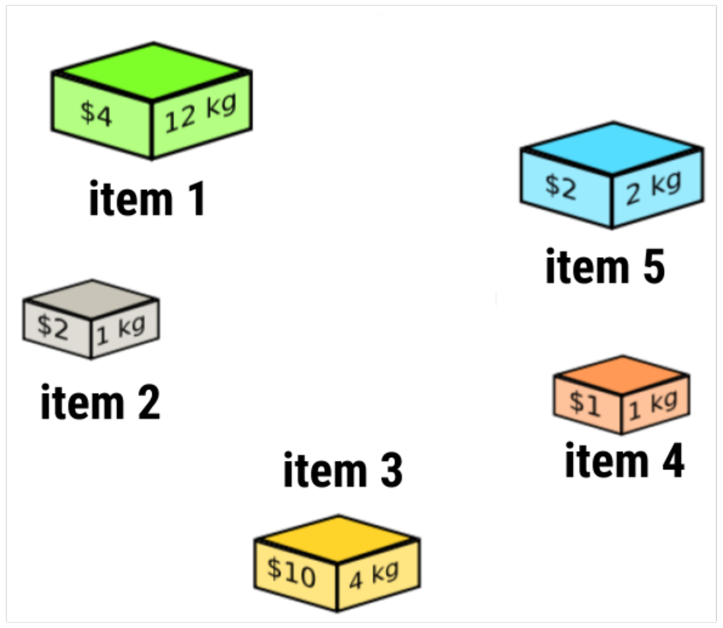


i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

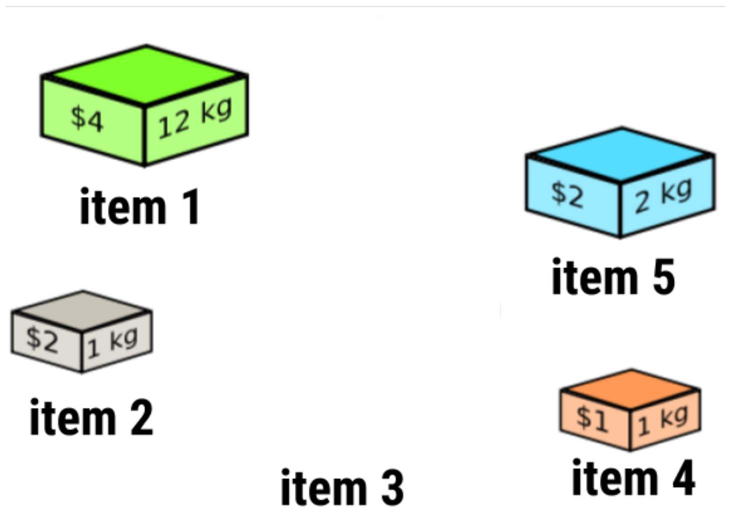
knapsack instance
(weight limit W = 11)

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

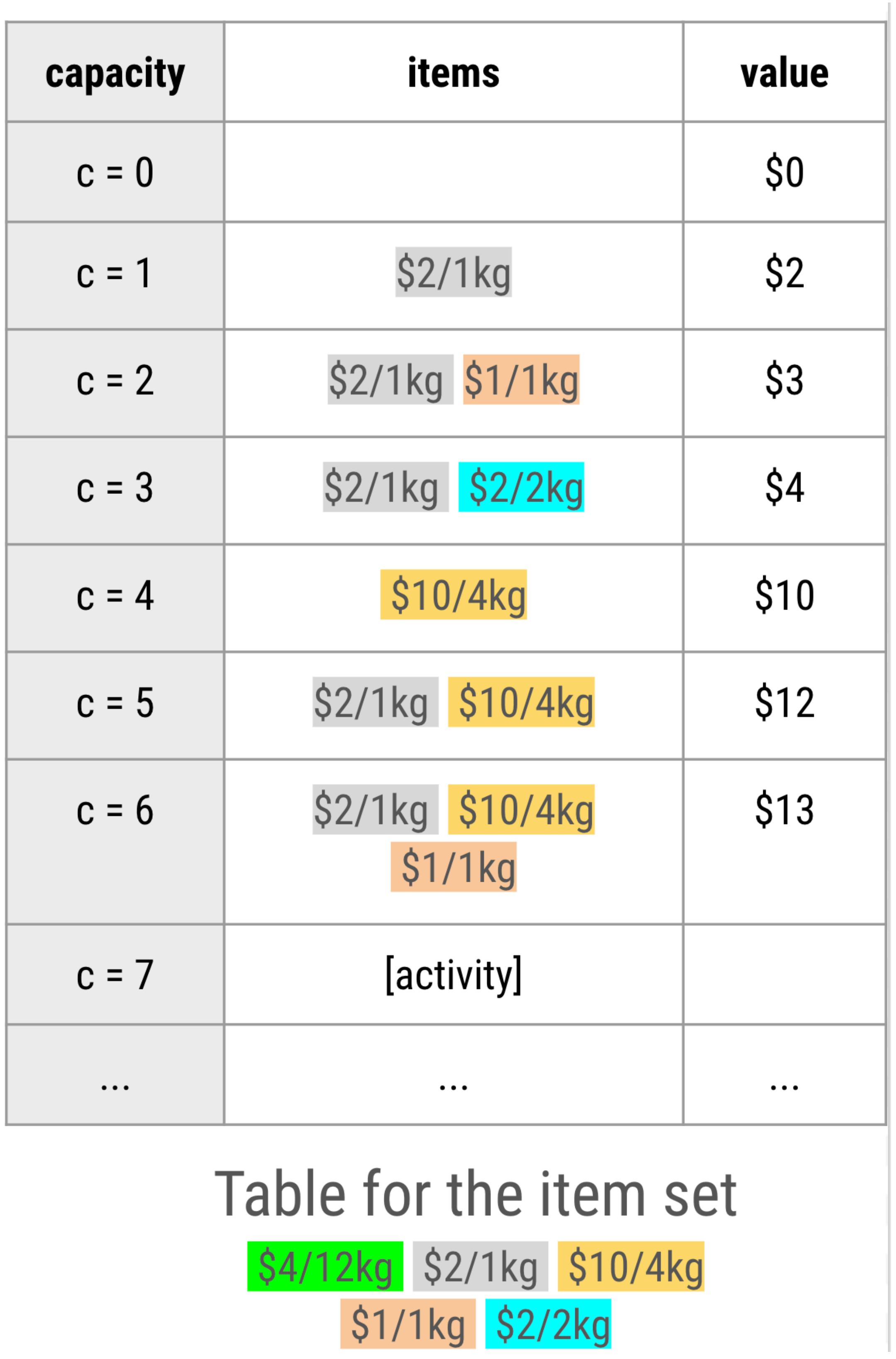


Partial Selection #2

- 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 $1 \leq j \leq n, \quad 1 \leq c \leq C$
- **Evaluation order?**
  - Row by row (i.e. item by item: for each item fill in each capacity one by one)
- **Final answer?** $\max\limits_{1 \leq c \leq C} T[n,c]$

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)