More DP: Text Segmentation and Edit Distance

Lecture 14

How to design DP algorithms

1. Find a “smart” recursion (The hard part)
    1. Formulate the sub-problem
    2. so that the number of distinct subproblems is small; polynomial in the original problem size.

How to design DP algorithms

1. Find a “smart” recursion (The hard part)
   1. Formulate the sub-problem
   2. so that the number of distinct subproblems is small; polynomial in the original problem size.

2. Memoization
   1. Identify distinct subproblems
   2. Choose a memoization data structure
   3. Identify dependencies and find a good evaluation order
   4. An iterative algorithm replacing recursive calls with array lookups

Part I

More Text Segmentation

A variation

Input A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function **IsStringinL**($string \quad x$) that decides whether x is in L, and non-negative integer k

Goal Decide if $w \in L^k$ using **IsStringinL**($string \quad x$) as a black box sub-routine

Example

Suppose L is *English* and we have a procedure to check whether a string/word is in the *English* dictionary.

- Is the string "isthisanenglishsentence" in $English^5$?
- Is the string "isthisanenglishsentence" in $English^4$?
- Is "asinineat" in $English^2$?
- Is "asinineat" in $English^4$?
- Is "zibzzzad" in $English^1$?

Recursive Solution

When is $w \in L^k$?

Recursive Solution

When is $w \in L^k$?
$k = 0$: $w \in L^k$ iff $w = \epsilon$
$k = 1$: $w \in L^k$ iff $w \in L$
$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L$ and $v \in L^{k-1}$

Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L$ and $v \in L^{k-1}$

Assume w is stored in array $A[1..n]$

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n - 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k - 1))
                Output YES

    Output NO
```

Analysis

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n − 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
                Output YES

    Output NO
```

- How many distinct sub-problems are generated by **IsStringinLk**$(A[1..n], k)$?

Analysis

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n − 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
                Output YES

    Output NO
```

- How many distinct sub-problems are generated by **IsStringinLk**$(A[1..n], k)$? $O(nk)$

Analysis

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n − 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
                Output YES

    Output NO
```

- How many distinct sub-problems are generated by **IsStringinLk**$(A[1..n], k)$? $O(nk)$
- How much space?

Analysis

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n − 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
                Output YES

    Output NO
```

- How many distinct sub-problems are generated by **IsStringinLk**$(A[1..n], k)$? $O(nk)$
- How much space? $O(nk)$

Analysis

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n − 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
                Output YES

    Output NO
```

- How many distinct sub-problems are generated by **IsStringinLk**$(A[1..n], k)$? $O(nk)$
- How much space? $O(nk)$
- Running time?

Analysis

```
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Ouput NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n − 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
                Output YES

    Output NO
```

- How many distinct sub-problems are generated by $\mathbf{IsStringinLk}(A[1..n], k)$? $O(nk)$
- How much space? $O(nk)$
- Running time? $O(n^2k)$

Naming subproblems and recursive equation

ISLk(i, h): a boolean which is 1 if $A[i..n]$ is in L^h, 0 otherwise

Base case: **ISLk**$(n+1, 0) = 1$ interpreting $A[n+1..n]$ as ϵ

Naming subproblems and recursive equation

ISLk(i, h): a boolean which is 1 if $A[i..n]$ is in L^h, 0 otherwise

Base case: **ISLk**$(n+1, 0) = 1$ interpreting $A[n+1..n]$ as ϵ

Recursive relation:

- **ISLk**$(i, h) = 1$ if $\exists i < j \leq n+1$ such that (**ISLk**$(j, h-1) = 1$ and **IsStringinL**$(A[i..(j-1]) = 1)$
- **ISLk**$(i, h) = 0$ otherwise

Alternately:
ISLk$(i, h) = \max_{i<j\leq n+1}$ **ISLk**$(j, h-1)$**IsStringinL**$(A[i..(j-1)]))$

Output: **ISLk**$(1, k)$

How to order bottom up computation?

	0	1	2	3	h			K
1	0							
2	0							
3	0							
4	0							
	0							
	0							
n	0							
n+1	1	0	0	0	0	0	0	0

i

Iterative Algorithm

```
IsStringinLstar-Iterative(A[1..n]):
    boolean ISLk[1..(n + 1), 0...k]
    ISLk[n + 1, 0] = TRUE
    for (i = 1 to n)
        ISLk[i, 0] = FALSE

    for (h = 1 to k)
        for (i = 1 to n)
            ISLk[i, h] = FALSE
            for (j = i + 1 to n + 1)
                    If (ISLk[j, h − 1] and IsStringinL(A[i..j − 1]))
                        ISLk[i, h] = TRUE
                        Break

    If (ISLk[1, k] = 1) Output YES
    Else Output NO
```

Running time: $O(n^2 k)$. **Space:** $O(nk)$

Another variant

Question: What if we want to check if $w \in L^i$ for some $0 \le i \le k$? That is, is $w \in \cup_{i=0}^{k} L^i$?

Part II

Edit Distance and Sequence Alignment

Spell Checking Problem

Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?

Spell Checking Problem

Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?

What does nearness mean?

Question: Given two strings $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a *distance* between them?

Spell Checking Problem

Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?

What does nearness mean?

Question: Given two strings $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a *distance* between them?

Edit Distance: minimum number of "edits" to transform x into y.

Edit Distance

Definition

Edit distance between two words X and Y is the number of letter insertions, letter deletions and letter substitutions required to obtain Y from X.

Example

The edit distance between FOOD and MONEY is at most **4**:

$$\mathbf{\underline{F}OOD} \rightarrow \mathbf{MO\underline{O}D} \rightarrow \mathbf{MON\underline{O}D} \rightarrow \mathbf{MONE\underline{D}} \rightarrow \mathbf{MONEY}$$

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F	**O**	**O**		**D**
M	**O**	**N**	**E**	**Y**

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F	**O**	**O**		**D**
M	**O**	**N**	**E**	**Y**

Formally, an alignment is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1,1), (2,2), (3,3), (4,5)\}$.
Cost of an alignment is the number of columns that do not contain the same letter twice.

Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

Applications

1. Spell-checkers and Dictionaries
2. Unix `diff`
3. DNA sequence alignment . . . but, we need a new metric

Similarity Metric

Definition

For two strings X and Y, the cost of alignment M is

1. [Gap penalty] For each gap in the alignment, we incur a cost δ.
2. [Mismatch cost] For each pair p and q that have been matched in M, we incur cost α_{pq}; typically $\alpha_{pp} = 0$.

Similarity Metric

Definition

For two strings X and Y, the cost of alignment M is

1. [Gap penalty] For each gap in the alignment, we incur a cost δ.
2. [Mismatch cost] For each pair p and q that have been matched in M, we incur cost α_{pq}; typically $\alpha_{pp} = 0$.

Edit distance is special case when $\delta = \alpha_{pq} = 1$.

An Example

Example

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

Cost $= \delta + \alpha_{ae}$

Alternative:

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

Cost $= 3\delta$

Or a really stupid solution (delete string, insert other string):

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

Cost $= 19\delta$.

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost **1** unit?

$$\boxed{374}$$
$$\boxed{473}$$

- 1
- 2
- 3
- 4
- 5

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost **1** unit?

$\boxed{373}$

$\boxed{473}$

1. 1
2. 2
3. 3
4. 4
5. 5

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost **1** unit?

$\boxed{37}$

$\boxed{473}$

- 1
- 2
- 3
- 4
- 5

Sequence Alignment

Input Given two words X and Y, and gap penalty δ and mismatch costs α_{pq}

Goal Find alignment of minimum cost

Edit distance

Basic observation

Let $X = \alpha x$ and $Y = \beta y$
α, β: strings.
x and y single characters.
Think about optimal edit distance between X and Y as alignment, and consider last column of alignment of the two strings:

α	x
β	y

or

α	x
βy	

or

αx	
β	y

Observation

Prefixes must have optimal alignment!

Try all possibilities

Observation

Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$. If (m, n) are not matched then either the mth position of X remains unmatched or the nth position of Y remains unmatched.

1. Case x_m and y_n are matched.
    1. Pay mismatch cost $\alpha_{x_m y_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$
2. Case x_m is unmatched.
    1. Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$
3. Case y_n is unmatched.
    1. Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$

Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$
Array $COST$ stores cost of matching two chars. Thus $COST[a, b]$ give the cost of matching character a to character b.

$EDIST(A[1..m], B[1..n])$
    If $(m = 0)$ return $n\delta$
    If $(n = 0)$ return $m\delta$
    $m_1 = \delta + EDIST(A[1..(m-1)], B[1..n])$
    $m_2 = \delta + EDIST(A[1..m], B[1..(n-1)]))$
    $m_3 = COST[A[m], B[n]] + EDIST(A[1..(m-1)], B[1..(n-1)])$
    return $\min(m_1, m_2, m_3)$

Recursive Algorithm

$EDIST(A[1..m], B[1..n])$
    If $(m = 0)$ return $n\delta$
    If $(n = 0)$ return $m\delta$
    $m_1 = \delta + EDIST(A[1..(m-1)], B[1..n])$
    $m_2 = \delta + EDIST(A[1..m], B[1..(n-1)]))$
    $m_3 = COST[A[m], B[n]] + EDIST(A[1..(m-1)], B[1..(n-1)])$
    return $\min(m_1, m_2, m_3)$

- How many distinct sub-problems will $EDIST(A[1..m], B[1..n])$ generate?

Recursive Algorithm

$EDIST(A[1..m], B[1..n])$
```
If (m = 0) return nδ
If (n = 0) return mδ
```
$m_1 = \delta + EDIST(A[1..(m-1)], B[1..n])$
$m_2 = \delta + EDIST(A[1..m], B[1..(n-1)]))$
$m_3 = COST[A[m], B[n]] + EDIST(A[1..(m-1)], B[1..(n-1)])$
return $\min(m_1, m_2, m_3)$

- How many distinct sub-problems will $EDIST(A[1..m], B[1..n])$ generate? $O(nm)$

Recursive Algorithm

$EDIST(A[1..m], B[1..n])$
    If $(m = 0)$ return $n\delta$
    If $(n = 0)$ return $m\delta$
    $m_1 = \delta + EDIST(A[1..(m-1)], B[1..n])$
    $m_2 = \delta + EDIST(A[1..m], B[1..(n-1)])$
    $m_3 = COST[A[m], B[n]] + EDIST(A[1..(m-1)], B[1..(n-1)])$
    return $\min(m_1, m_2, m_3)$

- How many distinct sub-problems will $EDIST(A[1..m], B[1..n])$ generate? $O(nm)$
- What is the running time if we memoize recursion?

Recursive Algorithm

$EDIST(A[1..m], B[1..n])$
```
    If (m = 0) return nδ
    If (n = 0) return mδ
    m1 = δ + EDIST(A[1..(m − 1)], B[1..n])
    m2 = δ + EDIST(A[1..m], B[1..(n − 1)]))
    m3 = COST[A[m], B[n]] + EDIST(A[1..(m − 1)], B[1..(n − 1)])
    return min(m1, m2, m3)
```

- How many distinct sub-problems will $EDIST(A[1..m], B[1..n])$ generate? $O(nm)$
- What is the running time if we memoize recursion? $O(nm)$ since each call takes $O(1)$ time to assemble the answers from to recursive calls and no other computation.

Recursive Algorithm

$EDIST(A[1..m], B[1..n])$
    If $(m = 0)$ return $n\delta$
    If $(n = 0)$ return $m\delta$
    $m_1 = \delta + EDIST(A[1..(m-1)], B[1..n])$
    $m_2 = \delta + EDIST(A[1..m], B[1..(n-1)]))$
    $m_3 = COST[A[m], B[n]] + EDIST(A[1..(m-1)], B[1..(n-1)])$
    return $\min(m_1, m_2, m_3)$

- How many distinct sub-problems will $EDIST(A[1..m], B[1..n])$ generate? $O(nm)$
- What is the running time if we memoize recursion? $O(nm)$ since each call takes $O(1)$ time to assemble the answers from to recursive calls and no other computation.
- How much space for memoization?

Recursive Algorithm

$\boldsymbol{EDIST(A[1..m], B[1..n])}$

```
    If (m = 0) return nδ
    If (n = 0) return mδ
    m1 = δ + EDIST(A[1..(m − 1)], B[1..n])
    m2 = δ + EDIST(A[1..m], B[1..(n − 1)]))
    m3 = COST[A[m], B[n]] + EDIST(A[1..(m − 1)], B[1..(n − 1)])
    return min(m1, m2, m3)
```

- How many distinct sub-problems will $EDIST(A[1..m], B[1..n])$ generate? $O(nm)$
- What is the running time if we memoize recursion? $O(nm)$ since each call takes $O(1)$ time to assemble the answers from to recursive calls and no other computation.
- How much space for memoization? $O(nm)$

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(nm)$ we name them to help us understand the structure better.

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(nm)$ we name them to help us understand the structure better.

Optimal Costs

Let $\mathrm{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\mathrm{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1, j-1), \\ \delta + \mathrm{Opt}(i-1, j), \\ \delta + \mathrm{Opt}(i, j-1) \end{cases}$$

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(nm)$ we name them to help us understand the structure better.

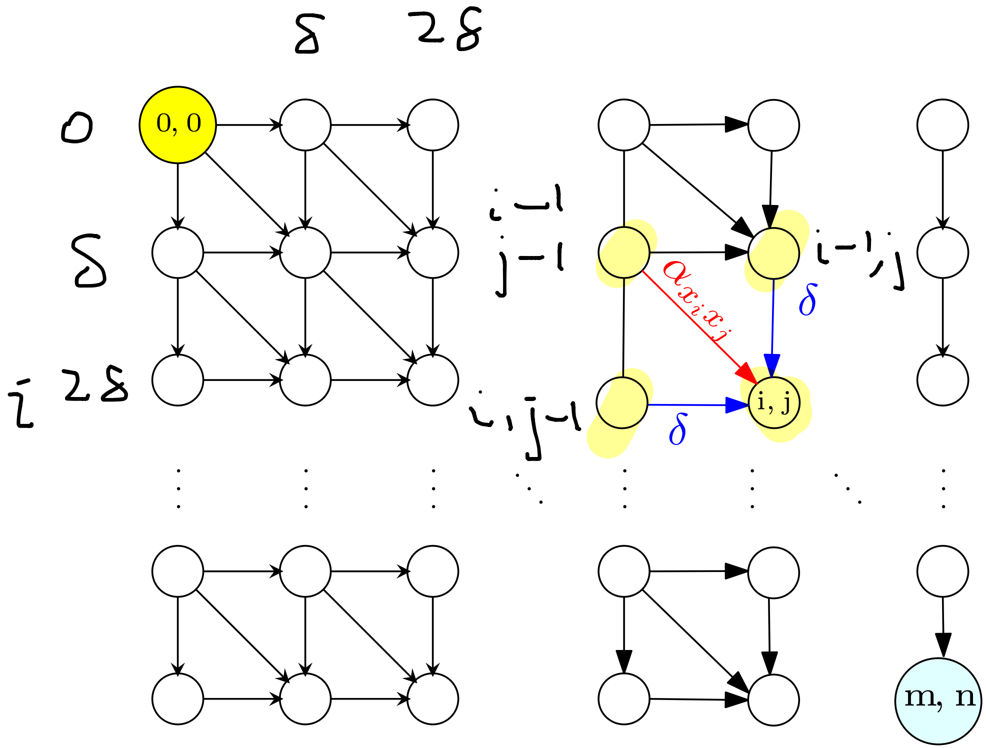
Optimal Costs

Let $\mathrm{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\mathrm{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1, j-1), \\ \delta + \mathrm{Opt}(i-1, j), \\ \delta + \mathrm{Opt}(i, j-1) \end{cases}$$

Base Cases: $\mathrm{Opt}(i, 0) = \delta \cdot i$ and $\mathrm{Opt}(0, j) = \delta \cdot j$

How to order bottom up computation?



Base case: $\mathrm{Opt}(i, 0) = \delta \cdot i$ and $\mathrm{Opt}(0, j) = \delta \cdot j$
Recursive relation: Fill in row by row (or column by column)

Removing Recursion to obtain Iterative Algorithm

```
int M[0..m][0..n]
Initialize all entries of M[i][j] to ∞
return EDIST(A[1..m], B[1..n])
```

$EDIST(A[1..m], B[1..n])$

 int $M[0..m][0..n]$

 for $i = 1$ to m **do** $M[i,0] = i\delta$

 for $j = 1$ to n **do** $M[0,j] = j\delta$

 for $i = 1$ to m **do**

  for $j = 1$ to n **do**

$$M[i][j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

Removing Recursion to obtain Iterative Algorithm

```
int M[0..m][0..n]
Initialize all entries of M[i][j] to ∞
return EDIST(A[1..m], B[1..n])
```

EDIST$(A[1..m], B[1..n])$

 int $M[0..m][0..n]$

 for $i = 1$ to m **do** $M[i, 0] = i\delta$

 for $j = 1$ to n **do** $M[0, j] = j\delta$

 for $i = 1$ to m **do**

  for $j = 1$ to n **do**

$$M[i][j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

Running time: $O(nm)$

Space: $O(nm)$

Sequence Alignment in Practice

1. Typically the DNA sequences that are aligned are about $\mathbf{10^5}$ letters long!
2. So about $\mathbf{10^{10}}$ operations and $\mathbf{10^{10}}$ bytes needed
3. The killer is the 10GB storage
4. Can we reduce space requirements?

Optimizing Space

1. Recall

$$M(i,j) = \min \begin{cases} \alpha_{x_i y_j} + M(i-1, j-1), \\ \delta + M(i-1, j), \\ \delta + M(i, j-1) \end{cases}$$

2. Entries in jth column only depend on $(j-1)$st column and earlier entries in jth column
3. Only store the current column and the previous column reusing space; $N(i,0)$ stores $M(i,j-1)$ and $N(i,1)$ stores $M(i,j)$

Computing in column order to save space

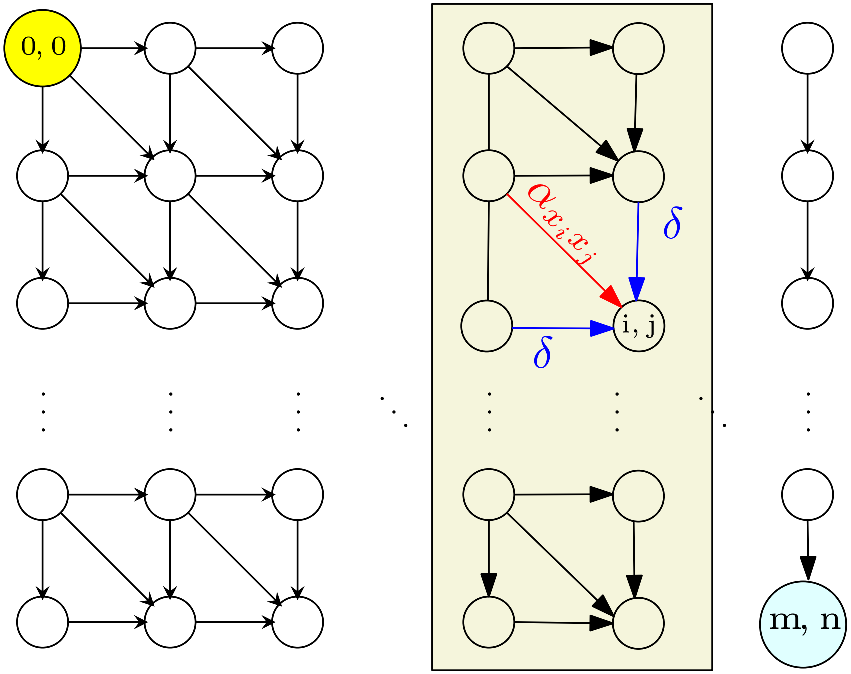


Figure: $\boldsymbol{M(i,j)}$ only depends on previous column values. Keep only two columns and compute in column order.

Space Efficient Algorithm

for all i **do** $N[i,0] = i\delta$
for $j = 1$ to n **do**
    $N[0,1] = j\delta$ (* corresponds to $M(0,j)$ *)
    for $i = 1$ to m **do**

$$N[i,1] = \min \begin{cases} \alpha_{x_i y_j} + N[i-1,0] \\ \delta + N[i-1,1] \\ \delta + N[i,0] \end{cases}$$

    for $i = 1$ to m **do**
        Copy $N[i,0] = N[i,1]$

Analysis

Running time is $O(mn)$ and space used is $O(2m) = O(m)$

Which data structure?

So far our memoization uses multi-dimensional arrays:

- Fibonacci numbers, 1-D array
- Text segmentation, suffix, 1-D array
- Longest increasing subsequence, suffix+index, 2-D array
- Edit distance, two prefixes, 2-D array

Which data structure?

So far our memoization uses multi-dimensional arrays:

- Fibonacci numbers, 1-D array
- Text segmentation, suffix, 1-D array
- Longest increasing subsequence, suffix+index, 2-D array
- Edit distance, two prefixes, 2-D array

Not always true.