Lecture 9: Dynamic programming

Algorithm

Jeong-Hun Kim

Table of Contents

Part 1

WARMING UP: FIBONACCI NUMBER

Warming up: Fibonacci number

❖ Fibonacci number

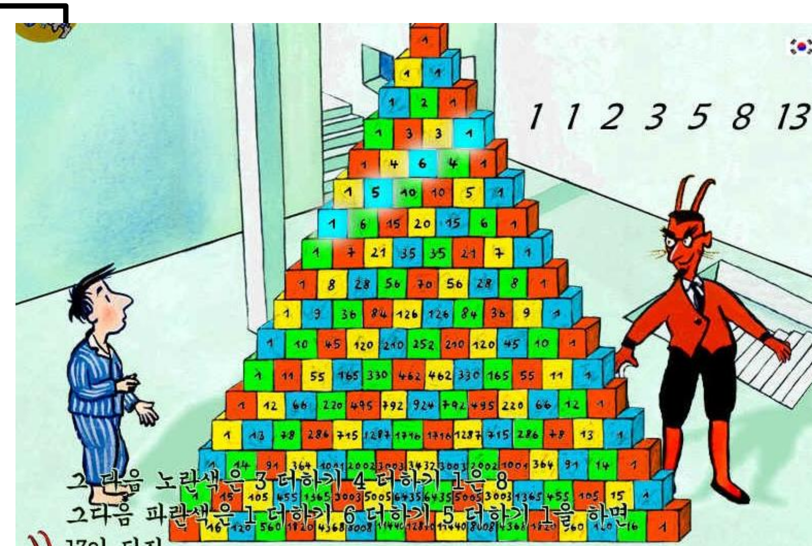
- For a natural number $\mathbb{N}$ that includes 0, the n-th Fibonacci number F_i is defined as follows:

$F_0 = 0, F_1 = 1$ (base case)

$F_n = F_{n-1} + F_{n-2}$ (inductive step)

Ex) F_5

$$
\begin{aligned}
F_5 &= F_4 + F_3 \\
&= (F_3 + F_2) + (F_2 + F_1) \\
&= (F_2 + F_1) + (F_1 + F_0) + (F_1 + F_0) + 1 \\
&= (F_1 + F_0) + 1 + (1 + 0) + (1 + 0) + 1 \\
&= (1 + 0) + 1 + 1 + 1 + 1 = 5
\end{aligned}
$$



https://cafe.naver.com/educd.cafe

Warming up: Fibonacci number

❖ Pseudo-code using recursion

```
Fib(n):
  if (n==0)
    return 0
  if (n==1)
    return 1
  return Fib(n-1) + Fib(n-2)   ←――――   Recursion
```

- What is the time complexity of the above algorithm?
  - Let $T(n)$ be the total number of additions needed to obtain the answer for Fib(n)
  - Recurrence relation:

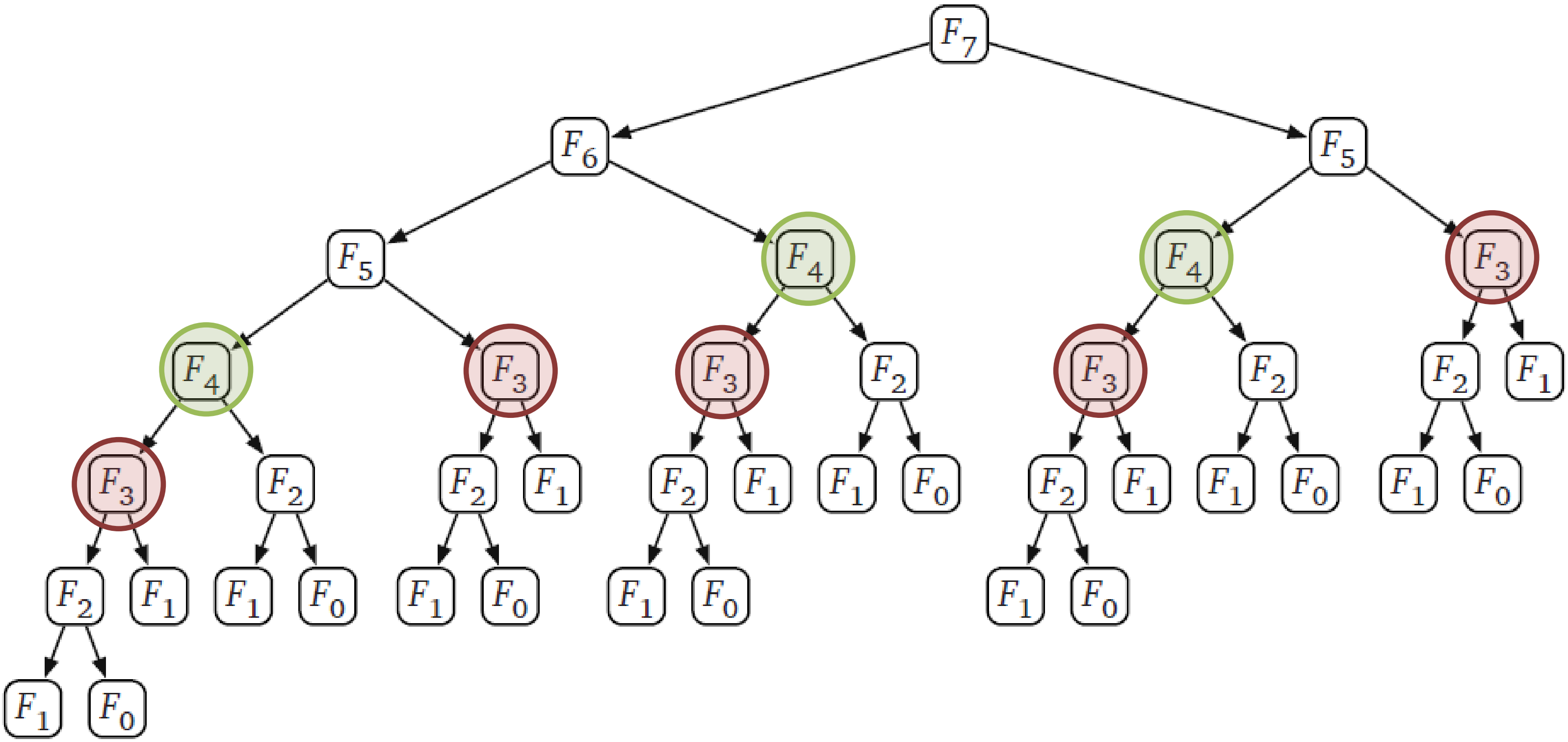
$$\boxed{T(n) = T(n-1) + T(n-2) + 1,\ T(0)=T(1)=0}$$

    – $T(n) \sim (1.618)^n$

    – What is the reason for this slowness?

Warming up: Fibonacci number

❖ Example) when calling Fib(7), the recursion tree is as follows:



- Problem definition:
  - Calling same functions multiple times while processing through recursive calls

Warming up: Fibonacci number

❖ Solution)

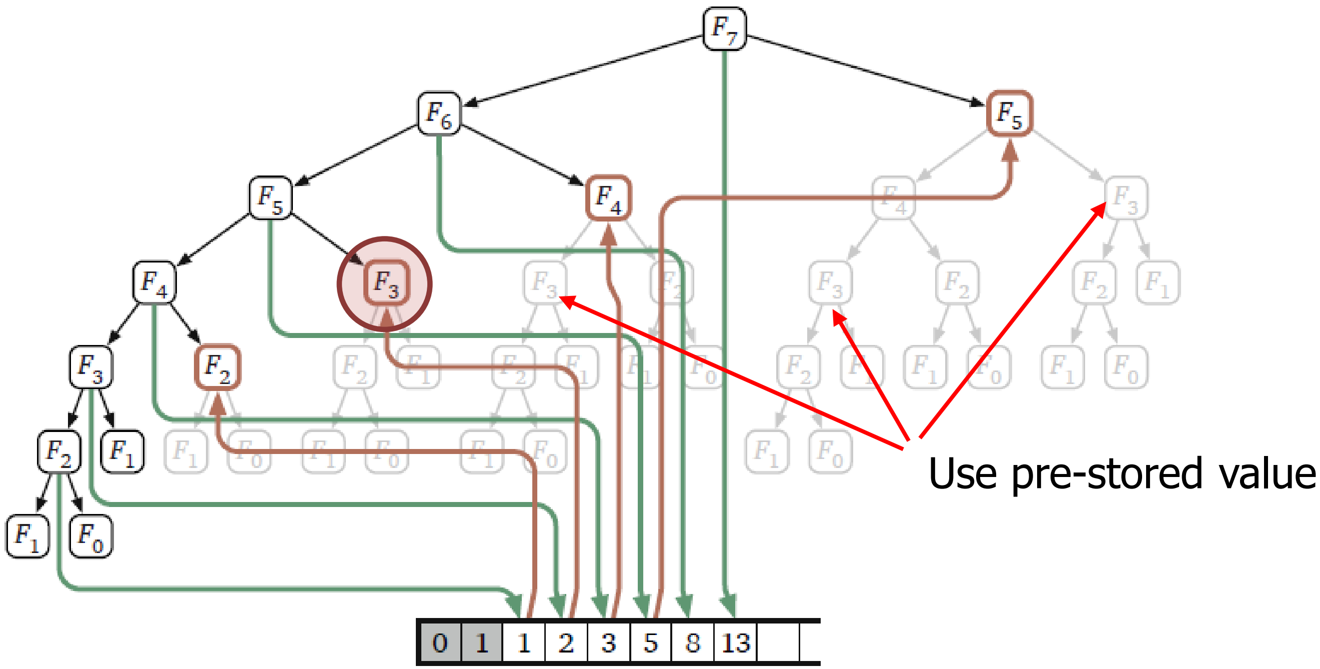
- Storing the results each time a recursive call is made and later reusing the stored results when the same calls are requested

```
global F[0..n] = {0, 1, null, …, null};
Fib(n):
  if (n==0)
    return 0
  if (n==1)
    return 1
  if F[n] == null
    F[n] = Fib(n-1) + Fib(n-2)
  return F[n]
```

- The time complexity of the above algorithm is?
  - The number of additions is O(n)
  - Addition is performed only once when F[n] is null
  - The number of storage operation is also O(n)
  - Thus, the time complexity is O(n)

Warming up: Fibonacci number

❖ The recursion tree for the improved algorithm:



- Memoization: technique of storing intermediate results in recursive calls
- Dynamic programming (DP)
  - A paradigm for solving problems using recursion and memoization

Part 2

DYNAMIC PROGRAMMING

Dynamic Programming

- Dynamic programming (DP)
  - Recursion + Memoization

- Solving problems using DP
  1. Precisely defining the problem
  2. Defining subproblems for the defined problem
     - In Fibonacci number, computing $F_{n'}$ is a subproblem ($n' < n$)
  3. Designing **a recurrence relation** using subproblems
  4. Solving the recurrence relation using **memorization**
     - Selecting a data structure for storing intermediate results
     - Checking the dependency between subproblems
     - Determining the order in which to solve subproblems based on their dependencies
  5. Analyzing time complexity and implementing an algorithm

Part 3

LONGEST INCREASING SUBSEQUENCE

Longest Increasing Subsequence

- Problem details
  - When given a sequence S of length n, where S = $a_1, a_2, \ldots, a_n$, a sequence S′ = $a_{i1}, a_{i2}, \ldots, a_{ik}$ is considered a subsequence of S if it satisfies $1 \leq i1 < i2 < \ldots < ik \leq n$

2	**4**	7	11	**3**	**4**	6	**8**	9	**Sequence S**

4	**3**	**4**	**8**	**Subsequence of S**

  - When $a_1 < a_2 < \ldots < a_n$, the sequence S is called an **increasing sequence**
  - When $a_1 \leq a_2 \leq \ldots \leq a_n$, the sequence S is referred to as a **non-decreasing sequence**

Longest Increasing Subsequence

- Longest increasing subsequence (LIS)
  - Input: sequence S = $a_1, a_2, \ldots, a_n$
  - Problem:
    - Find the length of the longest increasing subsequence (LIS) in sequence S

5 2 8 6 3 6 9 7 Sequence S

5 8 9 Increasing subsequence S

2 3 6 9 LIS of S (length = 4)

Longest Increasing Subsequence

- Problem: finding the length of the LIS in the sequence S
- First recursive approach
  - Let L(i) be defined as the length of the LIS formed by the first i elements of S, which are $a_1, a_2, \ldots, a_i$
  - Can L(i) be expressed in terms of L(1), L(2), …, L(i-1)?
  - **Case 1)** the increasing subsequence S_i of length L(i) does not include a_i
    - L(i) = L(i-1)
  - **Case 2)** Assume that S_i includes a_i
    - If a_j is the element immediately preceding a_i, then $a_j < a_i$
    - Since j cannot be determine directly, all possibilities need to be considered

$$\boxed{L(i) = 1 + \max_{j<i,\ a_j<a_i} L(j)}$$

It is not guaranteed that the LIS of length L(j) includes a_j

Longest Increasing Subsequence

- Problem: finding the length of the LIS in the sequence S (cont'd)
  - Defining subproblems correctly
    - Let L(i) be defined as the length of the LIS ending with a_i in the sequence formed by the first i elements of S, which are $a_1, a_2, \ldots, a_i$
    - Recurrence relation: $\boxed{L(i) = 1 + \max_{j<i,\ a_j<a_i} L(j)}$
    - The length of LIS of S is determined by $\max_{i=1}^{n} L(i)$ ($\neq L(n)$)
    - Number of subproblems = L(1), L(2), …, L(n) → n

Longest Increasing Subsequence

❖ Problem: finding the length of the LIS in the sequence S (cont'd)

$$L(i) = 1 + \max_{j<i,\ a_j<a_i} L(j)$$

- Pseudo-code

```
for i = 1 to n do
   L[i] = 1
   for j = 1 to i – 1 do
      if a_j < a_i and 1 + L[j] > L[i]
         L[i] = 1 + L[j]

Output max_{i=1}^{n} L[i]
```

- To compute the subproblem L(i), it need to know the values of L(1), …, L(i-1), so it start with L(1) and proceed to solve them from left to right

Longest Increasing Subsequence

❖ Example)

$$L(i) = 1 + \max_{j<i,\ a_j<a_i} L(j)$$

(1)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

(2)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Since there is no j < 2 that satisfies the given condition, L(2)=1

(3)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**2**	**1**	**1**	**1**	**1**	**1**

Since there is j = 1 or 2 that satisfies the given condition,
L(3) = L(1) + 1 = 2

(4)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**2**	**2**	**1**	**1**	**1**	**1**

Since there is a j = 1 or 2 that satisfies
The given condition, L(4) = L(1) + 1 = 2

Longest Increasing Subsequence

❖ Example)

$$L(i) = 1 + \max_{j<i,\ a_j<a_i} L(j)$$

(5)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**2**	**2**	**2**	**1**	**1**	**1**

Since there is a j=2 that satisfies the given condition,
L(5) = L(2) + 1 = 2

(6)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**2**	**2**	**2**	**3**	**1**	**1**

Since there is a j=5 that satisfies the given condition,
L(6) = L(5) + 1= 3

(7)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**2**	**2**	**2**	**3**	**4**	**1**

Since there is a j=6 that satisfies the given condition,
L(7) = L(6) + 1 = 4

(8)

	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**2**	**2**	**2**	**3**	**4**	**4**

Since there is a j=6 that satisfies the given condition,
L(7) = L(6) + 1 = 4

Longest Increasing Subsequence

❖ Example)

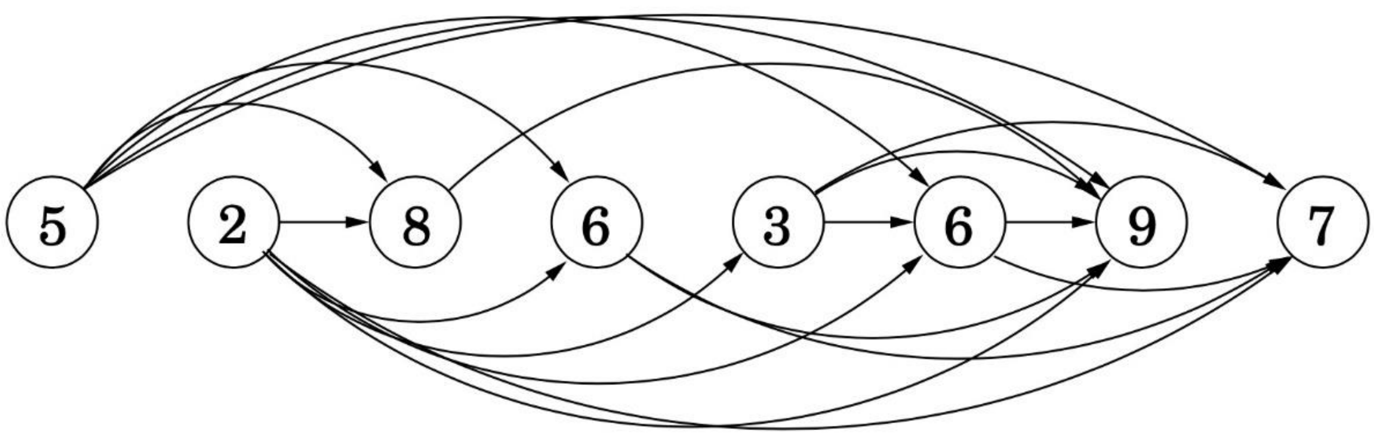
	1	2	3	4	5	6	7	8
S	**5**	**2**	**8**	**6**	**3**	**6**	**9**	**7**
L(i)	**1**	**1**	**2**	**2**	**2**	**3**	**4**	**4**
pre	**0**	**0**	**1**	**1**	**2**	**5**	**6**	**6**

- The length of LIS = L(7) = L(8) = 4
  - In practice, when updating L(i) = L(j) + 1, it can also separately store that the element immediately preceding the LIS ending with a_i is a_j in an array (pre)

- Time complexity
  - When calculating L(i), search for the appropriate L(j) that satisfies the condition → O(i-1)
    - Total: $O(1+2+3+\ldots+n-1) = O(n^2)$
  - Find the largest L(i) → O(n)
  - Total complexity: $O(n^2)$

Longest Increasing Subsequence

- Another approach
  - Let's define a directed graph G=(V, E) for the sequence S = $a_1, a_2, \ldots, a_n$ as follows:
    - V = $\{a_1, a_2, \ldots, a_n\}$
    - If there exist i and j s.t. $i < j$ and $a_i < a_j$, add the edge(i, j) to E



In all cases, G is a directed acyclic graph (DAG)

Longest Increasing Subsequence

- Another approach (cont'd)
  - The length of LIS = the length of the longest path in G
  - Since G is a DAG, its longest path can be computed using the following recurrence relation:
    - L(j) = the length of the longest path ending at vertex aj

$$\boxed{L(j) = 1 + \max_{(i,j)\in E} L(i)}$$

    - The length of LIS = $\max_{i=1}^{n} L(i)$

Longest Increasing Subsequence

- Another approach (cont'd)
  - Pseudo-code

```
for j = 1, 2, …, n:
   L(j) = 1 + max{L(i): (i, j) ∈ E}
return max_j L(j)
```

  - Time complexity
    - Generate a graph G (each vertex v_i can have a maximum of n-i edges) → $O(n-1 + n-2 + … + 1) = O(n^2)$
    - Calculate and store L(1), L(2), …, L(n) in order → O(m)
    - Find the maximum value among L(1), L(2), …, L(n), denoted as L(i) → O(n)
    - Total complexity = $O(n^2 + m)$

Part 4

EDIT DISTANCE

Edit Distance

❖ Motivation

- How to define an word that are ‘**similar**’ or ‘**close**’ to a given word
- Example)

Words that are ‘similar’ or ‘close’ to ‘genanrlty’

Edit Distance

- Edit distance
  - A measure indicating how ‘**similar**’ or ‘**close**’ two strings are
  - Edit distance between strings S1 and S2 is defined as the minimum number of operations required to transform S1 into S2 using the following three types of operations:
    - **Insertion:** add one symbol to S1 (position doesn’t matter)
      - ex) MONDT → MONEDT
    - **Deletion:** remove one symbol from S1 (position doesn’t matter)
      - ex) MONEDT → MONED
    - **Substitution:** change one symbol in S1 to another symbol (position doesn’t matter)
      - ex) MONED → MONEY

Edit Distance

- Example) S1 = SNOWY, S2 = SUNNY
  - Method 1
    - SNOWY → SSNOWY (insert S) → SSNOWNY (insert N) → SSNWNY (delete O) → SSNNY (delete W) → SUNNY (substitute S to U)
    - **Distance = 5**
  - Method 2
    - SNOWY → SUNOWY (insert U) → SUNOY (delete W) → SUNNY (substitute O to N)
    - **Distance = 3**
  - There is no way to transform "SNOWY" into "SUNNY" using fewer than 3 operations, so the edit distance between "SNOWY" and "SUNNY" is 3

Edit Distance

❖ Problem) calculate the edit distance between S1 and S2

- Gap table

S1	**S**	**-**	**N**	**O**	**W**	**Y**
S2	**S**	**U**	**N**	**N**	**-**	**Y**

Case 2 Case 3 Case 1

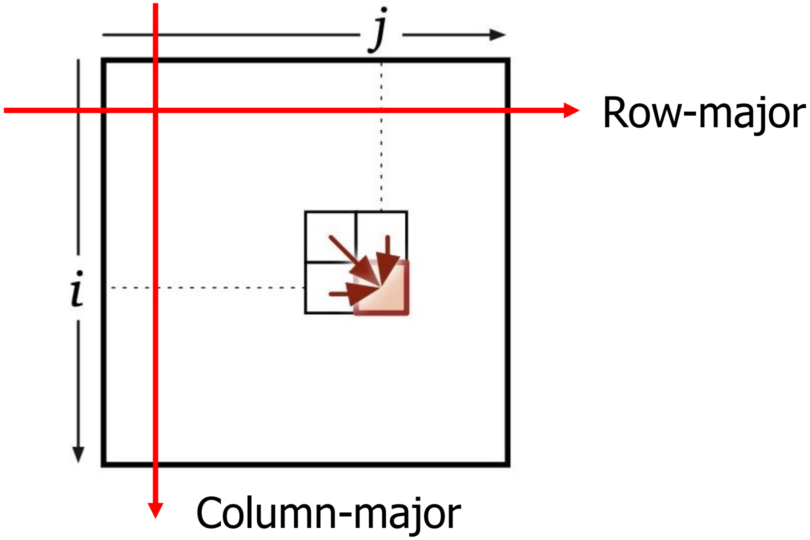
- Case 1: a column filled with S1 that differs from S2
  - Deletion in S1
- Case 2: S2 is filled, and S1 is empty in the column
  - Insertion in S1
- Case 3: Both S1 and S2 are filled, and they have different columns
  - Substitution of a symbol in S1

- Edit distance between S1 and S2
  - Case 1 + Case 2 + Case 3 = Number of mismatched columns

Edit Distance

❖ Problem) calculate the edit distance between S1 and S2 (cont'd)

- Memoization for edit(n, m)
  - Memoization structure: use a 2d array edit[0..n, 0..m]
  - Dependency:
    - edit[i, j] depends on edit[i-1, j], edit[i, j-1], and edit[i-1, j-1]
  - Computation order:
    - Row-major order or column-major order



Edit Distance

❖ Problem) calculate the edit distance between S1 and S2 (cont'd)

S1	**S**	**-**	**N**	**O**	**W**	**Y**
S2	**S**	**U**	**N**	**N**	**-**	**Y**

Edit distance = 3

- Key observation
  - Table obtained by removing the last column from the gap table of S1 and S2 represents the edit distance between the corresponding prefixes of S1 and S2

- Why?
  - Edit distance between S1 and S2 can be determined by the edit distance between the prefixes of S1 and S2

Edit Distance

❖ Problem) calculate the edit distance between S1 and S2 (cont'd)

- Let edit[i, j] be the edit distance between two prefixes S1[1, …, i] and S2[1, …, j]
- Based on the operation that occurs in the last column, it can design the following recurrence relation
- **Case 1:** when deletion occurs in the last column

$$\mathbf{edit[i, j] = edit[i, j\text{-}1] + 1}$$

Edit [i, j-1]

S1	**S**	**-**	**N**	**O**	**W**
S2	**S**	**U**	**N**	**N**	**-**

Edit [i, j]

Edit Distance

❖ Problem) calculate the edit distance between S1 and S2 (cont'd)

- **Case 2:** when insertion occurs in the last column

edit[i, j] = edit[i-1, j] + 1

edit [i-1, j]

S1	**S**	**-**	**N**	**O**	**-**
S2	**S**	**U**	**N**	**N**	**W**

edit [i, j]

- **Case 3:** when substitution occurs in the last column

edit[i, j-1] = edit[i-1, j-1] $\leftarrow$ S1[i] = S2[j]

edit[i, j-1] = edit[i-1, j-2] + 1 $\leftarrow$ S1[i] $\neq$ S2[j]

edit [i-1, j-1]

S1	**S**	**-**	**N**	**O**	**P**
S2	**S**	**U**	**N**	**N**	**W**

edit [i, j]

Edit Distance

❖ Problem) calculate the edit distance between S1 and S2 (cont'd)

- Base case) edit(i, 0) = edit(0, i) = i for all i (i deletions & additions)

edit[i, 0] = i

edit [i, 0]

S1	**S**	**U**	**N**	**O**	**Y**
S2	**-**	**-**	**-**	**-**	**-**

- Recurrence relation:

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

If |S1|=n and |S2|=m, then the edit distance between S1 and S2 is equal to edit(n, m)

Edit Distance

❖ Example) S1 = SNOW, S2 = SUNNY (row-major order)

(1)

		S	U	N	N	Y
	0	1	2	3	4	5
S	1					
N	2					
O	3					
W	4					

Base case

(2)

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0				
N	2					
O	3					
W	4					

S1[1] = S2[1]

edit[1][1] =min(edit[0][0], edit[0][1]+1, edit[1][0]+1) = 0

(3)

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0	1			
N	2					
O	3					
W	4					

S1[1] $\neq$ S2[2]

edit[1][2] =
min(edit[0][1]+1, edit[1][1]+1, edit[0][2]+1) = 1

Edit Distance

❖ Example) S1 = SNOW, S2 = SUNNY (row-major order)

(4)

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0	1	2		
N	2					
O	3					
W	4					

S1[1] ≠ S2[3]

edit[1][3] =
min(edit[0][2]+1, edit[1][2]+1, edit[0][3]+1) = 2

(5)

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0	1	2	3	
N	2					
O	3					
W	4					

S1[1] ≠ S2[4]

edit[1][4] =
min(edit[0][3]+1, edit[1][3]+1, edit[0][4]+1) = 3

Edit Distance

❖ Example) S1 = SNOW, S2 = SUNNY (row-major order)

(6)

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0	1	2	3	4
N	2					
O	3					
W	4					

S1[1] $\neq$ S2[5]

edit[1][5] =
min(edit[0][4]+1, edit[1][4]+1, edit[0][5]+1) = 4

Competing the first row

(7)

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0	1	2	3	4
N	2	1	1	1	2	3
O	3	2	2	2	2	3
W	4	3	3	3	3	3

S1[4] $\neq$ S2[5]

edit[4][5] =
min(edit[3][4]+1, edit[4][4]+1, edit[3][5]+1)

Edit distance between S1 and S2 = edit[4, 5] = 3

Edit Distance

❖ Pseudo-code

```
EDITDISTANCE(A[1..m], B[1..n]):
  for j ← 0 to n
      Edit[0, j] ← j
  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          ins ← Edit[i, j − 1] + 1
          del ← Edit[i − 1, j] + 1
          if A[i] = B[j]
              rep ← Edit[i − 1, j − 1]
          else
              rep ← Edit[i − 1, j − 1] + 1
          Edit[i, j] ← min{ins, del, rep}
  return Edit[m, n]
```

Summary

- Dynamic programming
  - Recursion
  - Memoization

- Longest increasing subsequence (LIS)

- Edit distance

- Assignment (~11/11 23:59:59)
  - Draw edit distance table (S1 = ‘homogeneous’, S2 = ‘heterogeneity’)
  - Time complexity of edit distance

Questions?

SEE YOU NEXT TIME!