

## Lecture15

---

# CSE 421, Introduction to Algorithms

Lecture 15, Winter 2024

Dynamic Programming

Longest Common Subsequence

1

# Announcements

- **Dynamic Programming Reading:**
  - **Weighted Interval Scheduling, Segmented Least Squares, Knapsack and Subset Sum**
  - 6.6 String Alignment
  - 6.8 Shortest Paths (Bellman-Ford)
- **Midterm, Friday, Feb 9**
  - Material through 6.3 and HW 5
  - Feb 8 Section will be Midterm review
    - Old exam problems on course homepage
  - Homework 6 due Feb 14

*m. Day*

# Longest Common Subsequence

- $C=c_1\dots c_g$  is a subsequence of  $A=a_1\dots a_m$  if  $C$  can be obtained by removing elements from  $A$  (but retaining order)
- $LCS(A, B)$ : A maximum length sequence that is a subsequence of both  $A$  and  $B$

$LCS(\text{BARTHOLEMEWSIMPSON}, \text{KRUSTYTHECLOWN})$   
= RTHOWN

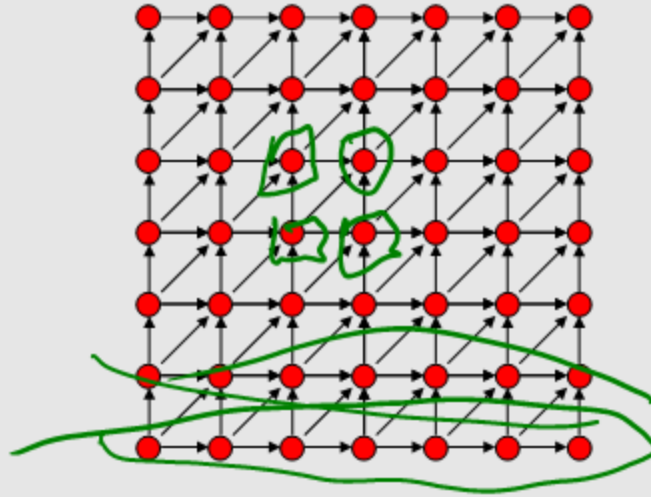


# Optimization recurrence

$$\text{If } a_j = b_k, \text{ Opt}[j,k] = 1 + \text{Opt}[j-1, k-1]$$

$$\text{If } a_j \neq b_k, \text{ Opt}[j,k] = \max(\text{Opt}[j-1,k], \text{Opt}[j,k-1])$$

# Dynamic Programming Computation



# Code to compute $\text{Opt}[n, m]$

```
for (int i = 1; i < n + 1; i++)
  for (int j = 1; j < m + 1; j++)
    if (A[ i ] == B[ j ] )
      Opt[ i, j ] = Opt[ i-1, j-1 ] + 1;
    else if (Opt[ i-1, j ] >= Opt[ i, j-1 ] )
      Opt[ i, j ] := Opt[ i-1, j ];
    else
      Opt[ i, j ] := Opt[ i, j-1];
```

# Storing the path information

$A[1..m]$ ,  $B[1..n]$

for  $i := 1$  to  $m$      $Opt[i, 0] := 0$ ;

for  $j := 1$  to  $n$      $Opt[0, j] := 0$ ;

$Opt[0, 0] := 0$ ;

for  $i := 1$  to  $m$

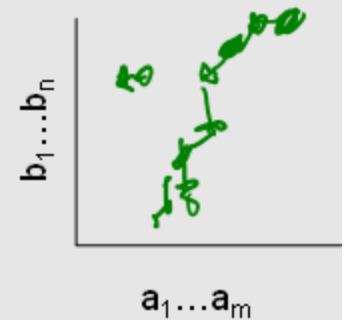
    for  $j := 1$  to  $n$

        if  $A[i] = B[j]$  {  $Opt[i, j] := 1 + Opt[i-1, j-1]$ ;  $Best[i, j] := \text{Diag}$ ; }

        else if  $Opt[i-1, j] \geq Opt[i, j-1]$

            {  $Opt[i, j] := Opt[i-1, j]$ ,  $Best[i, j] := \text{Left}$ ; }

        else     {  $Opt[i, j] := Opt[i, j-1]$ ,  $Best[i, j] := \text{Down}$ ; }







# How good is this algorithm?

- Is it feasible to compute the LCS of two strings of length 300,000 on a standard desktop PC? Why or why not.

(300,000)

$3 \times 10^5$

100,000,000,000

$9 \times 10^{10} \approx 10^{11}$

# Implementation 1

```
public int ComputeLCS() {
    int n = str1.Length;
    int m = str2.Length;

    int[,] opt = new int[n + 1, m + 1];
    for (int i = 0; i <= n; i++)
        opt[i, 0] = 0;
    for (int j = 0; j <= m; j++)
        opt[0, j] = 0;

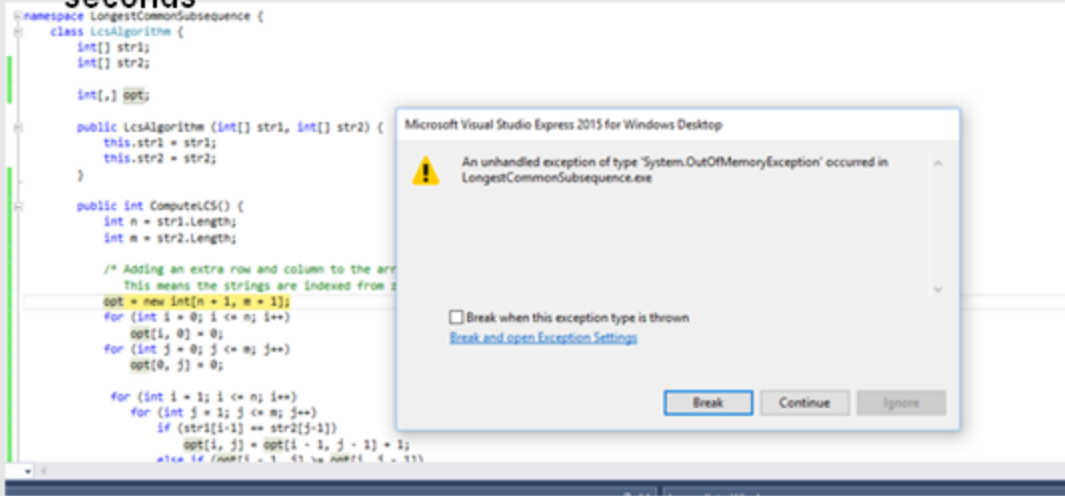
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (str1[i-1] == str2[j-1])
                opt[i, j] = opt[i - 1, j - 1] + 1;
            else if (opt[i - 1, j] >= opt[i, j - 1])
                opt[i, j] = opt[i - 1, j];
            else
                opt[i, j] = opt[i, j - 1];

    return opt[n,m];
}
```

10

# N = 17000

## Runtime should be about 5 seconds\*



\* Personal PC, 10 years old

Manufacturer:	Dell
Model:	Optiplex 990
Processor:	Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz 3.10 GHz
Installed memory (RAM):	8.00 GB (7.88 GB usable)
System type:	64-bit Operating System, x64-based processor

# Implementation 2

```
public int SpaceEfficientLCS() {
    int n = str1.Length;
    int m = str2.Length;
    int[] prevRow = new int[m + 1];
    int[] currRow = new int[m + 1];

    for (int j = 0; j <= m; j++)
        prevRow[j] = 0;

    for (int i = 1; i <= n; i++) {
        currRow[0] = 0;
        for (int j = 1; j <= m; j++) {
            if (str1[i - 1] == str2[j - 1])
                currRow[j] = prevRow[j - 1] + 1;
            else if (prevRow[j] >= currRow[j - 1])
                currRow[j] = prevRow[j];
            else
                currRow[j] = currRow[j - 1];
        }
        for (int j = 1; j <= m; j++)
            prevRow[j] = currRow[j];
    }

    return currRow[m];
}
```

12

$$N = 300000$$

N: 10000	Base 2 Length: 8096	Gamma: 0.8096	Runtime:00:00:01.86
N: 20000	Base 2 Length: 16231	Gamma: 0.81155	Runtime:00:00:07.45
N: 30000	Base 2 Length: 24317	Gamma: 0.8105667	Runtime:00:00:16.82
N: 40000	Base 2 Length: 32510	Gamma: 0.81275	Runtime:00:00:29.84
N: 50000	Base 2 Length: 40563	Gamma: 0.81126	Runtime:00:00:46.78
N: 60000	Base 2 Length: 48700	Gamma: 0.8116667	Runtime:00:01:08.06
N: 70000	Base 2 Length: 56824	Gamma: 0.8117715	Runtime:00:01:33.36

N: 300000 Base 2 Length: 243605 Gamma: 0.8120167  
Runtime:00:28:07.32

Chvatal-Sankoff Constant:  $\gamma_k$

13

# Observations about the Algorithm

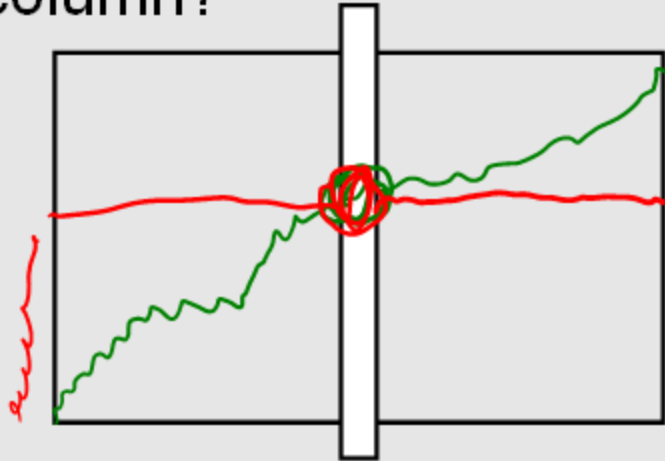
- The computation can be done in  $O(m+n)$  space if we only need one column of the Opt values or Best Values
- The algorithm can be run from either end of the strings

# Computing LCS in $O(nm)$ time and $O(n+m)$ space

- Divide and conquer algorithm
- Recomputing values used to save space

# Divide and Conquer Algorithm

- Where does the best path cross the middle column?



- For a fixed  $i$ , and for each  $j$ , compute the LCS that has  $a_i$  matched with  $b_j$



# Constrained LCS

- $LCS_{i,j}(A,B)$ : The LCS such that
  - $a_1, \dots, a_i$  paired with elements of  $b_1, \dots, b_j$
  - $a_{i+1}, \dots, a_m$  paired with elements of  $b_{j+1}, \dots, b_n$

- $LCS_{4,3}(\underbrace{abbacbb}_{\text{red}}, \underbrace{cbbaa}_{\text{red}}) = 2 + 0 = 2$

*Note: In the original image, blue brackets are drawn above the 'bb' in 'abbacbb' and above the 'bb' in 'cbbaa'. Red brackets are drawn below the 'abb' in 'abbacbb' and below the 'cbb' in 'cbbaa'.*

A = **RRSSRTTRTS**  
B = **RTSRRRSTST**

Compute  $LCS_{5,0}(A,B)$ ,  $LCS_{5,1}(A,B)$ , ...,  $LCS_{5,9}(A,B)$

0 1 1 2 3 3

A = **RRSSRTTRTS**  
 B = **RTSRRRSTST**

Compute  $LCS_{5,0}(A,B)$ ,  $LCS_{5,1}(A,B), \dots, LCS_{5,9}(A,B)$

j	left	right
0	0	4
1	1	4
2	1	3
3	2	3
4	3	3
5	3	2
6	3	2
7	3	1
8	4	1
9	4	0

4  
5  
4  
5  
6  
5  
5  
4  
5  
4

# Computing the middle column

- From the left, compute  $\text{LCS}(a_1 \dots a_{m/2}, b_1 \dots b_j)$
- From the right, compute  $\text{LCS}(a_{m/2+1} \dots a_m, b_{j+1} \dots b_n)$
- Add values for corresponding  $j$ 's



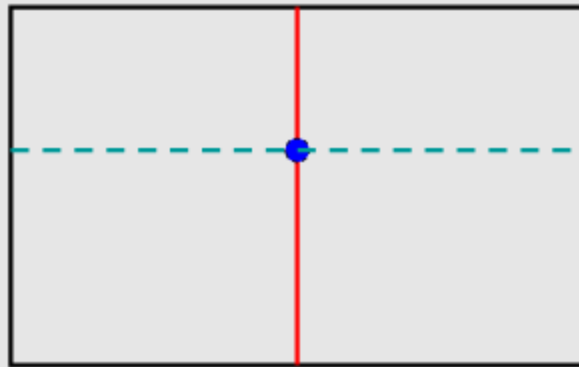
- Note – this is space efficient

# Divide and Conquer

- $A = a_1, \dots, a_m$                        $B = b_1, \dots, b_n$
- Find  $j$  such that
  - $\text{LCS}(a_1 \dots a_{m/2}, b_1 \dots b_j)$  and
  - $\text{LCS}(a_{m/2+1} \dots a_m, b_{j+1} \dots b_n)$  yield optimal solution
- Recurse

# Algorithm Analysis

- $T(m,n) = T(m/2, j) + T(m/2, n-j) + \underline{cnm}$   
for some  $j$ ,  $0 \leq j \leq n$



Prove by induction that

$$T(m,n) \leq 2cmn$$

Assume  $T(k, j) \leq 2ckj$

$$T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm$$

$$\begin{aligned} T(m,n) &\leq 2c \frac{m}{2} j + 2c \frac{m}{2} (n-j) + cnm \quad \text{for } k < m \\ &= \cancel{2c \frac{m}{2}} (j + n - j) + cnm \\ &= cmn + cnm = 2nm \end{aligned}$$

# Memory Efficient LCS Summary

- We can afford  $O(nm)$  time, but we can't afford  $O(nm)$  space
- If we only want to compute the length of the LCS, we can easily reduce space to  $O(n+m)$
- Avoid storing the value by recomputing values
  - Divide and conquer used to reduce problem sizes



# String Alignment Problem

- Align sequences with gaps

CAT TGA AT  
 CAGAT AGGA

- Charge  $\delta_x$  if character  $x$  is unmatched
- Charge  $\gamma_{xy}$  if character  $x$  is matched to character  $y$

Note: the problem is often expressed as a minimization problem, with  $\gamma_{xx} = 0$  and  $\delta_x > 0$

# Give the Optimization Recurrence for the String Alignment Problem

- Charge  $\delta_x$  if character  $x$  is unmatched
- Charge  $\gamma_{xy}$  if character  $x$  is matched to character  $y$

$$\text{Opt}[j, k] = \min \left( \begin{array}{l} \gamma_{xy} + \text{Opt}[j-1, i-1], \\ \delta_x + \text{Opt}[j-1, i], \\ \delta_y + \text{Opt}[j, i-1] \end{array} \right)$$

Let  $a_j = x$  and  $b_k = y$   
Express as minimization

# String edit with Typo Distance

- Find closest dictionary word to typed word
- $\text{Dist}('a', 's') = 1$
- $\text{Dist}('a', 'u') = 6$
- Capture the likelihood of mistyping characters

