

CS 545

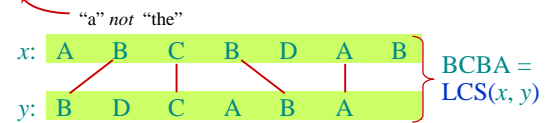
Dynamic Programming

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

Dynamic programming

Example 1: Longest Common Subsequence (LCS)

- Given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence common to them both.



Different phrasing: Find a set of a maximum number of segments, such that

- Each segment connects a character of x to an identical character of y ,
- Each character is used at most once
- Segments do not intersect.

Brute-force LCS algorithm

Check every subsequence of $x[1..m]$ to see if it is also a subsequence of $y[1..n]$.

Analysis

- Checking = $\Theta(m+n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time = $\Theta((m+n)2^m)$
 = exponential time.

Towards a better algorithm

Simplification:

- Look at the *length* of a longest-common subsequence.
- Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider *prefixes* of x and y .

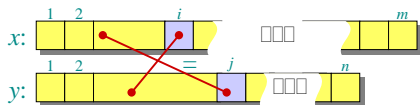
- Define $c[i, j] = |\text{LCS}(x[1..i], y[1..j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.

Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

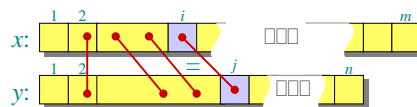
Proof: First Note that it is impossible that $x[i]$ is matched to an element in $y[1..j-1]$ and in addition $y[j]$ is matched to an element in $x[1..i-1]$



Recursive formulation-cont

Case (I): $x[i] = y[j]$. Claim: $c[i, j] = c[i-1, j-1] + 1$.

Proof.



We claim that there is a max matching that matches $x[i]$ to $y[j]$.

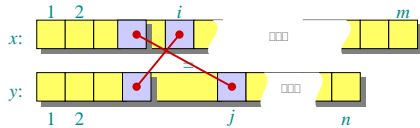
Indeed, if $x[i]$ is matched to $y[k]$ (for $k < j$) then $y[j]$ is unmatched (otherwise we have two crossing segments). Hence we can obtain another matching of the same cardinality by match $x[i]$ to $y[j]$.

This implies that we can match $x[1..i-1]$ to $y[1..j-1]$, and add the match $(x[i], y[j])$. So $c[i, j] = c[i-1, j-1] + 1$

Recursive formulation-cont

Case (II): $x[i] \neq y[j]$ Claim: $c[i, j] = \max\{c[i-1, j], c[i, j-1]\}$

Recall - in $\text{LCS}(x[1..i], y[1..j])$ it cannot be that both $x[i]$ and $y[j]$ are both matched.



If $x[i]$ is unmatched then

$$\text{LCS}(x[1..i], y[1..j]) = \text{LCS}(x[1..i-1], y[1..j])$$

If $y[j]$ is unmatched then

$$\text{LCS}(x[1..i], y[1..j]) = \text{LCS}(x[1..i], y[1..j-1])$$

So $c[i, j] = \max\{c[i-1, j], c[i, j-1]\}$

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \text{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y .

Recursive algorithm for LCS

```

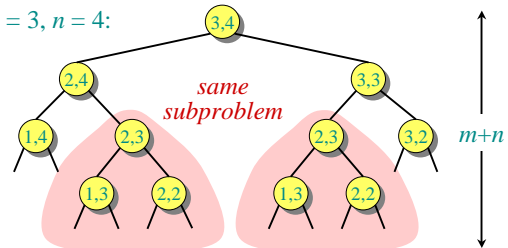
LCS(x, y, i, j)
  if ( i=0 or j=0) return 0
  if x[i] = y[j]
    then return LCS(x, y, i-1, j-1) + 1
  else return max { LCS(x, y, i-1, j),
                    LCS(x, y, i, j-1) }
    
```

To call the function $\text{LCS}(x, y, m, n)$

Worst-case: $x[i] \neq y[j]$, for all i, j in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree

$m = 3, n = 4$:



Height = $m + n \Rightarrow$ work potentially 2^{m+n} exponential. but we're solving subproblems already solved!

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn .

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```

LCS(x, y)
  for i=0 to m  c[i, 0] = 0
  for j=0 to n  c[0, j] = 0

  for i=1 to m
    for j=1 to n
      if (x[i] = y[j])
        then c[i, j] ← c[i-1, j-1] + 1
      else c[i, j] ← max { c[i-1, j], c[i, j-1] }
    
```

Time = $\Theta(mn)$ = constant work per table entry.
Space = $\Theta(mn)$.

LCS: Dynamic-programming algorithm

LCS(X,Y)="BCBA"

| | | | | | | | | |
|----|----|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | Y= | A | B | C | B | D | A | B |
| X= | B | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | D | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| | C | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| | A | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| | B | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
| | A | 0 | 1 | 2 | 2 | 3 | 4 | 4 |

X=B D C A B A
Y=A B C B D A B

Reconstruction $z=LCS(x,y)$

IDEA: Compute the table bottom-up. Fill z backward.

Observation: $c[i,j] \geq c[i-1,j]$ and $c[i,j] \geq c[i,j-1]$
Proof Sketch: We use a longer prefix, so there are more chars to be match.

LCS(x,y)="BCBA"

x=B D C A B A
y=A B C B D A B

LCS Reconstruction:

Set $i=m$; $j=n$; $k=c[i,j]$

While($k>0$) {

if ($c[i,j]>c[i-1,j]$ and $c[i,j]>c[i,j-1]$) {

$z[k]=x[i]$;

$i--$; $j--$; $k--$;

} else /* $c[i,j]=c[i-1,j]$ or $c[i,j]=c[i,j-1]$ */

if ($c[i,j]==c[i,j-1]$) $j--$;

else $i--$;

}

| | | | | | | | | |
|----|----|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | Y= | A | B | C | B | D | A | B |
| X= | B | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | D | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| | C | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| | A | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| | B | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
| | A | 0 | 1 | 2 | 2 | 3 | 4 | 4 |

Reconstructing $z=LCS(X,Y)$

Another idea – While filling $c[i,j]$, add arrows to each cell $c[i,j]$ specifying which neighboring cell $c[i,j]$ it got its value.

- $c[i,j].flag = "\backslash"$ if $c[i,j]=c[i-1,j-1]+1$
- $c[i,j].flag = "\uparrow"$ if $c[i,j]=c[i-1,j]$
- $c[i,j].flag = "\leftarrow"$ if $c[i,j]=c[i,j-1]$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | A | B | C | B | D | A | B |
| | 0 | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ |
| B | 0 | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ |
| D | 0 | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ |
| C | 0 | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ |
| A | 0 | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ |
| B | 0 | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ |
| A | 0 | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ | ↖ |

Example 2 of dynamic programming: Matrix Chain-Products



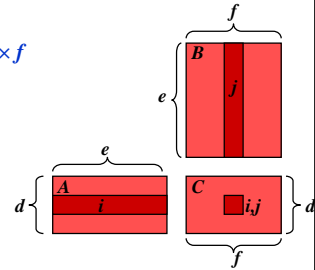
- Review: Matrix Multiplication.

$$- C = AB$$

$$- A \text{ is } d \times e, B \text{ is } e \times f$$

$$- O(def) \text{ time}$$

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] B[k, j]$$



Matrix Chain-Products

- **Matrix Chain-Product:**

– Compute $A = A_0 A_1 \dots A_{n-1}$

– A_i is $d_i \times d_{i+1}$

– Problem: How to parenthesize?

• Example 1. $(A_1 A_2)(A_3 A_4) = A_1(A_2(A_3 A_4)) = (A_1(A_2 A_3))A_4 = A_1((A_2 A_3)A_4) = \dots$

- Example 2

– B is 3×100

– C is 100×7

– D is 7×5

– $(BC)D$ $3 \times 100 \times 7 + 7 \times 5 \times 5 = 2275$ mults

– $B(CD)$ $3 \times 100 \times 5 + 100 \times 7 \times 5 = 5000$ mults



An Enumeration Approach

- **Matrix Chain-Product Alg.:**

– Try all possible ways to parenthesize

$$A = A_0 A_1 \dots A_{n-1}$$

– Calculate number of ops for each one

– Pick the one that is best

- Running time:

– # of parenthesizations = # of binary trees with n nodes

- **Exponential!**

• Called the n^{th} Catalan number – it is almost 4^n .

– This is a terrible algorithm!



A Greedy Approach



Repeatedly select the product that uses the fewest operations.

Counter-example:

- A is 101×11
- B is 11×9
- C is 9×100
- D is 100×99

- Idea selects $A((BC)D)$ $109989+9900+108900=228789$ mults
- Best is $(AB)(CD)$ $9999+89991+89100=189090$ mults

A "Recursive" Approach

- Define **subproblems**:
 - Find the best parenthesization of $A_i A_{i+1} \dots A_j$.
 - Let N_{ij} = # of operations done by this subproblem.
 - The optimal solution for the whole problem is $N_{0,n-1}$.
- **Subproblem optimality**: Assume the last multiplication taken place is multiplying $(A_0 \dots A_i)$ by $(A_{i+1} \dots A_{n-1})$.
 - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i} + N_{i+1,n-1}$ plus the time for the last multiply.
 - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

A Characterizing Equation



- Again assume the last multiplication is $(A_0 \dots A_i)(A_{i+1} \dots A_{n-1})$.
 - That is, we break at index i
- Consider all possible places for that final multiply (possible values of $0 \leq i \leq n-1$). That is...
 - $(A_0)(A_1 A_2 \dots A_{n-1})$, and $(A_0 A_1)(A_2 \dots A_{n-1})$, and
 - $(A_0 A_1 A_2)(A_3 \dots A_{n-1})$, $(A_0 \dots A_3)(A_4 \dots A_{n-1})$ etc till
 - $(A_0 A_{n-2})(A_{n-1})$.
- Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
 - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

I.e., break $(A_i \dots A_j)$, into $(A_i \dots A_k)(A_{k+1} \dots A_j)$,

A Dynamic Programming Algorithm



Since subproblems overlap, we don't use recursion.

Instead, we construct optimal subproblems "bottom-up."

$N_{i,j}$'s are easy, so start with them

Then do length 2,3,...

subproblems, and so on.

Running time:

$$O(n^3)$$

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: # of multiplications in optimal parenthesization of S

for $i \leftarrow 1$ to $n-1$ do $N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ to $n-1$ do //length of a run

for $i \leftarrow 0$ to $n-b-1$ do //start of run

$j \leftarrow i+b$ //end of run

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ to $j-1$ do //break pnt

$N_{i,j} \leftarrow \min\{N_{i,j},$

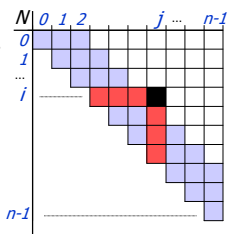
$N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

A Dynamic Programming Algorithm Visualization



$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i -th row and j -th column
- Filling in each entry in the N table takes $O(n)$ time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry (next slide).



Matrix Chain algorithm

How do we find the actual order of operations?



Algorithm *matrixChain*(S):

for $i \leftarrow 0$ to $n-1$ do

$N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ to $n-1$ do //length of a run

for $i \leftarrow 0$ to $n-b-1$ do //start of run

$j \leftarrow i+b$ //end of run

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ to $j-1$ do

sum = $N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}$

if (sum < $N_{i,j}$) then

$N_{i,j} \leftarrow$ sum

$O_{i,j} \leftarrow k$

return $N_{0,n-1}$

Example: ABCD

- A is 10×5

- B is 5×10

- C is 10×5

- D is 5×10

| N | 0 | 1 | 2 | 3 |
|---|---|-----|-------|----------|
| 0 | 0 | 500 | 0 | 1000 |
| 1 | A | AB | A(BC) | (A(BC))D |
| 2 | | B | BC | (BC)D |
| 3 | | | C | CD |
| | | | | D |

Recovering operations



- Example: ABCD
 - A is 10×5
 - B is 5×10
 - C is 10×5
 - D is 5×10

| | | | | |
|---|---|-----|-------|----------|
| N | 0 | 1 | 2 | 3 |
| 0 | 0 | 500 | 500 | 1000 |
| 1 | A | AB | A(BC) | (A(BC))D |
| 2 | | B | BC | (BC)D |
| 3 | | | C | CD |
| | | | | D |

// return expression for multiplying
// matrix chain A_1 through A_j

```
exp(i,j)
if (i=j) then // base case, 1 matrix
    return 'Ai'
else
    k = O[i,j] // see red values on left
    S1 = exp(i,k) // 2 recursive calls
    S2 = exp(k+1,j)
    return ('S1 S2')
```

The General Dynamic Programming Technique

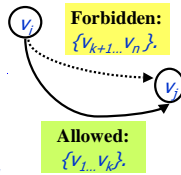


- Applies to a problem that at first seems to require a lot of time (often exponential), provided we have:
 - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m , and so on.
 - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems

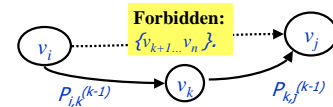
Example 3: All-Pairs Shortest Paths Floyd-Warshall alg



- Given a graph $G(V,E)$ with weights (positive and negative) assign to each edges. Assume $V = \{v_1, \dots, v_n\}$.
- Compute a matrix D such that $D[i,j]$ contains the length of the shortest path from v_i to v_j .
- Define $P_{ij}^{(k)}$ as the shortest path $v_i \rightarrow v_j$ that does **not** go through any of the vertices $\{v_{k+1}, \dots, v_n\}$. (that is, it is **allowed** to go through any of $\{v_1, \dots, v_k\}$.)
- $D_{ij}^{(k)}$ - the length of $P_{ij}^{(k)}$
- We compute D_0 first, then D_1 , etc.



This example appears in the shortest paths" chapter of CLRS (25.2)

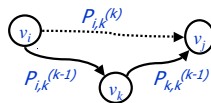


- Assume $D_{k-1}[i,j]$ has been computed ($1 < i, j < n$).
- We now want to compute $D_k[i,j]$. I.e. now we **can** (but don't have to) go through v_k on the shortest path $v_i \rightarrow v_j$.
- **Two possibilities:**
 - Going through v_k is longer, and better stick to $P_{ij}^{(k-1)}$ (previous found shortest path $v_i \rightarrow v_j$)
 - Use $P_{i,k}^{(k-1)}$, the shortest path $v_i \rightarrow v_k$ to reach v_k , and continue along $P_{k,j}^{(k-1)}$ to v_j .
- $D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$

Floyd Warshall-Pairs Shortest Paths Computing $D_k[i,j]$ for every i,j,k .

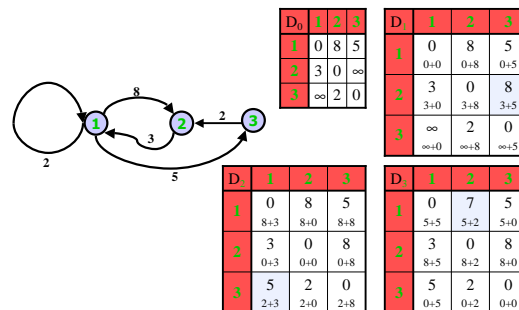


Algorithm *AllPair*(G) for all vertex pairs (i,j)
 if $i=j$ then $D_0[i,i] \leftarrow 0$
 else if (v_i, v_j) is an edge in G
 $D_0[i,j] \leftarrow w(v_i, v_j)$
 else
 $D_0[i,j] \leftarrow +\infty$



for $k \leftarrow 1$ to n do
 for $i \leftarrow 1$ to n do
 for $j \leftarrow 1$ to n do
 $D_k[i,j] = \min\{ D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j] \}$
 return D_n

Floyd's algorithm: example



Example 4: Edit distance

Given strings x, y , the **edit distance** $ed(x, y)$ between x and y is defined as the minimum number of operations that we need to perform on x , in order to obtain y .

Definition: An Operations (in this context)
Insertion/Deletion/Replacement of a **single** character.

Examples:
 $ed("aaba", "aaba") = 0$
 $ed("aaa", "aaba") = 1$
 $ed("aaaa", "abaa") = 1$
 $ed("baaa", "") = 4$
 $ed("baaa", "aab") = 2$

Example 4': ``Priced'' Edit distance $ed(x, y)$

Assume also given

$InsCost$ - the cost of a single **insertion** into x .
 $DelCost$ - the cost of a single **deletion** from x , and
 $RepCost$ - the cost of **replacing** one character of x
 by a different character.

Definition: Given strings x, y , the **edit distance** $ed(x, y)$ between x and y is the cheapest sequence of operations, starting on x and ending at y .

Problem: Compute $ed(x, y)$, and compute the sequence of operations.

Theorem:

Let $c[i, j] = ed(x[1..i], y[1..j])$ then

If $x[i] = y[j]$ then $c[i, j] = c[i-1, j-1]$

If $x[i] \neq y[j]$ then $c[i, j] = \min \{$
 $c[i-1, j] + InsCost,$
 $c[i, j-1] + DelCost,$
 $c[i-1, j-1] + RepCost,$
 $\}$

Algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```

ed(x, y)
  for i=0 to m  c[i, 0] = 0
  for j=0 to n  c[0, j] = 0

  for i=1 to m
    for j=1 to n
      if (x[i] == y[j])
        then c[i, j] ← c[i-1, j-1]
      else c[i, j] ← min{
                    c[i-1, j] + InsCost,
                    c[i-1, j-1] + RepCost,
                    c[i, j-1] + DelCost
                }
    
```

Time = $\Theta(mn)$ = constant work per table entry. Space = $\Theta(mn)$.