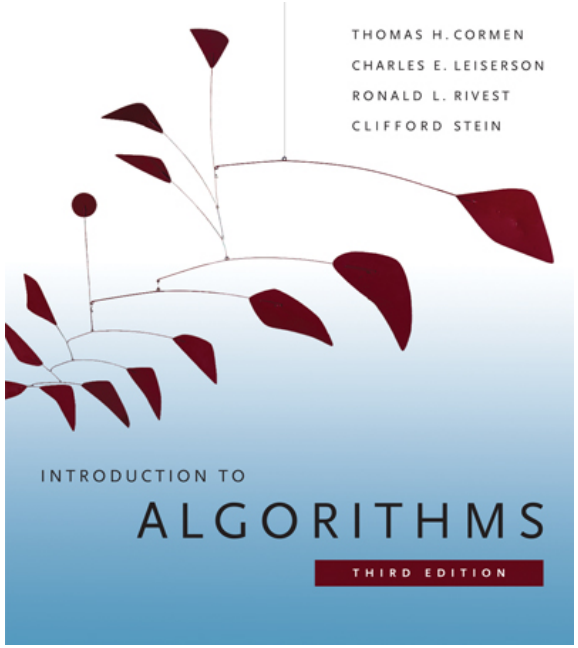
6.006- *Introduction to Algorithms*



Lecture 19

Prof. Constantinos Daskalakis

Lecture overview

- review of last lecture
  - key aspects of Dynamic Programming (DP)
  - all-pairs shortest paths as a DP
- a smarter DP for all-pairs shortest paths
- longest common subsequence

CLRS 15.3, 15.4, 25.1, 25.2

Dynamic Programming Definition

- DP ≈ Recursion + Memoization
- Typically, but not always, applied to optimization problems – so far: Fibonacci, Crazy eights, SPPs
- DP works when:
  - the solution can be produced by combining solutions to subproblems; e.g. $F_n=F_{n-1}+F_{n-2}$
  - the solution to each subproblem can be produced by combining solutions to sub-subproblems, etc;

    $F_{n-1}=F_{n-2}+F_{n-3}$ $F_{n-2}=F_{n-3}+F_{n-4}$

    moreover it's efficient when….
  - the total number of subproblems arising recursively is polynomial. $F_1, F_2,\ldots, F_n$

Dynamic Programming Definition

- DP ≈ Recursion + Memoization
- Typically, but not always, applied to optimization problems – so far: Fibonacci, Crazy eights, SPPs
- DP works when:

Optimal substructure

The solution to a problem can be obtained by solutions to subproblems.

$F_n = F_{n-1} + F_{n-2}$

Overlapping Subproblems

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

$F_1, F_2, \ldots, F_n$

All-pairs shortest paths

- **Input:** Directed graph $G = (V, E)$, where $V = \{1, \ldots, n\}$, with edge-weight function $w : E \to \mathsf{R}$.
- **Output**: An $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

Assumption: No negative-weight cycles

Dynamic Programming Approach

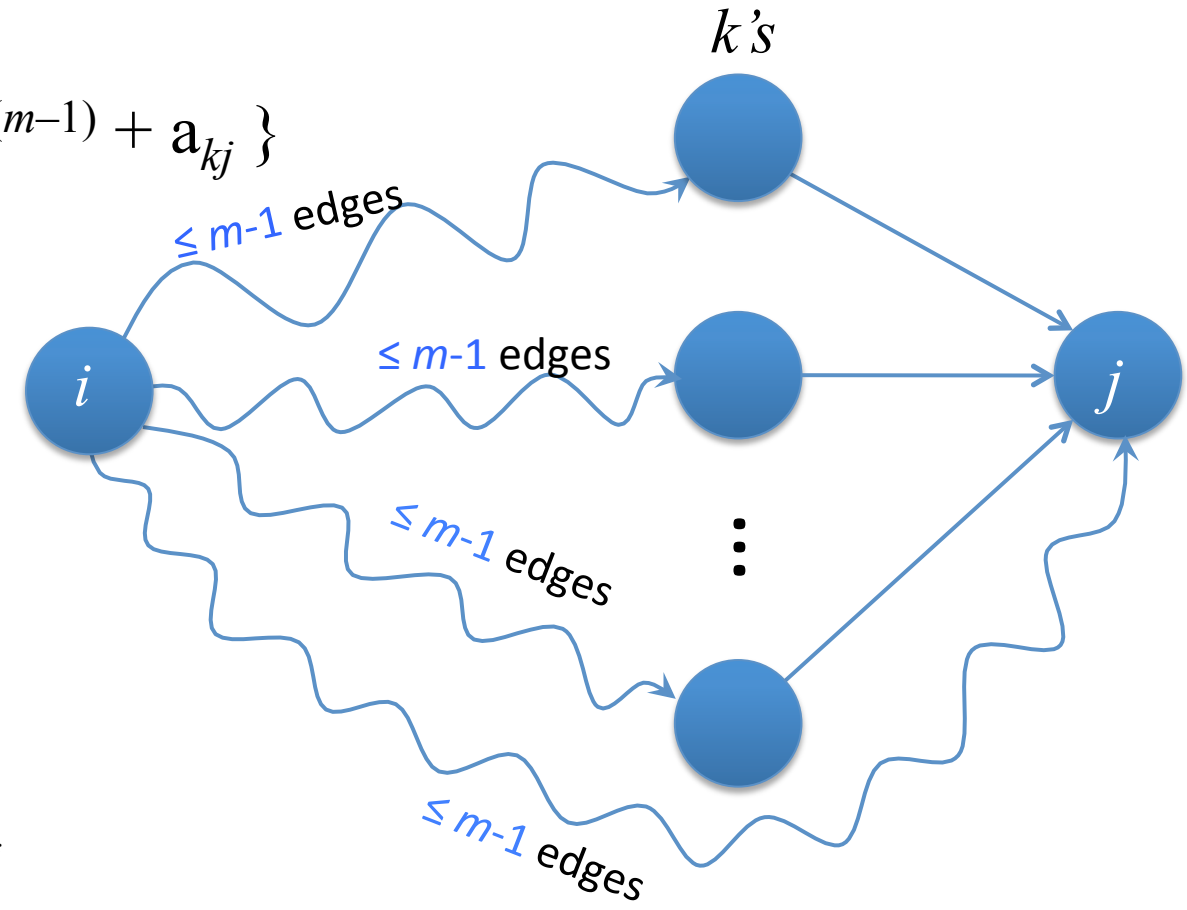
- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij} = w(i,j)$, if $(i, j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that uses <u>at most</u> m edges
- Want: $d_{ij}^{(n-1)}$
- $d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

Claim: For $m = 1, 2, \ldots, n-1$,

$$\mathbf{d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.}$$

Proof of Claim

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$




```
for k ← 1 to n
    if d_ij > d_ik + a_kj
        d_ij ← d_ik + a_kj
```

"Relaxation" (recall Bellman-Ford lecture)

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij} = w(i,j)$, if $(i, j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that only uses <u>at most</u> m edges
- Want: $d_{ij}^{(n-1)}$
- $d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

Claim: For $m = 1, 2, \ldots, n-1$,

$$\mathbf{d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.}$$

$O(n^4)$ - similar to n runs of Bellman-Ford

Another DP Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij} = w(i,j)$, if $(i, j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)}$ = weight of a shortest path from i to j that only uses at most m edges
- Want: $d_{ij}^{(n-1)}$
- $d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

Claim: For $m = 1, 2, \ldots, n-1$,

$$\mathbf{d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.}$$

$O(n^4)$ - similar to n runs of Bellman-Ford

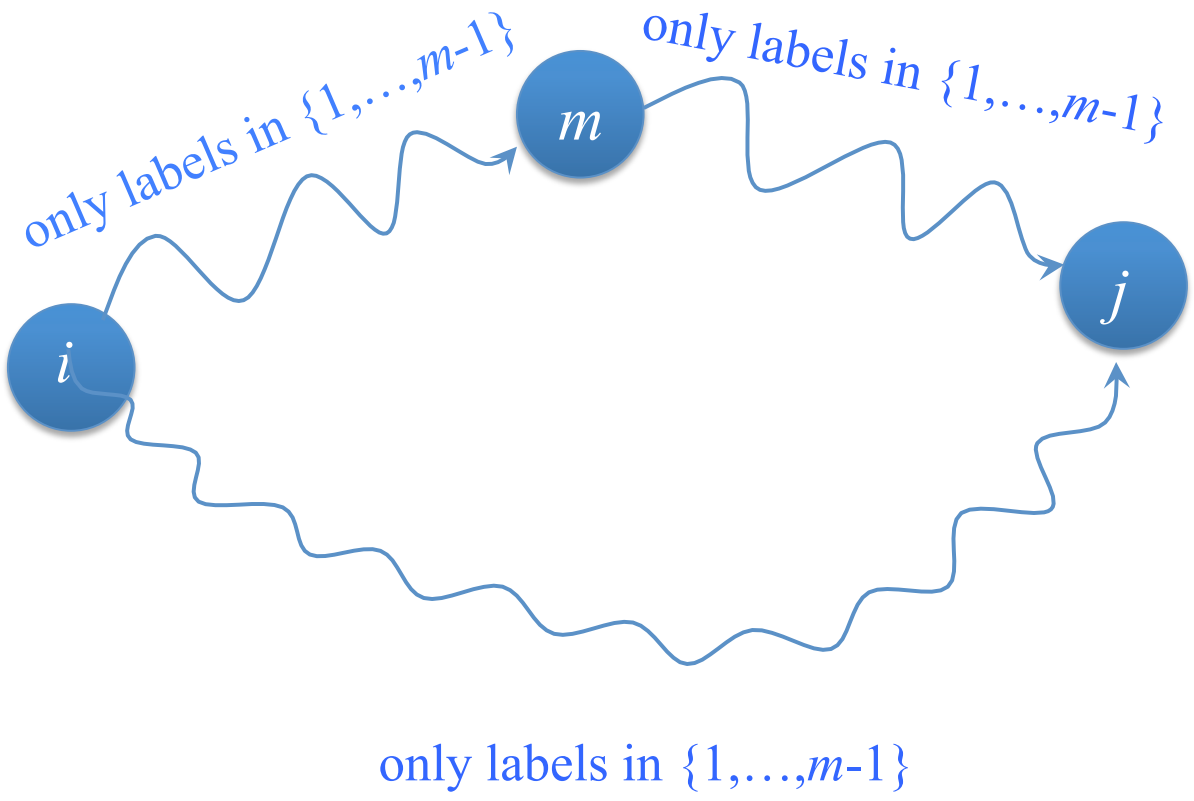
Another DP Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij} = w(i,j)$, if $(i, j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that only uses intermediate vertices from set $\{1,\ldots,m\}$
- Want: $d_{ij}^{(n)}$
- $d_{ij}^{(0)} = a_{ij}$;

Claim: For $m = 1, 2, \ldots, n$,

$$\mathbf{d_{ij}^{(m)} = \min\{d_{ij}^{(m-1)}, d_{im}^{(m-1)} + d_{mj}^{(m-1)}\}.}$$

Proof of Claim



$$\mathbf{d}_{ij}^{(m)} = \min\{\mathbf{d}_{ij}^{(m-1)}, \mathbf{d}_{im}^{(m-1)} + \mathbf{d}_{mj}^{(m-1)}\}$$

Another DP Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij} = w(i,j)$, if $(i, j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that only uses intermediate vertices from set $\{1,\ldots,m\}$
- Want: $d_{ij}^{(n)}$
- $d_{ij}^{(0)} = a_{ij}$;

Claim: For $m = 1, 2, \ldots, n$,

$$\mathbf{d_{ij}^{(m)} = \min\{d_{ij}^{(m-1)}, d_{im}^{(m-1)} + d_{mj}^{(m-1)}\}.}$$

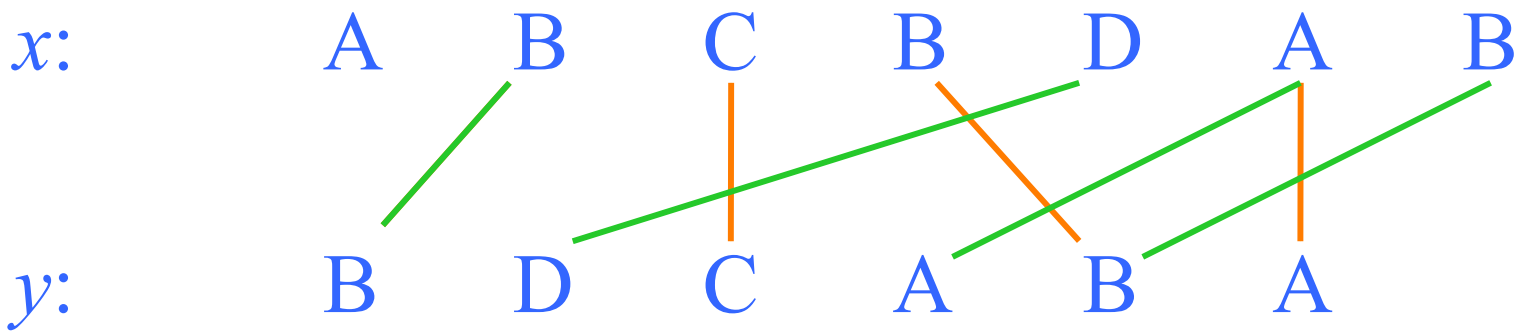
$O(n^3)$ running time (Floyd-Warshall)

Lecture overview

- review of last time
  - key aspects of Dynamic Programming (DP)
  - all-pairs shortest paths as a DP
- another DP for all-pairs shortest paths
- **longest common subsequence**

Longest Common Subsequence

- given two sequences x[1..*m*] and y[1..*n*], find a longest subsequence LCS(x,y) common to both:



- denote the length of a sequence *s* by |*s*|
- let us first try to get |LCS(x,y)|

Applications of LCS

- Tons in bioinformatics, e.g. long preserved regions in genomes
- file comparison, e.g. **diff**

original:

```
1  This part of the
2  document has stayed the
3  same from version to
4  version.  It shouldn't
5  be shown if it doesn't
6  change.  Otherwise, that
7  would not be helping to
8  compress the size of the
9  changes.
10
11 This paragraph contains
12 text that is outdated.
13 It will be deleted in th
14 near future.
15
16 It is important to spell
17 check this dokument. On
18 the other hand, a
19 misspelled word isn't
20 the end of the world.
21 Nothing in the rest of
22 this paragraph needs to
23 be changed. Things can
24 be added after it.
```

```
0a1,6
> This is an important
> notice! It should
> therefore be located at
> the beginning of this
> document!
>
8,14c14
< compress the size of the
< changes.
<
< This paragraph contains
< text that is outdated.
< It will be deleted in the
< near future.
---
> compress anything.
17c17
< check this dokument. On
---
> check this document. On
24a25,28
>
> This paragraph contains
> important new additions
> to this document.
```

new:

```
1  This is an important
2  notice! It should
3  therefore be located at
4  the beginning of this
5  document!
6
7  This part of the
8  document has stayed the
9  same from version to
10 version.  It shouldn't
11 be shown if it doesn't
12 change.  Otherwise, that
13 would not be helping to
14 compress anything.
15
16 It is important to spell
17 check this document. On
18 the other hand, a
19 misspelled word isn't
20 the end of the world.
21 Nothing in the rest of
22 this paragraph needs to
23 be changed. Things can
24 be added after it.
25
26 This paragraph contains
27 important new additions
28 to this document.
```

Brute force solution

- Given x[1..*m*] and y[1..*n*], how do we get the |LCS(x,y)| ?
- For every subsequence of x[1..*m*] , check if it is a subsequence of y[1..*n*]
- Analysis:
  - 2^m subsequences of x
  - each check takes $O(n)$ time ...
  - (two finger algorithm)
  - worst-case running time is $O(n2^m)$

Using prefixes

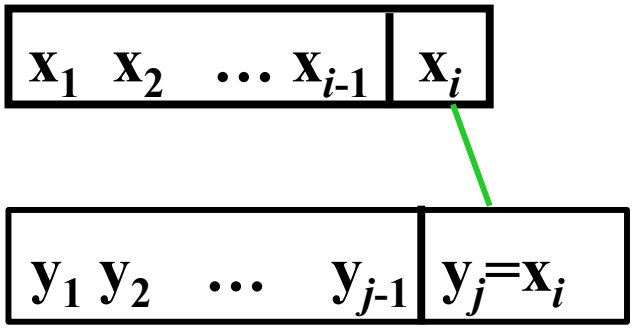
- consider prefixes of x and y
  - x[1..i] ith prefix of x[1..m]
  - y[1..j] jth prefix of y[1..n]

- subproblem: define c[i,j] = |LCS(x[1..i],y[1..j])|

- so c[m,n] = |LCS(x,y)|

- recurrence?

1) x[1..*i*] and y[1..*j*] end with $x_i = y_j$

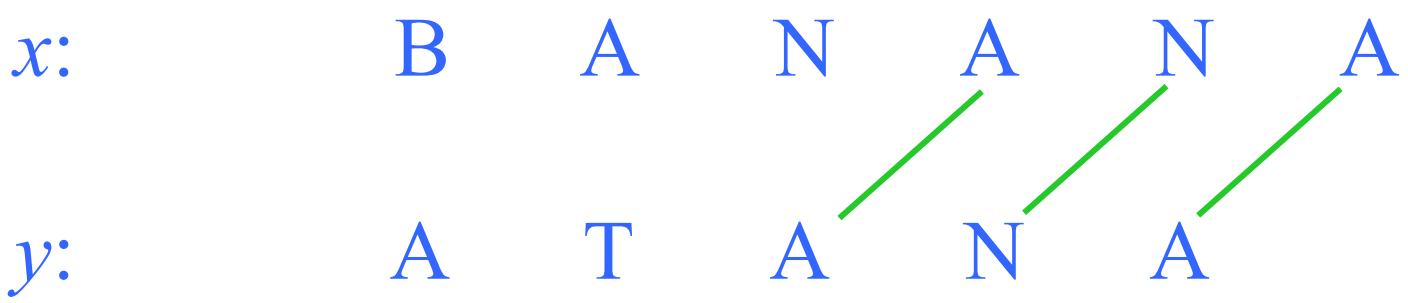


I might as well match x_i and y_j and look for LCS of x[1..*i*-1] and y[1..*j*-1].
So

$$c(i, j) = c(i\text{-}1, j\text{-}1)+1, \text{ if } x_i = y_j$$

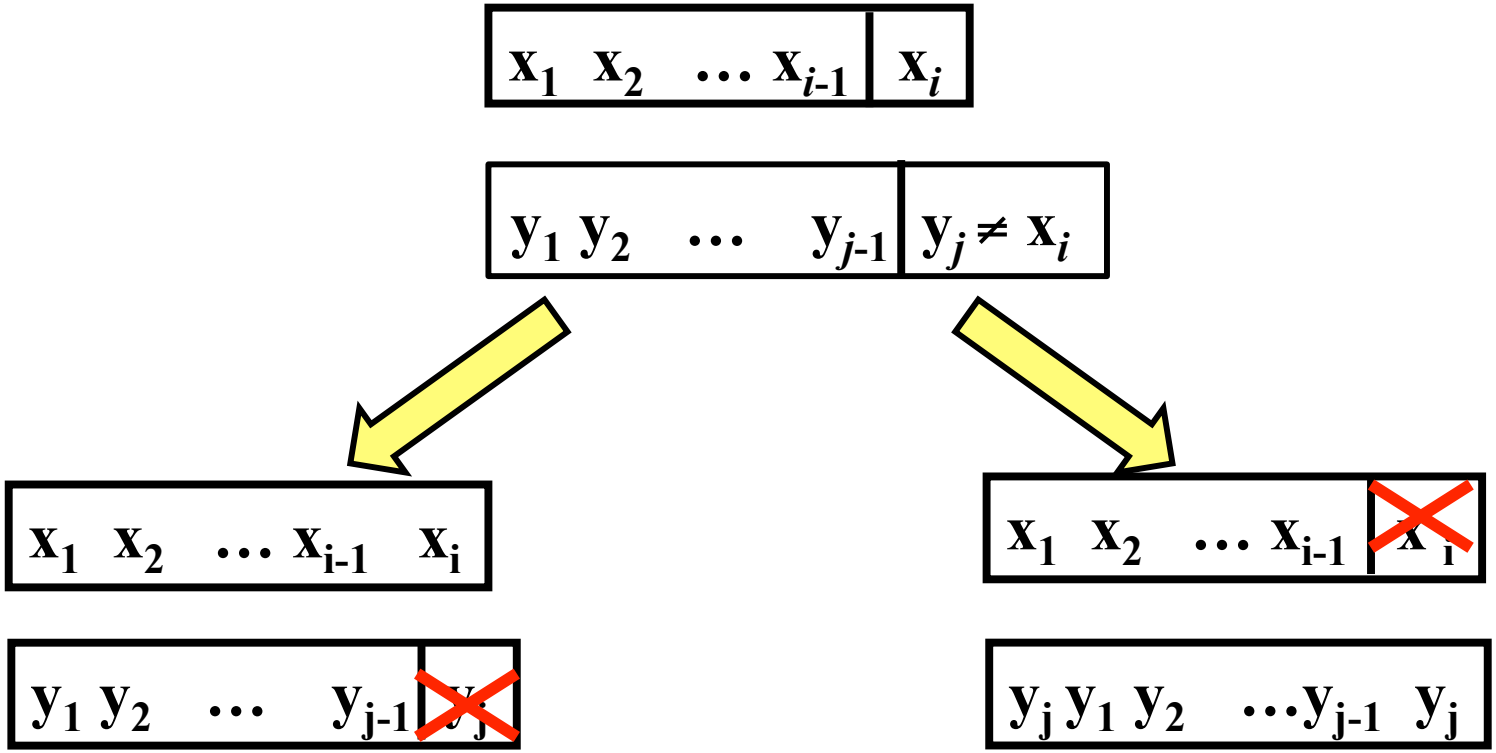
where recall c[*i*,*j*] = |LCS(x[1..*i*],y[1..*j*])|

Example - use of property 1



by inspection LCS of B A N and A T is A
so |LCS(x,y)| is 4

2) x[1..*i*] and y[1..*j*] end with $x_i \neq y_j$



LCS of x[1..*i*] and y[1..*j*-1]

LCS of x[1..*i*-1] and y[1..*j*]

$c(i, j)=\max\{c(i, j\text{-}1), c(i\text{-}1, j)\}$, if $x_i \neq y_j$

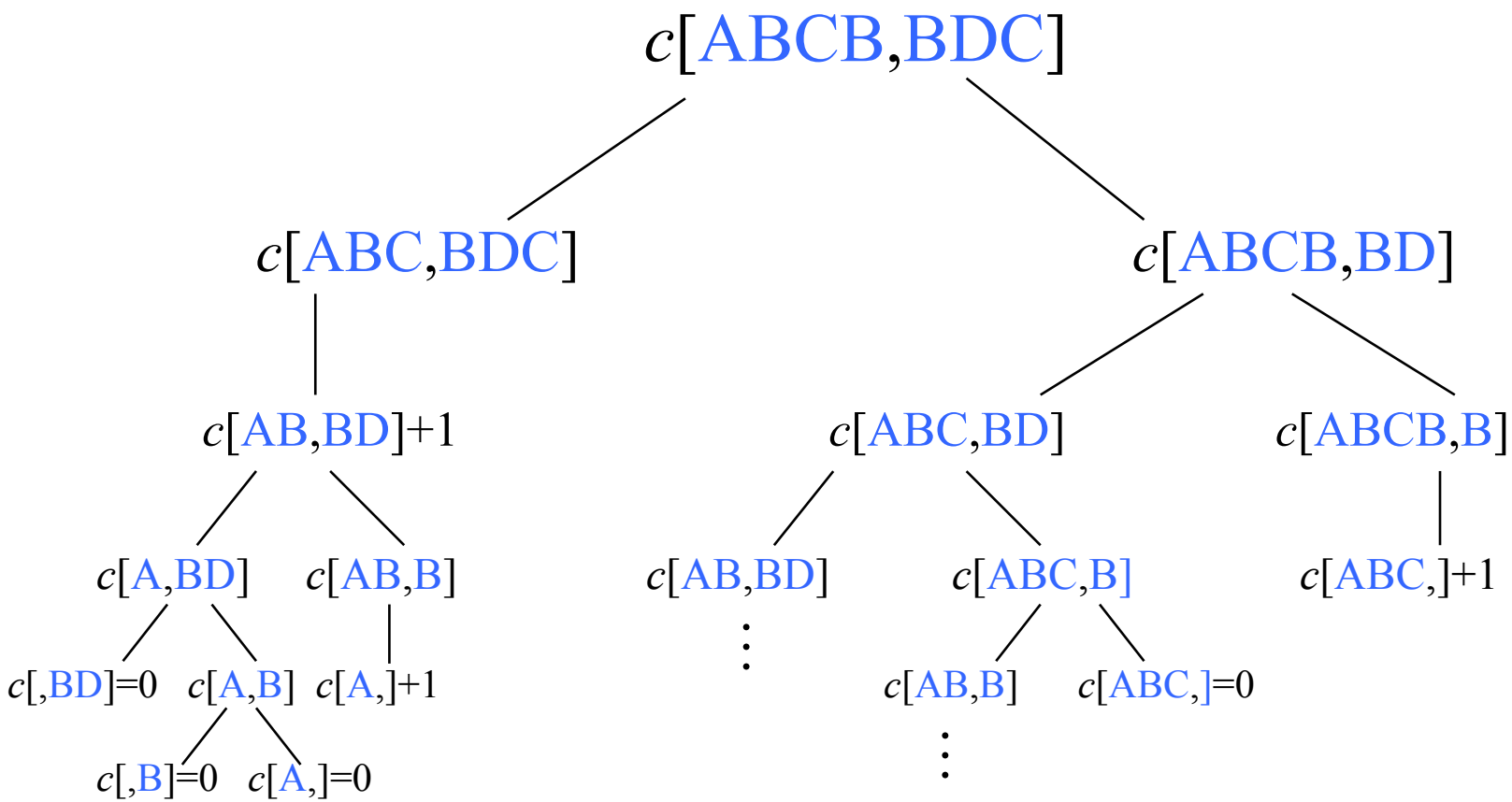
A recurrence, summary

- consider prefixes of x and y
  - x[1..*i*] ith prefix of x[1..*m*]
  - y[1..*j*] jth prefix of y[1..*n*]
- define c[*i*,*j*] = |LCS(x[1..*i*],y[1..*j*])|
  - so c[*m*,*n*] = |LCS(x,y)|
- recurrence:

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

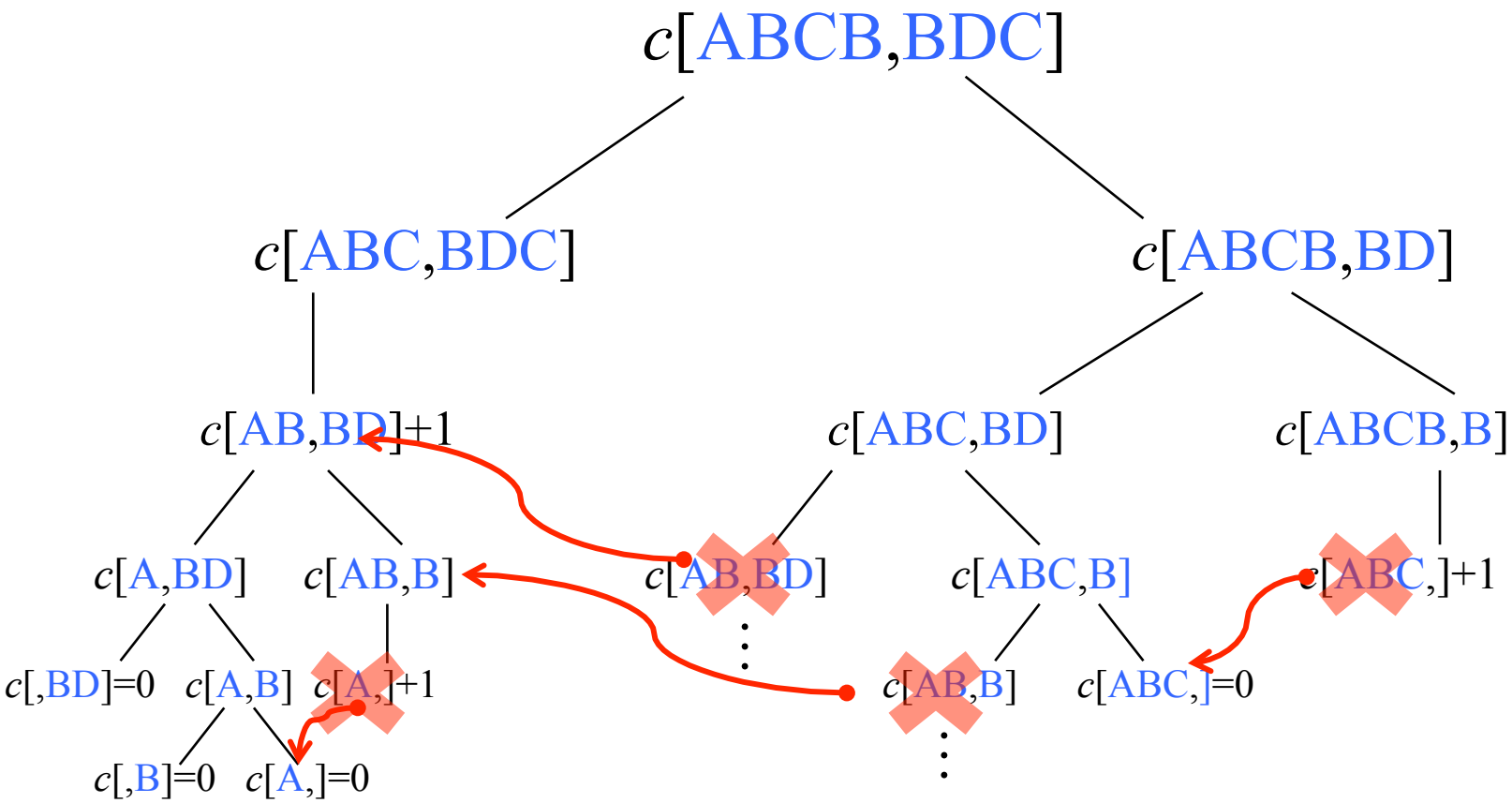
Solving LCS with Recursion

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



Solving LCS with Recursion+Memoization

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



Solving LCS with Recursion+Memoization

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

```
memo = { }
c(i, j):
    if (i, j) in memo: return memo[i, j]
    else if i=0 OR  j=0: return 0
    else if x_i=y_j: f = c(i-1, j-1)+1
    else f = max{c(i, j-1), c(i-1, j)}
    memo[i, j]=f
    return f
return c(n,m)
```

Solving LCS with Recursion+Memoization

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

- and the running time is $O(n \times m)$