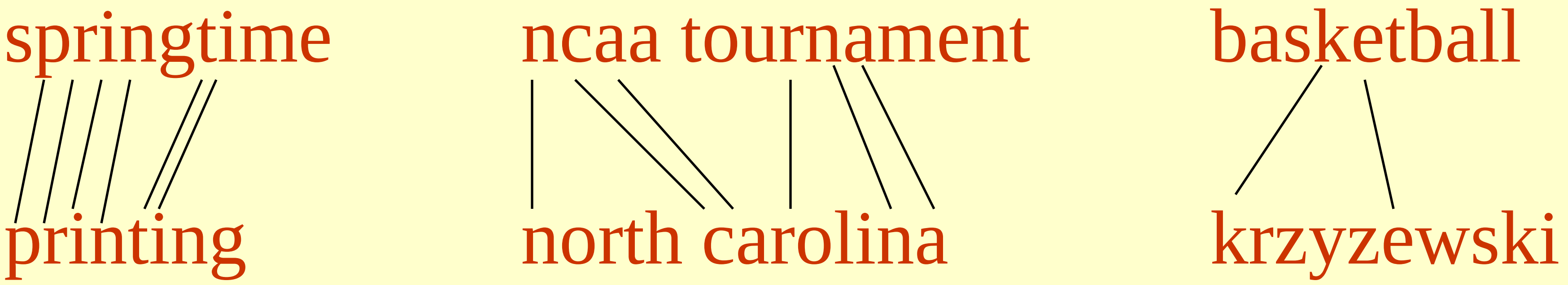
Dynamic Programming

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, find a common subsequence whose length is maximum.



Subsequence need not be consecutive, but must be in order.

Other sequence questions

- ***Edit distance:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, what is the minimum number of deletions, insertions, and changes that you must do to change one to another?
- ***Protein sequence alignment:*** Given a score matrix on amino acid pairs, $s(a,b)$ for $a,b \in \{\Lambda\} \cup A$, and 2 amino acid sequences, $X = \langle x_1,\ldots,x_m \rangle \in A^m$ and $Y = \langle y_1,\ldots,y_n \rangle \in A^n$, find the alignment with lowest score…

More problems

Optimal BST: Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i, build a binary search tree (BST) with minimum expected search cost.

Matrix chain multiplication: Given a sequence of matrices $A_1\ A_2\ \ldots\ A_n$, with A_i of dimension $m_i \times n_i$, insert parenthesis to minimize the total number of scalar multiplications.

Minimum convex decomposition of a polygon,

Hydrogen placement in protein structures, …

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - » Subproblems may share subsubproblems,
  - » However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - » Solving subproblems in a bottom-up fashion.
  - » Storing solution to a subproblem the first time it is solved.
  - » Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

We'll study these with the help of examples.

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, find a common subsequence whose length is maximum.

springtime　　ncaa tournament　　basketball

printing　　north carolina　　snoeyink

Subsequence need not be consecutive, but must be in order.

Naïve Algorithm

- For every subsequence of X, check whether it's a subsequence of Y .
- Time: $\Theta(n2^m)$.
  - » 2^m subsequences of X to check.
  - » Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, for second, and so on.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k\text{-}1}$ is an LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of $X_{m\text{-}1}$ and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and $Y_{n\text{-}1}$.

Notation:

prefix $X_i = \langle x_1,\ldots,x_i \rangle$ is the first i letters of X.

This says what any longest common subsequence must look like; do you believe it?

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k\text{-}1}$ is an LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of $X_{m\text{-}1}$ and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and $Y_{n\text{-}1}$.

Proof: (case 1: $x_m = y_n$)

Any sequence Z' that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,

(1) longest common subsequence (LCS) Z must end in $x_m = y_n$.

(2) $Z_{k\text{-}1}$ is a common subsequence of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, and

(3) there is no longer CS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, or Z would not be an LCS.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m,

(1) Z is a common subsequence of X_{m-1} and Y, and

(2) there is no longer CS of X_{m-1} and Y, or Z would not be an LCS.

Recursive Solution

- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Recursive Solution

$$c[\alpha,\beta]=\begin{cases}0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,}\\ c[\,prefix\alpha,\, prefix\beta]+1 & \text{if } \operatorname{end}(\alpha)=\operatorname{end}(\beta),\\ \max(c[\,prefix\alpha,\beta],c[\alpha,\, prefix\beta]) & \text{if } \operatorname{end}(\alpha)\neq\operatorname{end}(\beta).\end{cases}$$

c[springtime, printing]

c[springtim, printing] *c*[springtime, printin]

[springti, printing] [springtim, printin] ~~[springtim, printin]~~ [springtime, printi]

[springt, printing] [springti, printin] [springtim, printi] [springtime, print]

Recursive Solution

$$c[\alpha, \beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\alpha, prefix\beta] + 1 & \text{if } \operatorname{end}(\alpha) = \operatorname{end}(\beta), \\ \max(c[prefix\alpha, \beta], c[\alpha, prefix\beta]) & \text{if } \operatorname{end}(\alpha) \neq \operatorname{end}(\beta). \end{cases}$$

- Keep track of $c[\alpha,\beta]$ in a table of nm entries:
    - top/down
    - bottom/up

		p	r	i	n	t	i	n	g
s									
p									
r									
i									
n									
g									
t									
i									
m									
e									

Computing the length of an LCS

LCS-LENGTH (*X*, *Y*)

```
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j ] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
              then c[i, j ] ← c[i–1, j–1] + 1
                   b[i, j ] ← "↖"
              else if c[i–1, j ] ≥ c[i, j–1]
                      then c[i, j ] ← c[i– 1, j ]
                           b[i, j ] ← "↑"
                      else c[i, j ] ← c[i, j–1]
                           b[i, j ] ← "←"
return c and b
```

$b[i, j\,]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

$c[m,n]$ contains the length of an LCS of X and Y.

Time: $O(mn)$

Constructing an LCS

PRINT-LCS (b, X, i, j)

```
if i = 0 or j = 0
    then return
if b[i, j ] = "↖"
    then PRINT-LCS(b, X, i–1, j–1)
        print x_i
    elseif b[i, j ] = "↑"
        then PRINT-LCS(b, X, i–1, j)
else PRINT-LCS(b, X, i, j–1)
```

- Initial call is PRINT-LCS (b, X,m, n).
- When $b[i, j\,] = \nwarrow$, we have extended LCS by one character. So LCS = entries with $\nwarrow$ in them.
- Time: $O(m+n)$