6.006	Lecture 20	Apr. 24. 2008

Outline: Dynamic Programming II (of 4)
- review of big ideas & examples so far
- bottom-up implementation
- longest common subsequence
- parent pointers for guesses

Reading: CLRS 15

Summary:

* DP ≈ "controlled brute force"

* DP ≈ guessing + recursion + memoization

* DP ≈ dividing into reasonable # subproblems whose solutions relate — acyclicly — usually via guessing parts of solution

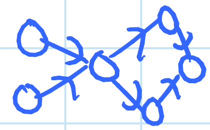
* time = # subproblems · time/subproblem
  - time/subproblem: treating recursive calls as $O(1)$ (usually mainly guessing)

- essentially an amortization
- count each subproblem only once; after first time, costs $O(1)$ via memoization

Examples:

Examples:	Fibonacci	Shortest Paths	Crazy Eights
subprobs.:	$fib(k)$, $0 \le k \le n$	$\delta_k(s,t)$ $\forall s, k<n$ = min. path $s \to t$ using $\le k$ edges	$trick(i)$ = longest trick from card i
#subprobs.:	$\Theta(n)$	$\Theta(V^2)$	$\Theta(n)$
guessing:	none	edge from s, if any	next card j
#choices:	1	$deg(s)$	$n-i$
relation:	$= fib(k-1) + fib(k-2)$	$= \min\{\delta_{k-1}(s,t)\} \cup \{w(s,v)+\delta_{k-1}(v,t) \mid v \in Adj[s]\}$	$= 1 + \max(trick(j)$ for $i<j<n$ if $match(c[i], c[j]))$
time/subpr.:	$\Theta(1)$	$\Theta(1+deg(s))$	$\Theta(n-i)$
DP time:	$\Theta(n)$	$\Theta(VE)$	$\Theta(n^2)$
orig. prob:	$fib(n)$	$\delta_{n-1}(s,t)$	$\max\{trick(i) \mid 0 \le i < n\}$
extra time:	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

Bottom-up implementation of DP: alternative to recursion

- subproblem dependencies form DAG 
- imagine topological sort
- iterate through subproblems in that order

⇒ when solving a subproblem, have already solved all dependencies

- often just: "solve smaller subproblems first"

e.g. Fibonacci:

```
    for k in range(n+1): fib[k] = ...
```

Shortest Paths:

```
    for k in range(n): for v in V: d[k,v,t] = ...
```

Crazy Eights:

```
    for i in reversed(range(n)): trick[i] = ...
```

- no recursion or memoized tests

⇒ faster in practice

- building DP table of solutions to all subprobs.
- can often optimize space:
  - Fibonacci: PS6
  - Shortest Paths: re-use same table $\forall k$

Longest common subsequence: (LCS)

A.K.A. edit distance, diff, CVS/SVN, spellchecking, DNA comparison, plagiarism detection, etc.

given two strings/sequences x & y, find longest common subsequence LCS(x,y)
sequential but not necessarily contiguous

- e.g.: H I E R O G L Y P H O L O G Y
  vs. M I C H A E L A N G E L O
  } → HELLO
- equivalent to "edit distance" (unit costs): # character insertions/deletions to transform $x \to y$
  (everything except the matches)

- brute force: try all $2^{|x|}$ subsequences of x
  $\Rightarrow \Theta(2^{|x|} \cdot |y|)$ time
- instead: DP on two sequences simultaneously

* useful subproblems for strings/sequences x:
  - suffixes $x[i:]$
  - prefixes $x[:i]$
  } $\Theta(|x|)$ ← cheaper ⇒ use if possible
  - substrings $x[i:j]$ } $\Theta(|x|^2)$

- idea: combine such subproblems for x & y
  (suffixes or prefixes work)

LCS DP:

- subproblem $c(i,j) = |LCS(x[i:], y[j:])|$ for $0 \le i,j < n$
  $\Rightarrow \Theta(n^2)$ subproblems
  - original problem $\approx c[0,0]$ (length ~ find seq. later)
- idea: either $x[i] = y[j]$ part of LCS or not $\Rightarrow$ either $x[i]$ or $y[j]$ (or both) not in LCS (with anyone)
- guess: drop $x[i]$ or $y[j]$? (2 choices)
- relation among subproblems:

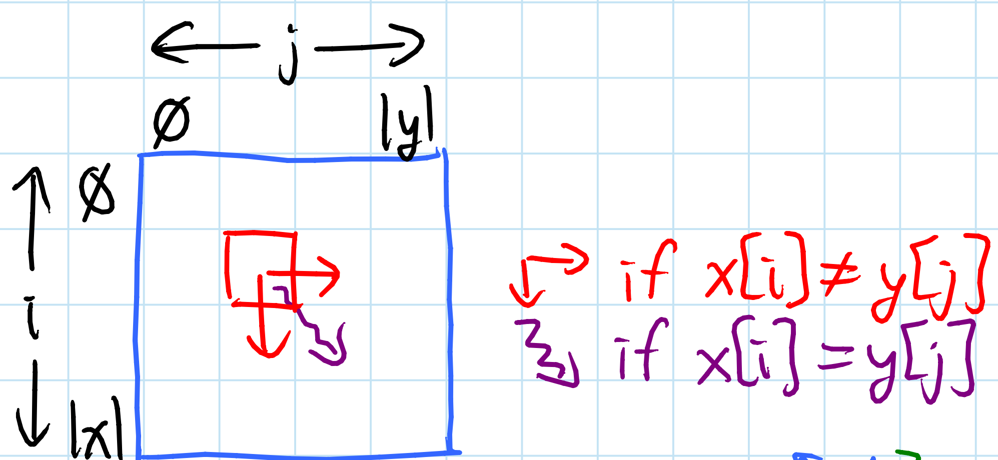
  if $x[i] = y[j]$: $c(i,j) = 1 + c(i+1, j+1)$
  (otherwise $x[i]$ or $y[j]$ unused ~ can't help)

  else: $c(i,j) = \max\{\underbrace{c(i+1,j)}_{x[i] \text{ out}}, \underbrace{c(i,j+1)}_{y[j] \text{ out}}\}$

  base cases: $c(|x|, j) = c(i, |y|) = \emptyset$

  $\Rightarrow \Theta(1)$ time per subproblem
  $\Rightarrow \Theta(n^2)$ total time for DP

- DP table:



[- linear space via antidiagonal order]

- recursive DP:

```
def LCS(x, y):
    seen = {}
    def c(i, j):
        if i ≥ len(x) or j ≥ len(y): return ∅
        if (i, j) not in seen:
            if x[i] == y[j]:
                seen[i, j] = 1 + c(i+1, j+1)
            else:
                seen[i, j] = max(c(i+1, j), c(i, j+1))
        return seen[i, j]
    return c(0, 0)
```

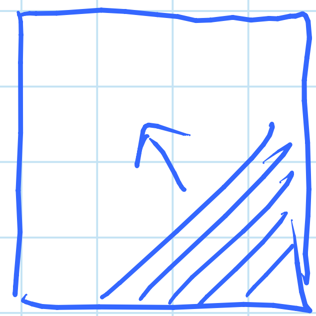
- bottom-up DP:

```
def LCS(x, y):
    c = {}
    for i in range(len(x)):
        c[i, len(y)] = ∅
    for j in range(len(y)):
        c[len(x), j] = ∅
    for i in reversed(range(len(x))):
        for j in reversed(range(len(y))):
            if x[i] == y[j]:
                c[i, j] = 1 + c[i+1, j+1]
            else:
                c[i, j] = max(c[i+1, j], c[i, j+1])
    return c[0, 0]
```

Recovering LCS: [material covered in recitation]

- to get LCS, not just its length, store parent pointers (like shortest paths) to remember correct choices for guesses:

```
if x[i] == y[j]:
    c[i,j] = 1 + c[i+1, j+1]
    parent[i,j] = (i+1, j+1)
else:
    if c[i+1, j] > c[i, j+1]:
        c[i,j] = c[i+1, j]
        parent[i,j] = (i+1, j)
    else:
        c[i,j] = c[i, j+1]
        parent[i,j] = (i, j+1)
```



- ... and follow them at the end:

```
lcs = []
here = (0, 0)
while c[here]:
    if x[i] == y[j]:
        lcs.append(x[i])
    here = parent[here]
```