Lecture Overview

- Review of big ideas & examples so far
- Bottom-up implementation
- Longest common subsequence
- Parent pointers for guesses

Readings

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 \times \text{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
<table>
<thead>
<tr>
<th>Examples:</th>
<th>Fibonacci</th>
<th>Shortest Paths</th>
<th>Crazy Eights</th>
</tr>
</thead>
<tbody>
<tr>
<td>subprobs:</td>
<td>fib(k)</td>
<td>δ_k(s, t) ∀ s, k &lt; n</td>
<td>trick(i) = longest</td>
</tr>
<tr>
<td></td>
<td>0 ≤ k ≤ n</td>
<td>= min path s → t</td>
<td>trick from card(i)</td>
</tr>
<tr>
<td>using ≤ k edges</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># subprobs:</td>
<td>Θ(n)</td>
<td>Θ(V^2)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>guessing:</td>
<td>none</td>
<td>edge from s, if any</td>
<td>next card j</td>
</tr>
<tr>
<td># choices:</td>
<td>1</td>
<td>deg(s)</td>
<td>n - i</td>
</tr>
<tr>
<td>relation:</td>
<td>= fib(k - 1) + fib(k - 2)</td>
<td>= min{δ_{k-1}(s, t)} + u{w(s, v) + δ_{k-1}(v, t)</td>
<td>= 1 + max(trick(j)) for i &lt; j &lt; n if match(c[i], c[j])</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time/subpr:</td>
<td>Θ(1)</td>
<td>Θ(1 + deg(s))</td>
<td>Θ(n - i)</td>
</tr>
<tr>
<td>DP time:</td>
<td>Θ(n)</td>
<td>Θ(VE)</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td>orig. prob:</td>
<td>fib(n)</td>
<td>δ_{n-1}(s, t)</td>
<td>max{trick(i), 0 ≤ i &lt; n}</td>
</tr>
<tr>
<td>extra time:</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>

Bottom-up implementation of DP:

- alternative to recursion
- subproblem dependencies form DAG (see Figure 1)
- imagine topological sort
- iterate through subproblems in that order
  ⇒ when solving a subproblem, have already solved all dependencies
- often just: “solve smaller subproblems first”

![Figure 1: DAG](image)

Example.

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 for memoized tests
  \[\implies\] faster in practice

• building \textbf{DP table} of solutions to all subprobs. can often optimize space:
  - Fibonacci: PS6
  - Shortest Paths: re-use same table \(\forall k\)

\textbf{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 \textbf{subsequence} \(\text{LCS}(x,y)\) sequential but not necessarily contiguous

- e.g., \texttt{H I E R O G L Y P H O L O G Y} vs. \texttt{M I C H A E L A N G E L O}
  
  common subsequence is \texttt{Hello}

- equivalent to “edit distance” (unit costs): \# character insertions/deletions to transform \(x\) \(\rightarrow\) \(y\) everything except the matches

- brute force: try all \(2^{|x|}\) subsequences of \(x\) \(\implies\) \(\Theta(2^{|x|} \cdot |y|)\) time

- instead: DP on two sequences simultaneously

\textbf{* Useful subproblems for strings/sequences \(x\):}

  - suffixes \(x[i:]\)
  
  - prefixes \(x[:i]\)
    
  The suffixes and prefixes are \(\Theta(|x|)\) \(\implies\) use if possible

  - substrings \(x[i:j]\) \(\Theta(|x|^2)\)

\textit{Idea:} Combine such subproblems for \(x\) \& \(y\) (suffixes and prefixes work)

\textbf{LCS DP}

\begin{itemize}
  \item subproblem \(c(i,j) = |\text{LCS}(x[i:],y[j:])|\) \(\text{for} 0 \leq i,j < n\)
  \[\implies\] \(\Theta(n^2)\) subproblems
  
  - original problem \(\approx c[0,0]\) (length \(\sim\) find seq. later)

  \item idea: either \(x[i]=y[j]\) part of LCS or not \(\implies\) either \(x[i]\) or \(y[j]\) (or both) not in LCS \((\text{with anyone})\)

  \item guess: drop \(x[i]\) or \(y[j]\)? \(\text{(2 choices)}\)
\end{itemize}
• relation among subproblems:

\[
\begin{align*}
\text{if } x[i] = y[j] & : c(i, j) = 1 + c(i + 1, j + 1) \\
\text{(otherwise } x[i] \text{ or } y[j] \text{ unused \sim can’t help)}
\end{align*}
\]

else: \( c(i, j) = \max\{c(i + 1, j), c(i, j + 1)\} \)

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

\implies \Theta(1) \text{ time per subproblem}

\implies \Theta(n^2) \text{ total time for DP}

• DP table: see Figure 2

\[ \text{Figure 2: DP Table} \]

• recursive DP:

```python
def LCS(x, y):
    seen = {}  # 
    def c(i, j):
        if i ≥ len(x) or j ≥ len(y) : return \phi
        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)] = φ
    for j in range(len(y)):
        c[len(x), j] = φ
    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]
                parent[i, j] = (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]
```