Admin: Quiz in-class 10/19

Today: Dynamic Programming

□ Longest Palindromic Subsequence
□ Optimal Binary Search Trees
□ Parsing for Context-Free Grammar
□ Parsing for Probabilistic Grammars

- Today’s problems deal with sequences (strings) where natural subproblem structure involves substrings
Longest Palindromic Subsequence

**Def:** A palindrome is a string that is unchanged when reversed.

**Examples:** radar, civic, t, bb, redder

**Given:** A string \( X[1..n] \) \( (n \geq 1) \)

**To find:** longest palindrome that is a subsequence

**Example:** Given "character"

output "carac"

**Note:** answer has length \( \geq 1 \) always.

[Ref: Problem 15-2 pg 405 of CLRS]
Ideas:

Let \( L(i, j) \) denote length of longest palindromic subsequence of \( X[i..j] \) for \( i \leq j \).

```python
def L(i, j):
    if i == j: return 1
    if X[i] != X[j]:
        return max(L(i+1, j), L(i, j-1))
    if i+1 == j: return 2
    return 2 + L(i+1, j-1)
```

As written, this program can run in exponential time: suppose all symbols \( X[i] \) distinct.

Let \( T(n) \) = running time on input of length \( n \).

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n-1) & \text{if } n > 1 
\end{cases}
\]

\[
= 2^{n-1}
\]
But there are only \( \binom{n^2}{2} = \Theta(n^2) \) distinct subproblems; each is an \((i,j)\) pair with \(i \leq j\).

By solving each subproblem only once, running time reduces to:

\[
\Theta(n^2) \cdot \Theta(1) = \Theta(n^2)
\]

\# subproblems \hspace{1cm} \text{time to solve subproblem, given that smaller ones solved.}

Can either:

1. **memoize** implementation above
   
   E.g., hash inputs/outputs for \(L\), & before solving subproblem, check to see if it is already solved, with soln in hash table.

2. solve subproblems in order of increasing “size” \((j-i)\), so smaller ones solved first. (“bottom up”)

**Common features to most DP problems:**

- recursive soln is exponential, but only a polynomial \# of distinct subproblems
- memoization, or bottom-up approach, gives polynomial running time, assuming each subproblem easy to solve given soln to smaller subproblems.
Optimal Binary Search Trees

Given: keys \( k_1, k_2, \ldots, k_n \) \((k_1 < k_2 < \ldots < k_n)\)
\( \text{wlog } k_i = i \) (e.g. search probabilities)

Weights \( w_1, w_2, \ldots, w_n \)

Find: BST that minimizes "cost"

\[
\sum_{i=1}^{n} w_i \cdot (\text{depth}_{T}(k_i) + 1)
\]

(equivalently \( \sum_{i=1}^{n} w_i \cdot \text{depth}_{T}(k_i) \), but we'll use)

Example: \( w_i = p_i = \text{probability of searching for } k_i \)

Then we are minimizing expected search cost.

[Ref: §15.5 of CLRS]
Let \( b_n = \# \) binary trees on \( n \) keys

\[
\frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix} \quad \text{"Catalan numbers"}
\]

\[
\simeq \Theta \left( \frac{4^n}{n^{3/2}} \right) \quad \text{exponentially!}
\]

(see Prob. 12-4 CLRS pg 306)

Too many to enumerate unless \( n \) is small.

\[
\begin{align*}
\text{\( n = 2 \)} & \quad \begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \draw (1) -- (2);
\end{tikzpicture}
\end{array} \\
\begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{\( b_2 = 2 \)} & \quad \begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{\( w_1 + 2w_2 \)} & \quad \text{\( 2w_1 + w_2 \) cost}
\end{align*}
\]

\[
\begin{align*}
\text{\( n = 3 \)} & \quad \begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array} \\
\begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array} \\
\begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{\( w_1 + 2w_2 + w_3 \)} & \quad \text{\( 2w_1 + 3w_2 + w_3 \) cost}
\end{align*}
\]

\[
\begin{align*}
\text{\( n = 3 \)} & \quad \begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array} \\
\begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array} \\
\begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{\( \text{\( b_3 = 5 \)} \)} & \quad \begin{array}{c}
\begin{tikzpicture}
  \node (1) at (0,0) {1};
  \node (2) at (0.5,0) {2};
  \node (3) at (0.5,0.5) {3};
  \draw (1) -- (2) -- (3);
\end{tikzpicture}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{\( \text{\( 3w_1 + 2w_2 + w_3 \)} \)} & \quad \text{\( 2w_1 + 3w_2 + w_3 \) cost}
\end{align*}
\]

\[
\begin{align*}
\text{\( \text{\( w_1 + 3w_2 + 2w_3 \)} \)} & \quad \text{\( w_1 + 2w_2 + 3w_3 \)}
\end{align*}
\]
Let $w(i, j) = w_i + w_i + \cdots + w_j$ \quad i \leq j

Let $e(i, j) =$ cost of optimal BST on $K_i, K_{i+1}, \ldots, K_j$

(so $e(1, n)$ is what we are after)

Suppose we try to solve problem in greedy manner

1. **Pick** $K_r$ is some "greedy" manner
   (e.g. key with max weight $w_r$ ???)

2. **Solve** induced subproblems
   
   \[ e(i, r-1) \quad \text{and} \quad e(r+1, j) \]
   optimally ("optimal substructure property")

But 😞 we don't know how to do 1!
Claim:
\[ e(i, j) = \begin{cases} 
  w_c & \text{if } i = j \\
  \min_{i \leq r \leq j} \left( e(i, r - 1) + e(r + 1, j) + w(i, j) \right) & \text{else}
\end{cases} \]

(The " + w(i,j)" accounts for both searching for root \( K_r \), and fact that all other elements "pushed down a level" in overall problem, compared to subproblems—they all have now to compare against root \( K_r \)).

Greedy, when it works, knows how to make local choice in globally optimal manner. (i.e. pick \( K_r \).)

Doesn't work here.

Dyn Prog tries all ways of making local choice, & takes advantage of "overlapping subproblems" via memoization to get poly-time algorithm.

Above recurrence can be solved in time \( \Theta(n^3) \):
\[
\frac{\Theta(n^2)}{\# \text{subproblems}} \cdot \frac{\Theta(n)}{\text{time per subproblem}} = \Theta(n^3)
\]
Parsing a string given context-free grammar (CFG):

terminal symbols: $\Sigma = \{a, b, c, \ldots \}$ (letters, digits, special)

nonterminal symbols: $N = \{A, B, C, \ldots \}$

rules describing structure of strings in language

nonterminal $\rightarrow$ given sequence of terminals & nonterminals

grammar = set of such rules

$S = \Sigma \cup N$ (all symbols)

Example:

\[
\begin{align*}
A & \rightarrow b \\
A & \rightarrow aB \\
B & \rightarrow BBb \\
B & \rightarrow CC \\
C & \rightarrow AB \\
C & \rightarrow a
\end{align*}
\]

$A \rightarrow aB \rightarrow aCC \rightarrow \ldots \rightarrow abaaa$

parse tree of string abaaa of nonterminals showing how it can be produced, starting from nonterminal $A$.

$A \rightarrow abaaa$

"$A$ derives abaaa"
Given: string $X[1..n]$ of symbols from $\Sigma$

CFG with nonterminals $N = \{ A_1, \ldots, A_m \}$
and $r$ rules
and $S = \Sigma \cup N$ symbols

Determine: if $A_1 \Rightarrow^* X[1..n]$

Let $P(i, j, V) =$

True if \( V = X[i] \) and $i = j$ or
\[
(V \in N \& V \Rightarrow^* X[i..j])
\]

$1 \leq i \leq j \leq n$

$V \in S$

$(n \cdot |S|)$ subproblems; we want $P(1, n, A_1)$
Solve recursively:

```python
def P(i, j, V):
    if V in Σ:
        return (i == j and X[i] == V)
    for each rule V → Y:
        if P(i, j, Y):
            return True
    for each rule V → YZ:
        for each k ∈ {i, ..., j - 1}:
            if P(i, k, Y) and P(k + 1, j, Z):
                return True
    return False
```

memoize!

# subproblems = (n²) * |Σ|

time to solve each is Θ(n³ * r)

Total time is Θ(n³ * |Σ| * r)

(Note: exercise to massage grammar first so there are no loops of form A₁ → A₂, A₂ → A₅, ..., A₅ → A₃; indeed, no RHS of rule with only single nonterminal)
Probabilistic Grammars

- Suppose now each rule has a probability $p_i$.

- Probability of derivation = product of probs of steps

\[
\begin{align*}
A & \rightarrow b \quad 1/2 \\
A & \rightarrow aB \quad 1/2 \\
& \quad \rightarrow a\ \ B \quad 1/2 \\
B & \rightarrow Bb \quad 3/4 \\
B & \rightarrow CC \quad 1/4 \\
& \quad \rightarrow a\ \ CC \quad 1/4 \\
C & \rightarrow AB \quad 5/8 \\
C & \rightarrow a \quad 3/8 \\
& \quad \rightarrow ab\ \ BC \quad 1/2 \\
& \quad \rightarrow ab\ \ a\ \ C \quad 3/8 \\
& \quad \rightarrow ab\ a\ a\ a\ \ a \quad 3/8 \\
\end{align*}
\]

\[
\prod p_i = 5.15 \times 10^{-4}
\]

- What is derivation of maximum probability?

- What is total probability of all derivations?

Call desired function $f$ or
Only minor changes to previous program needed.
replace predicate \( P(i, j, v) \) by \( f(i, j, v) \)
adjust program to take max, or sum, over rules. E.g. for total probability at all derivations:

```python
def f(i, j, v):
    sum = 0
    if v in \( \Sigma \):
        if i==j & X[i]=v:
            return 1
        else:
            return 0
    for each rule \( v \rightarrow y \):
        sum = sum + f(i, j, y)
    for each rule \( v \rightarrow y z \):
        for each \( k \in \{i, \ldots, j-1\} \):
            sum = sum + f(i, k, y) * f(k+1, j, z)
    return sum
```

Note: Special case is Viterbi alg, for regular languages:
all rules have form \( A \rightarrow a \) or \( A \rightarrow aB \)