Lecture 16: Dyn. Prog. Subproblems

Dynamic Programming Review

- Recursion where subproblems dependencies **overlap**
- "Recurse but reuse" (Top down: record and lookup subproblem solutions)
- "Careful brute force" (Bottom up: do each subproblem in order)

Dynamic Programming Steps (SR. BST)

1. Define **Subproblems** subproblem $x \in X$
   - Describe the meaning of a subproblem **in words**, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous subsequences
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** Subproblems $x(i) = f(x(j), \ldots)$ for one or more $j < i$
   - State topological order to argue relations are acyclic and form a DAG
3. Identify **Base** Cases
   - State solutions for all reachable independent subproblems
4. Compute **Solution** from Subproblems
   - Compute subproblems via top-down memoized recursion or bottom-up
   - State how to compute solution from subproblems (possibly via parent pointers)
5. Analyze Running **Time**
   - $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = W$ for all $x \in X$, then $|X| \times W$

Rod Cutting

- Given a rod of width n and the value $v(w)$ of any rod piece of integral width w for $1 \leq w \leq n$, cut the rod to maximize the value of cut rod pieces.
- Example: $n = 7$, $v = [0, 1, 10, 13, 18, 20, 31, 32]$
- Maybe greedily take most valuable per unit width?
- Nope! $\arg\max_w v[w]/w = 6$, and partitioning $[6, 1]$ yields 32 which is not optimal!
- Solution: $v[2] + v[2] + v[3] = 10 + 10 + 13 = 33$
- Maximization problem on value of partition

1. **Subproblems**
   - $\boxed{x(w)\text{: maximum value obtainable by cutting rod of width } w}$
2. **Relate**
   - First piece has some width p **(Guess!)**
   - $x(w) = \max\{v(p) + x(w - p) \mid p \in \{1, \ldots, w\}\}$
   - (draw dependency graph)
   - Subproblems $x(w)$ only depend on strictly smaller w, so acyclic
3. **Base**
   - $x(0) = 0$ (length zero rod has no value!)
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Maximum value obtainable by cutting rod of width n is $x(n)$
   - Store choices to reconstruct cuts
   - If current rod length w and optimal choice is w', remainder is piece $p = w - w'$
   - (path in subproblem DAG!)
5. **Time**
   - # subproblems: n
   - work per subproblem: $O(w)$
   - $O(n^2)$ running time

```python
# recursive
x = {}
def cut_rod(w, v):
    if w < 1:   return 0                           # base case
    if w not in x:                                 # check memo
        for piece in range(1, w + 1):              # try piece
            x_ = v[piece] + cut_rod(w - piece, v)  # recurrence
            if (w not in x) or (x[w] < x_):        # update memo
                x[w] = x_
    return x[w]
```

```python
# iterative
def cut_rod(n, v):
    x = [0] * (n + 1)                              # base case
    for w in range(n + 1):                         # topological order
        for piece in range(1, w + 1):              # try piece
            x_ = v[piece] + x[w - piece]           # recurrence
            if x[w] < x_:                          # update memo
                x[w] = x_
    return x[n]
```

```python
# iterative with parent pointers
def cut_rod_pieces(n, v):
    x = [0] * (n + 1)                              # base case
    parent = [None] * (n + 1)                      # parent pointers
    for w in range(1, n + 1):                      # topological order
        for piece in range(1, w + 1):              # try piece
            x_ = v[piece] + x[w - piece]           # recurrence
            if x[w] < x_:                          # update memo
                x[w] = x_
                parent[w] = w - piece              # update parent
    w, pieces = n, []
    while parent[w] is not None:                   # walk back through parents
        piece = w - parent[w]
        pieces.append(piece)
        w = parent[w]
    return pieces
```

Longest Common Subsequence

- Given two strings A and B, find a longest (not necessarily contiguous) subsequence of A that is also a subsequence of B.
- Example: $A =$ `akdfsjhlj`, $B =$ `adfjlkhoqeipr`
- Solution: `adfjl` or `adfjh`, both have length 5
- Maximization problem on length of subsequence

1. **Subproblems**
   - $\boxed{x(i, j) \text{ length of longest common subsequence of prefixes } A[:i] \text{ and } B[:j]}$
2. **Relate**
   - Either last characters match or they don't **(Guess!)**
   - If last characters match, some longest common subsequence will use them
   - (if no LCS uses last matched pair, using it will only improve solution)
   - (if an LCS uses last in $A[i]$ and not last in $B[j]$, matching $B[j]$ is also optimal)
   - If they do not match, they cannot both be in a longest common subsequence
   - $x(i, j) = \begin{cases} x(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max\{x(i-1, j), x(i, j-1)\} & \text{otherwise} \end{cases}$
   - (draw subset of all rectangular grid dependencies)
   - Subproblems $x(i, j)$ only depend on strictly smaller $i + j$, so acyclic
3. **Base**
   - $x(i, 0) = x(0, j) = 0$ (one string is empty)
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Length of longest common subsequence of A and B is $x(|A|, |B|)$
   - Store parent pointers to reconstruct subsequence
   - If the parent pointer decreases both indices, add that character
5. **Time**
   - # subproblems: $(|A| + 1)(|B| + 1)$
   - work per subproblem: $O(1)$
   - $O(|A||B|)$ running time

```
def lcs(A, B):
    a, b = len(A), len(B)
    x = [[0] * (a + 1) for _ in range(b + 1)]
    for i in range(a + 1):
        for j in range(b + 1):
            if A[i] == B[j]:
                x[i][j] = x[i - 1][j - 1] + 1
            else:
                if x[i - 1][j] < x[i][j - 1]:
                    x[i][j] = x[i - 1][j]
                else:
                    x[i][j] = x[i][j - 1]
    return x[a][b]
```