Lecture 16: Dyn. Prog. Subproblems

Dynamic Programming Review

- Recursion where subproblem dependencies overlap, forming DAG
- "Recurse but re-use" (Top down: record and lookup subproblem solutions)
- "Careful brute force" (Bottom up: do each subproblem in order)

Dynamic Programming Steps (SRT BOT)

1. **Subproblem** definition subproblem \( x \in X \)
   - Describe the meaning of a subproblem in words, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
   - Often multiply possible subsets across multiple inputs
   - Often record partial state: add subproblems by incrementing some auxiliary variables

2. **Relate** subproblem solutions recursively \( x(i) = f(x(j), \ldots) \) for one or more \( j < i \)
   - Identify a question about a subproblem solution that, if you knew the answer to, reduces the subproblem to smaller subproblem(s)
   - Locally brute-force all possible answers to the question

3. **Topological order** to argue relation is acyclic and subproblems form a DAG

4. **Base** cases
   - State solutions for all (reachable) independent subproblems where relation breaks down

5. **Original problem**
   - Show how to compute solution to original problem from solutions to subproblem(s)
   - Possibly use parent pointers to recover actual solution, not just objective function

6. **Time** analysis
   - \( \sum_{x \in X} \text{work}(x) \), or if \( \text{work}(x) = O(W) \) for all \( x \in X \), then \( |X| \cdot O(W) \)
   - \( \text{work}(x) \) measures nonrecursive work in relation; treat recursions as taking \( O(1) \) time
Longest Common Subsequence (LCS)

- Given two strings $A$ and $B$, find a longest (not necessarily contiguous) subsequence of $A$ that is also a subsequence of $B$.
- Example: $A = \text{hieroglyphology}, B = \text{michaelangelo}$
- Solution: hello or heglo or iello or ieglo, all length 5
- Maximization problem on length of subsequence

1. Subproblems
   - $x(i, j) =$ length of longest common subsequence of suffixes $A[i:]$ and $B[j:]$
   - For $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$  

2. Relate
   - Either first characters match or they don’t
   - If first characters match, some longest common subsequence will use them
   - (if no LCS uses first matched pair, using it will only improve solution)
   - (if an LCS uses first in $A[i]$ and not first in $B[j]$, matching $B[j]$ is also optimal)
   - If they do not match, they cannot both be in a longest common subsequence
   - **Guess** whether $A[i]$ or $B[j]$ is not in LCS
     - $x(i, j) = \begin{cases}  
     x(i+1, j+1) + 1 & \text{if } A[i] = B[j] \\
     \max\{x(i+1, j), x(i, j+1)\} & \text{otherwise} 
     \end{cases}$
   - (draw subset of all rectangular grid dependencies)

3. Topological order
   - Subproblems $x(i, j)$ depend only on strictly larger $i$ or $j$ or both
   - Simplest order to state: Decreasing $i+j$
   - Nice order for bottom-up code: Decreasing $i$, then decreasing $j$

4. Base
   - $x(i, |B|) = x(|A|, j) = 0$ (one string is empty)

5. Original problem
   - Length of longest common subsequence of $A$ and $B$ is $x(0, 0)$
   - Store parent pointers to reconstruct subsequence
   - If the parent pointer increases both indices, add that character to LCS
6. Time

- # subproblems: \((|A| + 1) \cdot (|B| + 1)\)
- work per subproblem: \(O(1)\)
- \(O(|A| \cdot |B|)\) running time

```python
def lcs(A, B):
a, b = len(A), len(B)
x = [[0] * (b + 1) for _ in range(a + 1)]
for i in reversed(range(a)):
    for j in reversed(range(b)):
        if A[i] == B[j]:
            x[i][j] = x[i + 1][j + 1] + 1
        else:
            x[i][j] = max(x[i + 1][j], x[i][j + 1])
return x[0][0]
```
Longest Increasing Subsequence (LIS)

- Given a string $A$, find a longest (not necessarily contiguous) subsequence of $A$ that strictly increases (lexicographically).
- Example: $A = \text{carbohydrate}$
- Solution: $\text{abort}$, of length 5
- Maximization problem on length of subsequence
- Attempted solution:
  - Natural subproblems are prefixes or suffixes of $A$, say suffix $A[i:]$
  - Natural question about LIS of $A[i:]$: is $A[i]$ in the LIS? (2 possible answers)
  - But then how do we recurse on $A[i+1:]$ and guarantee increasing subsequence?
  - Fix: add constraint to subproblems to give enough structure to achieve increasing property

1. Subproblems
   - $x(i) =$ length of longest increasing subsequence of suffix $A[i:]$ that includes $A[i]$
   - For $0 \leq i \leq |A|$

2. Relate
   - We’re told that $A[i]$ is in LIS (first element)
   - Next question: what is the second element of LIS?
     - Or $A[i]$ might be the last element of LIS
   - $x(i) = \max\{1 + x(j) \mid i < j < |A|, A[j] > A[i]\} \cup \{1\}$

3. Topological order
   - Decreasing $i$

4. Base
   - No base case necessary, because we consider the possibility that $A[i]$ is last

5. Original problem
   - What is the first element of LIS? Guess!
   - Length of LIS of $A$ is $\max\{x(i) \mid 0 \leq i < |A|\}$
   - Store parent pointers to reconstruct subsequence
6. Time

- # subproblems: $|A|$
- work per subproblem: $O(|A|)$
- $O(|A|^2)$ running time
- Exercise: speed up to $O(|A|\log |A|)$ by doing only $O(\log |A|)$ work per subproblem, via AVL tree augmentation

```python
def lis(A):
    a = len(A)
    x = [1] * a
    for i in reversed(range(a)):
        for j in range(i, a):
            if A[j] > A[i]:
                x[i] = max(x[i], 1 + x[j])
    return max(x)
```
Alternating Coin Game

- Given sequence of $n$ coins of value $v_0, v_1, \ldots, v_{n-1}$
- Two players (“me” and “you”) take turns
- In a turn, take first or last coin among remaining coins
- My goal is to maximize total value of my taken coins, where I go first
- First solution exploits that this is a zero-sum game: I take all coins you don’t

1. Subproblems
   - Choose subproblems that correspond to the state of the game
   - For every contiguous subsequence of coins from $i$ to $j$, $0 \leq i \leq j < n$
   - $x(i, j) =$ maximum total value I can take starting from coins of values $v_i, \ldots, v_j$

2. Relate
   - I must choose either coin $i$ or coin $j$ (Guess!)
   - Then it’s your turn, so you’ll get value $x(i + 1, j)$ or $x(i, j - 1)$, respectively
   - To figure out how much value I get, subtract this from total coin values
   - $x(i, j) = \max\{v_i + \sum_{k=i+1}^j v_k, x(i+1, j), v_j + \sum_{k=i}^{j-1} v_k, x(i, j - 1)\}$

3. Topological order
   - Increasing $j \quad i$

4. Base
   - $x(i, i) = v_i$

5. Original problem
   - $x(0, n - 1)$
   - Store parent pointers to reconstruct strategy

6. Time
   - # subproblems: $\Theta(n^2)$
   - work per subproblem: $\Theta(n)$ to compute sums
   - $\Theta(n^3)$ running time
   - Exercise: speed up to $\Theta(n^2)$ time by precomputing all sums $\sum_{k=i}^j v_k$ in $\Theta(n^2)$ time, via dynamic programming (!)
• Second solution uses subproblem expansion: add subproblems for when you move next

1. Subproblems

• Choose subproblems that correspond to the full state of the game
• Contiguous subsequence of coins from $i$ to $j$, and which player $p$ goes next
• $x(i, j, p) =$ maximum total value I can take when player $p \in \{\text{me, you}\}$ starts from coins of values $v_i, \ldots, v_j$

2. Relate

• Player $p$ must choose either coin $i$ or coin $j$ (Guess!)
• If $p = \text{me}$, then I get the value; otherwise, I get nothing
• Then it’s the other player’s turn
• $x(i, j, \text{me}) = \max \{v_i + x(i + 1, j, \text{you}), v_j + x(i, j - 1, \text{you})\}$
• $x(i, j, \text{you}) = \min \{x(i + 1, j, \text{me}), x(i, j - 1, \text{me})\}$

3. Topological order

• Increasing $j \quad i$

4. Base

• $x(i, i, \text{me}) = v_i$
• $x(i, i, \text{you}) = 0$

5. Original problem

• $x(0, n - 1, \text{me})$
• Store parent pointers to reconstruct strategy

6. Time

• # subproblems: $\Theta(n^2)$
• work per subproblem: $\Theta(1)$
• $\Theta(n^2)$ running time
Subproblem Constraints and Expansion

- We’ve now seen two examples of constraining or expanding subproblems
- If you find yourself lacking information to check the desired conditions of the problem, or lack the natural subproblem to recurse on, try subproblem constraint/expansion!
- More subproblems and constraints give the relation more to work with, so can make DP more feasible
- Usually a trade-off between number of subproblems and branching/complexity of relation
- More examples next lecture