Lecture 19: Dynamic Programming IV

Dynamic Programming

- SR. BST: Subproblems, Relate, Base cases, Solution, analyze running Time
- Subproblems are often subsets of the problem: prefixes, suffices, contiguous subsequences
- It's often useful/necessary to **expand subproblems to remember extra state information**
  - Usually by including additional parameters
  - Bellman-Ford was a good example: $x(v, k)$, storing a node v **and** a path-length k
  - Watch out: this can greatly increase the number of subproblems!

Arithmetic Parenthesization (Allowing Negatives)

- Input: arithmetic expression containing n integers, with integers a_i and a_{i+1} separated by binary operator $o_i(a, b)$ from $\{+, \times\}$
- Allow **negative** integers!
- Output: Where to place parentheses to maximize the evaluated expression
- Example: $7 + (-4) \times 3 + (-5) \rightarrow ((7) + ((-4) \times ((3) + (-5)))) = 15$

1. **Subproblems**
   - Sufficient to maximize each subarray? No! $(-3) \times (-3) = 9 > (-2) \times (-2) = 4$
   - $\boxed{x(i, j, +1)\text{: maximum parenthesized evaluation of subsequence from integer } i \text{ to } j}$
   - $\boxed{x(i, j, -1)\text{: minimum parenthesized evaluation of subsequence from integer } i \text{ to } j}$
2. **Relate**
   - Guess location of outer-most parenthesis, last operation evaluated
   - $x(i, j, +1) = \max\{o_k(x(i, k, s_1), x(k+1, j, s_2)) \mid k \in \{i, \ldots, j-1\},\ s_1, s_2 \in \{-1, +1\}\}$
   - $x(i, j, -1) = \min\{o_k(x(i, k, s_1), x(k+1, j, s_2)) \mid k \in \{i, \ldots, j-1\},\ s_1, s_2 \in \{-1, +1\}\}$
   - Subproblems $x(i, j, s)$ only depend on strictly smaller $j - i$, so acyclic
3. **Base**
   - $x(i, i, s) = a_i$

4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Maximum evaluated expression is given by $x(1, n, +1)$
   - Store parent pointers (two!) to find parenthesization, (forms binary tree!)
5. **Time**
   - # subproblems: less than $n \times n \times 2 = O(n^2)$
   - work per subproblem $O(n) \times 2 \times 2 = O(n)$
   - $O(n^3)$ running time

Egg Drop

- Drop eggs from floors of an n story building
- Want to find highest floor an egg can be dropped without breaking
- Want to minimize the number of drops for a fixed number of eggs
- If you only have one egg, test each floor going up until it breaks (n)
- If you have infinite eggs, binary search ($\log n$)
- If allowed to break at most k eggs, somewhere in between
- Solve using dynamic programming!

1. **Subproblems:**
   - Store number of floors remaining to check and number of unbroken eggs
   - $\boxed{x(f, e)\text{: minimum number of drops to check any sequence of } f \text{ floors using } e \text{ eggs}}$
2. **Relate:**
   - Case 1: drop an egg from a floor and it breaks
   - Case 2: drop an egg from a floor and it does not break
   - In the worst cast, an adversary picks the case that maximizes drops

   $$x(f, e) = 1 + \min\Big\{\max\{x(i-1, e-1), x(f-i, e)\} \;\Big|\; 1 \le i \le f\Big\}$$

   - Subproblems $x(f, e)$ only depend on subproblems with strictly fewer floors, so acyclic
3. **Base Cases**
   - $x(0, e) = 0$ (nothing to distinguish)
   - $x(f, 0) = \infty$ if $f > 0$ (can't succeed without eggs)
4. **Solution:**
   - Solve subproblems via recursive top down or iterative bottom up
   - For bottom up, can solve in order of increasing f, then increasing e
   - Final answer is $x(n, k)$
   - Can store parent pointers to reconstruct worst case optimal floor sequence
5. **Time:**
   - # subproblems: $(n + 1)k$
   - work per subproblem: $O(f) = O(n)$
   - $O(n^2k)$ running time