Quiz 3 Review

Scope

- Quiz 1 & Quiz 2 material fair game but explicitly **not emphasized**
- 4 lectures on dynamic programming, L15-L18, 2 Problem Sets, PS7-PS8
- Recursive framework (SRTBOT)
- Dynamic Programming: subproblem dependencies **overlap**, forming a DAG

Recursive Framework (SRT BOT)

1. **Subproblem** definition
   - Describe meaning of subproblem **in words**, in terms of parameters $x \in X$
   - Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
   - Often multiply possible subsets across multiple inputs
   - Often "remember" information by maintaining some auxiliary state (expansion)
   - Often expand based on state of integers in problem space (pseudopolynomial?)
2. **Relate** recursively
   - Relate subproblem solutions recursively $x(i) = f(x(j), \ldots)$ for one or more $j < i$
   - Identify question about subproblem solution whose answer would let you reduce to smaller subproblem(s), then locally brute-force all possible answers to the question
3. **Topo**logical order
   - Argue relation is acyclic (defines "smaller" subproblems), 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

Problem 1. Future Investing (adapted from S18 PS9)

Tiffany Bannen stumbles upon a lottery chart dropped by a time traveler from the future, which lists winning lottery numbers and positive integer cash payouts for the next n days. Tiffany wants to use this information to make money, but is worried if she plays winning numbers every day, lottery organizers will get suspicious. As such, she decides to play the lottery **infrequently**: at most **twice in any seven day period**. Describe a $O(n)$-time algorithm to determine the maximum amount of lottery winnings Tiff can win in the next n days by playing the lottery infrequently.

Solution:

1. **Subproblems**
   - Let $L(i)$ be the cash payout of playing the lottery on day $i \in \{1, \ldots, n\}$
   - Need to keep track of most recent two plays (or equivalently, restrictions on future plays)
   - $x(i, j)$: maximum lottery winnings playing on suffix of days from i to n, assuming play on day i and next allowable play is on day $i + j$
   - for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, 6\}$
2. **Relate**
   - Tiffany will play again on some day in future. Guess!
   - It is never optimal to go 11 days without playing the lottery, as playing on the 6th day would be valid and strictly increase winnings
   - The next play can be on day $i + k$ for $k \in \{j, \ldots, 11\}$
   - If next play on $i + k$ for $k \in \{1, \ldots, 6\}$, next allowable play is on day $i + 7$
   - If next play on $i + k$ for $k \in \{7, \ldots, 11\}$, next allowable play is on day $i + k + 1$
   - $x(i, j) = L(i) + \max\{x(i + k, \max\{1, 7 - k\}) \mid k \in \{i, \ldots, 11\} \text{ and } i + k \leq n\}$
3. **Topo**
   - Subproblems $x(i, j)$ only depend on strictly larger i, so acyclic
4. **Base**
   - $x(n, j) = L(i)$ for all $j \in \{1, \ldots, 6\}$
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Guess first play (within first seven days)
   - Solution to original problem is $\max\{x(i, 1) \mid i \in \{1, \ldots, 7\}\}$
   - (Can store parent pointers to reconstruct days played)
6. **Time**
   - # subproblems: $6n$
   - work per subproblem: $O(1)$
   - work for original: $O(1)$
   - $O(n)$ running time

Problem 2. Oh Charlie, Where Art Thou? (adapted from S18 PS9)

A wealthy family, Alice, Bob, and their young son Charlie are sailing around the world when they encounter a massive storm. Charlie is thrown overboard, presumed drowned. Twenty years later, a man comes to Alice and Bob claiming to be Charlie. Alice and Bob are excited, but skeptical. Alice and Bob order a DNA matching test from the genetic testing company 46AndThee. Given three length-n DNA sequences from Alice, Bob, and Charlie, the testing center will determine **ancestry** as follows: if Charlie's DNA can be partitioned into two (not necessarily contiguous) subsequences of equal length, where one is a subsequence of Alice's DNA, and the other is a subsequence of Bob's DNA, the Charlie is their son. For example, suppose Alice's DNA is `AATT` and Bob's DNA is `CCGG`. If Charlie's DNA were `CATG`, he would be matched as their son, since `CATG` can be partitioned into disjoint subsequences `CG` and `AT` which are subsequences of Alice and Bob's DNA respectively. However, Charlie would be found to be an imposter if his DNA were `AGTC`. Describe an $O(n^4)$-time algorithm to determine whether Charlie is a fraud.

Solution:

1. **Subproblems**
   - Let A, B, and C be the relevant length-n DNA sequences from Alice, Bob, and Charlie.
   - Want to match **some** characters of A and B to **all** characters of C
   - $x(i, j, k_i, k_j)$: true if can match a length-k_i subsequence of suffix $A[i :]$ and a length-k_j characters from prefix $B[j :]$ to all characters in suffix $C[(n - k_i - k_j) :]$ (the suffix containing the last $k_i + k_j$ characters), and false otherwise.
   - for $i, j \in \{0, \ldots, n\}$ and $k_i, k_j \in \{0, \ldots, n/2\}$ (assume n is even)
2. **Relate**
   - Must match character $C[i]$; if $A[i] = C[i]$ or $B[i] = C[i]$ recurse on remainder
   - Alternatively, do not use either $A[i]$ or $B[i]$
   - $$x(i, j, k_i, k_j) = \text{OR} \left\{ \begin{array}{ll} x(i+1, j, k_i+1, k_j) & \text{if } A[i] = C[n - k_i - k_j] \text{ and } k_i > 0, \\ x(i, j+1, k_i, k_j+1) & \text{if } B[j] = C[n - k_i - k_j] \text{ and } k_j > 0, \\ x(i+1, j, k_i, k_j) & \text{if } i < n, \\ x(i, j+1, k_i, k_j) & \text{if } j < n \end{array} \right\}$$
3. **Topo**
   - Subproblem $x(i, j, k_i, k_j)$ only depends on strictly smaller $i + j$, so acyclic
4. **Base**
   - $x(n, n, 0, 0)$ is true (all matched!)
   - $x(n, j, k_i, k_j)$ false if $k_i > 0$ (no more characters in A)
   - $x(i, n, k_i, k_j)$ false if $k_j > 0$ (no more characters in B)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(n, n, n/2, n/2)$

6. **Time**
   - # subproblems: $O(n^4)$
   - work per subproblem: $O(1)$
   - $O(n^4)$ running time

Problem 3. Sweet Tapas (adapted from S18 Final)

Obert Ratkins is having dinner at an upscale tapas bar, where he will order many small plates. There are n plates of food on the menu, where information for plate i is given by a triple of non-negative integers (v_i, c_i, s_i): the plate's volume v_i, calories c_i, and sweetness $s_i \in \{0, 1\}$ (the plate is sweet if $s_i = 1$ and not sweet if $s_i = 0$). Obert is on a **diet**: he wants to eat no more than k calories during his meal, but wants to fill his stomach as much as possible. He also wants to order exactly $s < n$ sweet plates, without purchasing the same dish twice. Describe an $O(nks)$-time algorithm to find the maximum volume of food Obert can eat given his diet.

Solution:

1. **Subproblems**
   - $x(i, j, s')$: maximum volume of food possible purchasing a suffix of plates p_i to p_{n-1}, using at most j calories, ordering exactly s' sweet plates
   - for $i \in \{0, \ldots, n\}$, $j \in \{0, \ldots, k\}$, $s' \in \{0, \ldots, s\}$
2. **Relate**
   - Either order plate p_i or not. Guess!
   - If order p_i, get v_i in volume but use c_i calories
   - If p_i is sweet, need to order one fewer sweet plate
   - $x(i, j, s') = \max \left\{ \begin{array}{ll} v_i + x(i+1, j - c_i, s' - s_i) & \text{if } c_i \le j \text{ and } s_i \le s', \\ x(i+1, j, s') & \text{always} \end{array} \right\}$
3. **Topo**
   - Subproblems $x(i, j, s')$ only depend on strictly larger i, so acyclic
4. **Base**
   - $x(n, j, 0) = 0$ and any j (no more plates to eat)
   - $x(n, j, s') = -\infty$ for $s' > 0$ and any j (no more plates, but still need to eat sweet)
5. **Original**
   - Solution given by $x(0, k, s)$
6. **Time**
   - # subproblems: $O(nks)$
   - work per subproblem: $O(1)$
   - $O(nks)$ running time

Problem 4. Gokemon Po (adapted from S18 Final)

Kash Etchum wants to play Gokemon Po, a new augmented reality game. The goal is to catch a set of n monsters who reside at specific locations in her town. Monsters must be obtained in a specific order: before Kash can obtain monster m_i, she must have already obtained all monsters m_j for $j < i$. To obtain monster m_i, Kash may either purchase the monster in-game for positive integer c_i dollars, or she may catch m_i for free from that monster's location. If Kash is not at the monster's location, she will have to pay a ride share service to drive her there. The **minimum possible cost** to transport from the location of monster m_i to the location of monster m_j via ride sharing is denoted by the positive integer $s(i, j)$. Given the price lists of in-game purchases and ride sharing trips between all pairs of monsters, describe an $O(n^2)$-time algorithm to determine the minimum amount of money Kash must spend in order to catch all n monsters, assuming that she starts at the location of monster m_1.

Solution:

1. **Subproblems**
   - $x(i, j)$: min cost of catching monsters m_i to m_n starting at location m_j for $j \leq i$
2. **Relate**
   - If at location of monster, catch it for free!
   - Otherwise, either acquire monster m_i by purchasing or ride-sharing to location. Guess!
   - If purchase spend c_i dollars, else need to ride share to m_i from m_j
   - $x(i, j) = \begin{cases} x(i+1, j) & \text{if } j = i \\ \min\{c_i + x(i+1, j), s(j, i) + x(i, i)\} & \text{otherwise} \end{cases}$
3. **Topo**
   - Subproblems $x(i, j)$ only depend on strictly larger $i + j$ so acyclic
4. **Base**
   - $x(n + 1, j) = 0$ for any j (no cost to collect no monsters)
5. **Original**
   - Solution given by $x(1, 1)$
6. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time