Week 2 Recitation

1 This Week's Recap

1. **DP mindset:** define states so that their *dependency graph is acyclic* (e.g., a topological order), then compute in an order that respects dependencies.
2. **Shortest paths in a DAG (even with negative edges):** topologically sort vertices, then for each vertex v take $\text{dist}[v] = \min_{(u,v)}(\text{dist}[u] + \ell(u,v))$.
3. **Subset / bitmask DP:** Example: counting topological orderings with
$$\text{dp}[S] = \sum_{v \in S,\ v \text{ can be last}} \text{dp}[S \setminus \{v\}].$$
4. **String DP on prefixes:** edit distance / LCS use 2D prefix states (first i characters vs first j characters) with $O(1)$ transitions, giving $O(n^2)$-type runtimes.
5. **Faster LCS when one string is small:** compress the DP by tracking, for each prefix of the short string and each length ℓ, the earliest position in the long string achieving LCS length ℓ; support "next occurrence" queries via preprocessing + binary search.
6. **Prefix vs interval DP on sequences:** decide whether you truly need intervals or if prefixes suffice. Example: LIS with state "best increasing subsequence ending at i", yielding an $O(n^2)$ DP.
7. **Interval DP via "last operation":** for "collapse a sequence/string" problems, a common trick is to condition on the *final* merge/deletion. Example: chain matrix multiplication with $\text{cost}[i,j] = \min_{k \in (i,j)} \big(a_i a_k a_j + \text{cost}[i,k] + \text{cost}[k,j]\big)$ in $O(n^3)$.
8. **CFG parsing idea (interval + prefix DP):** for each substring $S[l,r]$, track which single characters it can collapse to; to test a rule $c_i \to S_i$, run a helper DP that matches a prefix of S_i to a partition of $S[l,r]$. Runtime is something like $O(n^3|\mathcal{G}|)$ where $\mathcal{G}$ is the size of the allowed input operations.

2 Another CFG Parsing Example

Problem 1: Palindrome Shrinking***

The input is a string s of length n. In one operation, I can pick any *palindrome* substring of s and delete it from s. Find the minimum number of operations needed to turn s into the empty string, in time $O(n^3)$.

A *palindrome* is a string that is the same forwards and backwards. For example, a, $abba$, and $abcba$ are all palindromes, but $abca$ is not.

Sources:

- `https://codeforces.com/problemset/problem/607/B`
- (earlier cite) `https://vjudge.net/problem/HRBUST-1847`

Solution sketch. We do interval DP. DP$[i, j]$ denotes the minimum number of steps needed to turn the range $s[i,\dots, j]$ into an empty string by deleting palindromes.

Let us understand the DP transition. To do this, consider the leftmost character. Consider the step on which it is deleted.

Case 1: It's deleted by itself. This would lead to $\mathrm{DP}[i, j] \le 1+\mathrm{DP}[i+1, j]$.

Case 2: It's deleted by pairing with $i+1$. This only can happen if $s[i] = s[i+1]$. In this case, $\mathrm{DP}[i, j] \le 1+\mathrm{DP}[i+2, j]$.

Case 3: i is deleted along with k, only can happen if $s[i] = s[k]$. In this case, note that at some point, the range $s[i+1,\dots, k-1]$ was deleted by using palindrome deletions. We can take the last one that was deleted, and append $s[i]$ and $s[k]$ to that deletion (at the front and the back). This leads to the transition

$$\mathrm{DP}[i, j] \le \mathrm{DP}[i+1, k-1]+\mathrm{DP}[k+1, j],$$

only if $s[i] = s[k]$.

The total runtime is $O(n^3)$ - n^2 states and $O(n)$ per state.