Pragmatic Programming

Session 4 - Dynamic Programming

Max Nilsson

16th November

The Idea

- Combine **recursion** with **memoization**.
- The hard parts are
  - Which states should I use for the recursion?
  - How should I store the cached values in memory?
  - How should I iterate through the recursion tree? Top-Down or Bottom-Up? There are pros and cons with both approaches.
- Luckily, in `C++` there is usually not much slowdown associated with multiple recursive calls and there is no pre-determined recursion depth that must be manually changed (looking at you `python`).
- A good thing is that you can almost always easily convince yourself of your algorithms complexity and if it is viable beforehand.
- Even though dp is almost surely overhyped as a buzzword, when it works, it is in my opinion one of the most magical things in algorithmic problem solving.

The Design Pattern I Always Use

```
const int mxn = 500;
int dp[mxn][mxn]; // initialized as -1
int getdp(int i, int j) {
    // return for basecase
    // return if already computed in dp
    int res;
    // compute res with recursive calls to getdp
    dp[i][j] = res;
    return res;
}
```

If the space complexity of `dp` is $\mathcal{O}(n)$ and the recursive calculations of `res` are $\mathcal{O}(m)$ then the overall complexity of `getdp` is (at most) $\mathcal{O}(nm)$.

Live Coding Example

- This theory is really all that is necessary for solving most dynamic programming problems. The rest is really just practice.
- Tip: Look out for small input limits as hints that the problem can be solved with (2-D) dynamic programming.
- Now *I Will* live code Hamming Ellipses!

This Week

- Skim read parts of chapter 9 in Sannemo[1] and chapter 6 in Laaksonen[2].
- Solve half, rounded down, of this weeks 11 problems.

[1] Johan Sannemo. "Principles of Algorithmic Problem Solving". In: *Draft version* (2018).

[2] Antti Laaksonen. *Guide to competitive programming*. Springer, 2020.