DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

Last week we covered interval based dynamic programs, which occur on a sequence, and have intervals as states. These programs are tricky in how states get combined: multiple intervals can be merged to form a single interval, with some characters skipped.

This week we will focus on something ‘easier’, where the only allowed states are prefixes, aka. $DP[i]$ corresponds to indices $[1 \ldots i]$. The transitions of such dynamic programs are also much more restricted: going from $DP[i]$ to $DP[j]$ for some $j > i$ more or less means that indices $i \ldots j$ get assigned to the same interval.

This type of seemingly easier dynamic program become more complicated for two reasons:

1. Transforming problems to this form often requires algebraic manipulations (based on what the desired transition formula looks like). Such transforms sometimes don’t have natural interpretations in terms of the original problem.

2. There is more room to augment prefixes with other states that track what ‘comes out’ of the prefixes. This often leads to complex higher dimensional states.

There is also the additional dimension of the transitions are often computable in (amortized) sublinear time through the use of data structures. We will address this systematically later on this term, but will first give an example of this.

**Problem 3.1.** Given a static list of $n$ integers $x[1 \ldots n]$, for $n$ pairs of indices $l_i \leq r_i$, compute the sums of the corresponding intervals $x[l_i \ldots r_i]$.

Observe that

$$\text{Sum}(x[l \ldots r]) = \text{Sum}(x[1 \ldots r]) - \text{Sum}(x[1 \ldots (l-1)])$$

This means if we store

$$S[i] = \sum_{k=1}^{i} x[k],$$

we can answer the sum of any subinterval in $O(1)$ time. The transition for computing $S[i]$ can also be written as a dynamic program with the base case $S[0] = 0$ and

$$S[i] = S[i-1] + x[i]$$
can be computed in increasing order. Of course, this is a very watered down ‘dynamic program’ as the transitions are all unconditional, but there is always room to extend such things :-). On the problem set there is one that asks for the max value that excludes the interval \([l \ldots r]\), and idea is to combine together the intervals \(1 \ldots (l - 1)\) and \((r + 1) \ldots n\).

The more classical example is longest increasing subsequence:

**Problem 3.2.** Given a list of numbers \(x[1 \ldots n]\), remove the fewest number so that the rest is increasing.

The dynamic program state is \(DP[i]\) being the length of the longest increasing subsequence that ends at location \(i\).

**Problem 3.3.** Given a dictionary with words of total length \(n\), build another string \(s\) by gluing together (without overlapping) the fewest number of words from the dictionary.

The states here are just the fewest number of words we need to glue to form \(s[1 \ldots n]\), and we can build up these states by adding a word to a prefix.

Specifically, we can go from \(s[1 \ldots i]\) to \(s[1 \ldots j]\) if \(s[i + 1 \ldots j]\) is in the dictionary. There are \(O(n^2)\) transitions, and we can check whether each substring is in the dictionary in \(O(n)\) time, for a total running time of \(O(n^3)\).

We can also reduce the runnign time down to \(O(n^2)\) by optimizing the time spent performing the transitions. Instead of checking all pairs \(i < j\), we loop through all the strings in the dictionary instead, and check if each one matches the portion of \(s\) starting from \(i\), \(s[i \ldots n]\). Because the total lengths of the strings is \(n\), this takes \(O(n)\) per state, for a total of \(O(n^2)\).

**Problem 3.4.** You are given a sequence of length \(n\) with entries labeled \(X\), \(\_\) or \(\ldots\). For a value of \(k\) and lengths \(c[1] \ldots c[k]\) given in the input, determine if the \(s\) can be turned into \(X\) and \(\ldots\) appropriately so that there are exactly \(k\) consecutive segments of \(X\)s in the array, and \(j\)th one has length exactly \(c[j]\).

This problem is available at [https://dmoj.ca/problem/ioi16p4](https://dmoj.ca/problem/ioi16p4)

This is a problem where some kind of state augmentation is needed.

We can go paint the prefixes, and remember how many blocks we have encountered, and how many \(X\)s into the current block are we. This leads to \(O(kn)\) states per prefix, for a total runtime of \(O(n^2k)\).

To do better, note that the transitions are uniquely determined if we’re within a block of \(X\)s. So we only need to track whether it’s possible for the \(j\)th block to finish at location \(i\). That is, we define

\[
DP[i][j] = \text{if we can paint things so that the } j\text{th block finish at } i.
\]

The transition must come from \(DP[k][j - 1]\) for some \(k < i\): here we need to reason about the structure of \(s[k + 1 \ldots i]\): the last \(c[j]\) spaces must be turned into \(X\)s, and everything before that need to be turned into \(\ldots\). By playing a bit with configurations, we get:
1. There can’t be $s$ in the last $C[j]$ cells.

2. There cannot be a $X$ in locations $k + 1 \ldots i - c[j]$.

With prefix sum structures, these can be checked in $O(1)$ time. On the other hand, the first condition is just a true or false, while the second condition gives that we can check use any $k$ to the right of the first $X$ before $i - c[j]$. When we increase on $i$, this value must be only increasing. So we have another two-pointer sweep situation, and the whole table filling can be done in $O(nk)$ time.

A more complicated prefix based DP (from [https://codeforces.com/gym/102392/problem/B](https://codeforces.com/gym/102392/problem/B)):

**Problem 3.5.** There are $n$ quests in a game, each of which can be completed at most once (but can be done in any order).

You’d like to complete some subset of them in order to level up twice, the first time needing $s_1$ experience, the second time needing a total of $s_2 > s_1$ experience.

Each quest gains $x_i$ experience when completed at level 1, and $y_i \leq x_i$ experience when completed at level 2. It also takes time $t_i \geq r_i$ respectively.

Find the shortest amount of time needed to get a total of $s_2$ experience in $O(n \cdot s_1 \cdot s_2)$ time.

Here it’s unclear how, or why we should order the states.

At a first glance, what’s needed is a 2-dimensional state akin to knapsack: amount of experience gained in level 1, and amount gained during level 2.

Note that the main difficulty is that we are ‘forced’ to level up once a total of $s_1$ experience has been gained. So there is advantage in using a higher experienced event to ‘jump over’ the bar here: if we crossed at at a total of $X$ exp, the last event need to have $X - x_i < s_1$, which means the larger the $x_i$ the better.

So we need to sort the quests in increasing order of $x_i$, and do DP on the states corresponding to minimum total time needed to get to a total of $z_1$ EXP on level 1, and $z_2$ EXP on level 2.

Number guessing also fits this type of questions:

**Problem 3.6.** We play the usual ‘guess the number’ game, except there is total budget of $x$, and bigger/smaller costs $y$ and $z$ respectively (for $y \neq z$). Determine whether one can guess correctly the number between $[1, n]$ under such a budget.

There are two ways of building states based on this, one based on the length of the intervals remaining, the other based on the budget remaining.

For the first one, we let $MinBudget[m]$ denote the minimum budget needed to find a number in an interval of length $n$. Then we need to enumerate all possible query points $q$, giving the recurrence

$$MinBudget[m] = \min_{1 \leq q \leq m} \max \{MinBudget[q - 1] + x, MinBudget[q + 1] + y\}$$
with some special cases for empty intervals (which couldn’t happen as $a$ has to be in there).

Alternatively we can track the max length that can be answered with a budget of $x$. The recurrence is actually simpler: we simply maximize the lengths of the two sides under the remaining budget:

$$
MaxBudget[x] = 1 + MaxBudget[x - y] + MaxBudget[x - z].
$$

We finish with a problem on the deep end of things. Prefix based dynamic programs get really really hard.

**Problem 3.7.** Given a length $n$ permutation, you want to sort it with minimum total cost using the following operation: given indices $i$ and $j$, move $i$ to position $j$, while keeping relative order of other entries, at a cost of $i + j$. 