NOI.PH Training: DP 3

Advanced DP patterns

Kevin Charles Atienza

Contents

1 Introduction

Dynamic programming is very flexible! It can be applied anywhere there's a need to store and reuse previous results, hence it can be combined with anything: data structures, graphs, math, etc.

In this document, we will be discussing more DP patterns, and other ways of thinking about DP.

A few notes:

- This module is mostly aimed at returning students, so first-time students, don't worry too much if you get lost at some point! But you should still generally be able to follow.
- This assumes that you have read the DP 1 and DP 2 modules.
- *Tip*: To get the most of the lessons, we recommend doing every Exercise, since they try to illustrate/teach a point about the topic being discussed.

2 States and transitions

As you have learned, dynamic programming is all about solving a problem by reducing it to a similar but smaller problem that's "one step lower". As we consider and discover more advanced DP techniques, this basic fact will remain true. However, there are other ways of thinking about DP.

In this section, we will discuss a perspective on DP in terms of *states*, and *transitions* between them.

Note: This is not a fundamentally new idea, but having a different perspective will probably not hurt; in fact, it may potentially help you come up with a recurrence easier than with other perspectives.

2.1 Longest common subsequence

The idea of **state** is essentially just "everything you need to know to describe something". So "state" is just a description of the thing you're considering, and should ideally be self-contained.

To illustrate this idea, let's consider the standard DP-solvable problem **longest common subsequence**, or LCS.

Problem 2.1. Given two words, s and t, find the length of the longest word w that is a subsequence of both s and t.

2.1.1 A solution

Now, you can probably solve this problem with DP by finding a recurrence for the function $\ell(i,j)$, which is defined as the longest common subsequence of the prefixes $s_{0\ldots i}$ and $t_{0\ldots j}$. (We will use zero-indexing.) By definition, the answer is just $\ell(|s|,|t|)$. Now, with the usual way of DP thinking that you may have learned, you can probably arrive at a recurrence like this, after a bit of reasoning on the different cases:

$$\ell(i,j) = \begin{cases} \max(\ell(i,j-1), \ell(i-1,j), \ell(i-1,j-1)) & \text{if } s_i \neq t_j \\ \ell(i-1,j-1)+1 & \text{if } s_i = t_j. \end{cases}$$

Exercise 2.1. Prove the above recurrence for $\ell(i,j)$.

Exercise 2.2. Show that we can slightly simplify the recurrence into

$$\ell(i,j) = \begin{cases} \max(\ell(i,j-1), \ell(i-1,j)) & \text{if } s_i \neq t_j \\ \ell(i-1,j-1)+1 & \text{if } s_i = t_j. \end{cases}$$

Exercise 2.3. We have omitted the base cases for $\ell(i,j)$. What are they?

Using this recurrence, we can solve the problem in $\mathcal{O}(|s||t|)$ by computing a table of $\ell(i,j)$ for all possible i and j. (This table can be a 2D array or a map, depending on your implementation.)

2.1.2 An alternative derivation

Although the solution above is perfectly fine, in this section we shall describe an alternative way of coming up with DP recurrences.

To solve the problem, let's think about how we can mechanistically construct the LCS, say w, from scratch. The most obvious way is to construct w *letter by letter*, from left to right.

- What should the first letter, w_0, be? Well, w_0 can be anything, as long as it can be matched in both words s and t. So our choice w_0 has to be present in both words. If there are many occurrences of w_0, then we have to choose which ones we want to match w_0 with. Say we want to match w_0 to s_i and t_j.
- Now, we need to choose the second letter, w_1. We can play the same game and choose w_1 however we want, but now we have a few new constraints based on our choice of w_0. Note that by matching w_0 to s_i and t_j, we limit the letters that can be matched to w_1 (and in fact, all subsequent w_k too) to the suffixes $s_{i+1,\ldots}$ and $t_{j+1,\ldots}$. So somehow, we need to remember our past choices when we want to continue the process.
- ...

It should not be hard to continue this line of reasoning.

Now, before looking for more insights, let's stop here for now and think about whether we have enough information to construct an algorithm to find the LCS. Don't worry about about the running time yet. Note that, while playing our game above, we needed to remember our past choices, so in effect, we have a certain *state* which represents everything we have done so far, since this will affect our future choices. In this case, our state will be three strings:

$$(w_{0,1\ldots,\ell-1},\ s_{i_0,i_1,\ldots,i_{\ell-1}},\ t_{j_0,j_1,\ldots,j_{\ell-1}}),$$

where:

- $w_{0,1\ldots,\ell-1}$ is the common subsequence we've constructed so far, with ℓ being its length.
- $s_{i_0,i_1,\ldots,i_{\ell-1}}$ is the subsequence of s corresponding to w.
- $t_{j_0,j_1,\ldots,j_{\ell-1}}$ is the subsequence of t corresponding to w.

Now, given such a state, our next step is to select the next letter, w_ℓ. Such a choice *transitions* us to a new state. The transition itself can be encoded into a recurrence that allows us to solve the problem with recursion. To do so, let's define the function $f(S)$ as: "given the current state S, what is the longest common subsequence you can form by starting from state S?"

The recurrence for $f(S)$ can be expressed by encoding what it means to select the next letter w_ℓ, for example, like this:

```
// Warning: this code is horribly inefficient!

string s, t;
// our state is, as expressed above,
// (w[0..l-1], s[i[0],...,i[l-1]], t[j[0],...,j[l-1]])
int f(int l, string w, vector<int> i, vector<int> j) {
    // extend the size of w, i, j by one
    w.push_back();
    i.push_back();
    j.push_back();
    int ans = l;
    for (w[l] = 'a'; w[l] <= 'z'; w[l]++) {
        for (i[l] = l==0 ? 0 : i[l-1]+1; i[l] < s.length(); i[l]++) {
            if (w[l] == s[i[l]]) {
                for (j[l] = l==0 ? 0 : j[l-1]+1; j[l] < t.length(); j[l]++) {
                    if (w[l] == t[j[l]]) {
                        ans = max(ans, f(l + 1, w, i, j));
                    }
                }
            }
        }
    }
    return ans;
}
```

The *initial state* represents the beginning of the construction, and *terminal states* represent the end. Thus, the initial state in our case is $(\varepsilon, \varepsilon, \varepsilon)$, where ε represents the empty string, and the terminal states are the base cases, which is when there are no more common letters in both strings.

Now, this is definitely a valid recurrence that reduces the problem to smaller subproblems. However, there's one big problem: the states are too big! Even by memoizing the function above, the state $S = (w_{0,1\dots}, s_{i_0,i_1\dots}, t_{j_0,j_1\dots})$ is simply too big, and there's very little overlap in the states since we're essentially remembering the whole history of our choices!

But let's think about it a bit: do we really need to remember all our past choices? Actually, we don't! Taking a closer look at the above code, we see that there are only a few things that can affect our future choices and answer, namely:

- `l`, or ℓ, for the answer,
- `i[l-1]` to determine the starting point for `i[l]`, and
- `j[l-1]` for similar reasons.

Therefore, we can vastly simplify our recurrence by simplifying our state to just three numbers: (ℓ, i, j), where i and j are the indices of the last letter that we chose in s and t, respectively. We don't even need to remember the word we're constructing!

With this new state, we can define[1] $f(S) = f(\ell, i, j)$ this way:

```
string s, t;
int f(int l, int i, int j) {
    int ans = l;
    for (char W = 'a'; W <= 'z'; W++) {
        for (int I = l==0 ? 0 : i+1; I < s.length(); I++) {
            if (W == s[I]) {
                for (int J = l==0 ? 0 : j+1; J < t.length(); J++) {
```

[1] Note that there's a slight abuse of notation here; $S = (\ell, i, j)$, so we really should be writing it as $f((\ell, i, j))$. But it's too many parentheses, so we will abuse notation.

```
                    if (W == t[J]) {
                        ans = max(ans, f(l + 1, I, J));
                    }
                }
            }
        }
    }
    return ans;
}
```

Our initial state could be $(0, -1, -1)$.

Exercise 2.4. What does -1 mean in this context?

So the thing we did above is one way of thinking about states and transitions: the *state* of the DP solution is our *memory*, the collection of everything we need to remember in order to make future choices.

Memoizing this function is now effective since the number of states, (ℓ, i, j), is not that big; in this case, there are $\mathcal{O}(|s||t|\min(|s|, |t|))$ states. We can now estimate the running time of the memoized version as

$$\begin{aligned}&\mathcal{O}((\text{number of states}) \cdot (\text{running time of transition}))\\ &= \mathcal{O}(|s||t|\min(|s|, |t|)) \cdot \mathcal{O}(\alpha|s||t|)\\ &= \mathcal{O}(\alpha|s|^2|t|^2\min(|s|, |t|),\end{aligned}$$

which is now clearly polynomial (in $|s|$, $|t|$ and the alphabet size α).

Note that this is still quite far from the the $\mathcal{O}(|s||t|)$ solution we had earlier, but we can get there by using a few observations and insights.

Exercise 2.5. Show that we can use an alternative state (ℓ, i', j'), where $i' = i+1$ and $j' = j+1$. What do i' and j' mean in this context?

Also, what is the initial state?

2.1.3 Optimizing the solution

Improving a DP solution usually consists of two parts:

- Making the state size smaller.
- Making the transition step faster.

Let's tackle both in turn.

For the first part, we have a three-integer state (ℓ, i', j'). Can we reduce it further? It turns out that we can! Notice that we can remove ℓ, i.e., we can just use the state (i', j'). The reason is that it doesn't really affect our decision-making; although ℓ affects the output of the function, it only adds itself at the end, to the output of what we actually did without it.

This means that we can pass the responsibility of "adding ℓ" to the one who called the function, since they will be the one who will need it.

The main lesson here is to *only remember the things that affect your decisions, not results.*[2]

However, since ℓ is no longer available, $f(\ell, i', j')$ cannot be computed as originally defined. Thus, we need to define a new function, $f'(i', j')$, that has a slightly different output.

Exercise 2.6. What would be the output of the alternative function, $f'(i', j')$?

For the second part, we want to optimize the transition step. There are a couple of inefficiencies in our code above:

- First, we're matching a given letter `W` across all its occurrences in s and t, but in fact, we can just choose the *leftmost* occurrence.
- Next, we are looping across the alphabet with `W`, but actually, we don't have to do that. We can just select `I` first and then choose `W` to be `s[I]`.

Exercise 2.7. Explain why these are true.

Finally, we can improve the transition step by coming up with a *better* one. Note that the transition step above considers the next letter of the common subsequence w, but as it turns out, this idea results in a slow transition, since we have to find a common letter in both s and t which seems to require loops.

To improve it, we should notice that there's an alternative way of *constructing* any common subsequence, and it involves a sequence of three very simple actions:

- Remove the first letter of s.
- Remove the first letter of t.
- If the first letters of s and t are equal, then add that letter to the subsequence, and remove both first letters.

Exercise 2.8. Show that any sequence of these steps produces a common subsequence. Also, show that any common subsequence can be obtained from this process.

Here, we can clearly see that what we're *really* attempting to construct is a sequence of these commands, and we also note that a two-integer state like (i', j') captures "everything we need to keep in our memory" to perform any future actions.

Even better, since there are only three possible actions, the transition step is $\mathcal{O}(1)$!

Exercise 2.9. Show that we can drastically improve the transition to the following:

$$f'(i', j') = \begin{cases} \max(f'(i'+1, j'), f'(i', j'+1)) & \text{if } s_{i'} \neq t_{j'} \\ f'(i'+1, j'+1) + 1 & \text{if } s_{i'} = t_{j'}. \end{cases}$$

Now, notice that the latter exercise shows that we end up with basically the same recurrence as earlier (except that we're working with suffixes now), but we derived it with a totally different perspective.

[2] Maybe there's a life lesson here somewhere?

In summary, we can think of DP in terms of states and transitions by coming up with a way to construct the thing we're looking for step by step, and by remembering only the things we need to remember at every point to make our future choices (a.k.a., the *states*), and determining how to *transition* from one state into the other. The running time is roughly

$$\mathcal{O}((\text{number of states}) \cdot (\text{running time of transition})),$$

so we get a better solution by:

- reducing the total number of distinct states; alternatively, by maximizing the *overlaps* between subproblems (to make memoization effective), and
- optimizing the transition.

Whether you find this perspective easier to grasp than others depends on the way you think, but in any case, having more perspectives could potentially help you derive DP recurrences more easily. It is also a bit dependent on the problem; e.g., some problems may be easier to think about in terms of states and transitions, and some with other perspectives.

2.2 Partitions

Let's consider another example.

Problem 2.2. A **partition** of an integer n is a way of writing n as a sum of positive integers, where the *order doesn't matter.* Hence, 4 has 5 distinct partitions:

- 4,
- $1 + 3$ which is the same as $3 + 1$ (because order doesn't matter),
- $2 + 2$,
- $1 + 1 + 2$ which is the same as $1 + 2 + 1$ and $2 + 1 + 1$,
- $1 + 1 + 1 + 1$.

Let $p(n)$ be the number of distinct partitions of n, e.g., $p(4) = 5$. Given n, compute $p(n)$.

You begin to attempt to find a recurrence for $p(n)$, but you soon find this difficult. To begin, consider forming a partition of n. Now, you might think, "well, we can choose the last summand s to be anything from 1 to n, and what's left is a partition of $n - s$," and then you might say that by summing, we have

$$p(n) \overset{?}{=} \sum_{1 \le s \le n} p(n - s).$$

Unfortunately, this is wrong since it fails into account the fact that order doesn't matter, so for example it counts $1 + 2 = 2 + 1$ twice!

After a bit of trial and error, you might conclude that finding a recurrence purely in $p(n)$ is somewhat difficult.[3] And indeed, there's some truth to this; it seems that there's something "missing" in the function $p(n)$ that you somehow need to finish that recurrence.

[3] Of course, there's the pentagonal number theorem, which gives a pure recurrence in $p(n)$, but this is a highly nontrivial result found by Euler using somewhat advanced techniques, well beyond what we're trying to go for here.

Instead, let's think constructively. How do we systematically form a partition in a way that every distinct partition is enumerated only once? The trouble is that something like $1+3+1+2$ has many many different representations, and we only want to count it once. Well, the simplest way is to *sort* the partition! The reason is clear: two partitions are the same iff they have identical sorted versions.

Armed with this, we now attempt to enumerate the sorted partitions of n. Being constructive, let's construct a partition step by step, from left to right, say $n = s_0 + s_1 + \ldots + s_{k-1}$.

- What would s_0 be? It can be anything from 1 to n. Cool.
- Next, what would s_1 be? Well, this time, we want $s_0 \le s_1$ to make the partition sorted. But we also want $s_0 + s_1 \le n$ since we want the s_is to eventually add up to n.

  Thus, s_1 can be anything from s_0 and $n - s_0$.
- Next, what would s_2 be? Similarly, we note that it can be anything from s_1 to $n - s_0 - s_1$.
- Next, what would s_3 be? Similarly, we note that it can be anything from s_2 to $n - s_0 - s_1 - s_2$.
- etc. In general, s_ℓ can be anything from $s_{\ell-1}$ to $n - s_0 - s_1 - \ldots - s_{\ell-1}$.
- If, at any point, $n - s_0 - s_1 - \ldots - s_{\ell-1}$ becomes zero, then that means we have successfully constructed a partition of n, so we stop.

Again, we can use the state $(s_{0,1,\ldots,\ell-1})$ representing all the choices we've made so far, and the idea above gives us a way to transition to the next state by choosing s_ℓ. However, this state is too big, and as before, we don't need to remember our whole history to make our future choices.[4] Our decision for s_ℓ only depends on two things:

- $s_{\ell-1}$, which is the *last* number we've chosen, and
- $n - s_0 - s_1 - \ldots - s_{\ell-1}$, which is the *remaining* sum we have to make.

That's it! As we chug along, constructing some partition, we don't need to remember a lot of the things we did. We don't even need to remember ℓ, the length of our current partition!

Armed with this, let's use the smaller state of just two integers: $(\mathit{last}, \mathit{remain})$, to represent the two values above. Then we get the following recurrence:

```
int f(int last, int remain) {

    if (remain == 0) return 1;  // terminal state/base case

    int ans = 0;
    for (int curr = last; curr <= remain; curr++) {
        ans += f(curr, remain - curr); // if we select the next element as
                                       // 'curr', then there will be
                                       // 'remain - curr' left to fill.
    }
    return ans;
}
```

[4] Sorry, Rizal!

or, as expressed in math notation,

$$f(\ell, r) = \begin{cases} 1 & \text{if } r = 0 \\ \sum_{c=\ell}^{r} f(c, r - c) & \text{otherwise.} \end{cases}$$

Exercise 2.10. What is the initial state?

This gives us an $\mathcal{O}(n^3)$ algorithm to compute $p(n)$ (assuming you memoize it, of course), and that's cool.

Exercise 2.11. Show that the algorithm above runs in $\mathcal{O}(n^3)$.

Hint: $\mathcal{O}(n^3) = \mathcal{O}(n^2) \cdot \mathcal{O}(n)$.

Note that $\mathcal{O}(n^3)$ is not the best solution we can have for this problem; we shall see how to improve it later.

3 Linear recurrences and matrix exponentiation

Not all DP solutions are equal. As we have seen before, different states/transitions yield DP solutions of differing efficiencies. We always want to find a DP solution that's as fast as possible, or at least fast enough to fit the constraints of the problem.

In general, optimizing DP solutions boils down to practice (lots of it!) and creative thinking. However, in some occasions, optimizing the DP is a bit more straightforward, since certain classes of DP solutions have a few well-known techniques in optimizing them.

In some circumstances, we can even improve the running time of our DP solution by **a lot**. We will see this really soon. In this section, we will describe a class of problems that share a particular optimization technique: those whose transitions are *linear recurrences.*

Note: Most problems in this section involve counting and/or mostly enormous numbers, so computations are assumed to be done modulo some number like $10^9 + 9$.

3.1 Illustrative examples

3.1.1 String counting 1

Let's start with a simple example.

> **Problem 3.1.** Suppose our alphabet is the set of three letters: `N`, `O`, `I`. Given n, how many n-letter words are there from this alphabet?

This is clearly a combinatorics question, but let's pretend we don't know how to combinatorics this problem. Suppose we want to approach this nail with DP, our only hammer in this module. What would that mean?

Again, let's try to construct such a word $w_{0\ldots,n-1}$, letter by letter, from left to right.

- What would w_0 be? It can be either `N`, `O`, or `I`.
- What would w_1 be? It can be either `N`, `O`, or `I`.
- What would w_2 be? It can be either `N`, `O`, or `I`.
- etc.

This doesn't sound too complicated since our past choices don't seem to affect any of our future choices! This means that we don't even need to remember anything about our past choices. We only need to remember a single integer i, the remaining number of letters to be chosen.[5] Using i, we found out that

$$f(i) = f(i-1) + f(i-1) + f(i-1),$$

or simply

$$f(i) = 3 \cdot f(i-1),$$

with terminal state $f(0) = 1$ since there is exactly one empty string. Now, $f(n)$ can be computed with DP in $\mathcal{O}(n) \cdot \mathcal{O}(1) = \mathcal{O}(n)$. Done!

[5] You can also consider only remembering the state i', the index of the last letter we've chosen, but choosing i this way is more elegant. You'll see why soon, and once you understand, you'll get used to it.

... but, I feel you. I know you're not particularly happy with this solution. After all, if we look closer, it's fairly easy to see that $f(n)$ is just 3^n, and we can even compute $f(n)$ in $\mathcal{O}(\log n)$ time using *fast exponentiation*![6] This tells us that there's something deeper going on. At the very least, it tells us that some recurrences can be optimized further.

3.1.2 String counting 2

Now, let's solve a slightly harder problem.

> **Problem 3.2.** Using the same alphabet `NOI`, how many strings of length n are there that do not contain the substring `OO`?

This time, we can't find a closed form solution like before. But DP is flexible, so let's construct our `OO`-avoiding string, again from left to right:

- What would w_0 be? It can be `N`, `O`, or `I`.
- What would w_1 be? It depends on w_0. If w_0 is `O`, it can only be `N` or `I`, otherwise, it can be anything: `N`, `O`, or `I`.
- What would w_2 be? It depends on w_1. If w_1 is `O`, it can only be `N` or `I`, otherwise, it can be anything: `N`, `O`, or `I`.
- etc.

This time, past choices affect our future choices, but only barely—we only need to remember the last letter! Thus, our state could be (c, i) where i is the number of letters that remain to be chosen, and c is the last letter chosen.

> **Exercise 3.1.** What is the initial state? Are we forced to invent a special state like $(\texttt{'N/A'}, n)$?

Using this, we now arrive at our decision-making for the next letter to choose:

```
int f(char c, int i) {
    if (i == 0) return 1;

    int ans = 0;
    ans += f('N', i-1);                 // choosing 'N'
    if (c != 'O') ans += f('O', i-1);  // choosing 'O'
    ans += f('I', i-1);                 // choosing 'I'

    return ans;
}
```

Or, in math terms,

$$\begin{aligned} f(\texttt{N}, i) &= f(\texttt{N}, i-1) + f(\texttt{O}, i-1) + f(\texttt{I}, i-1) \\ f(\texttt{O}, i) &= f(\texttt{N}, i-1) + f(\texttt{I}, i-1) \\ f(\texttt{I}, i) &= f(\texttt{N}, i-1) + f(\texttt{O}, i-1) + f(\texttt{I}, i-1) \end{aligned}$$

[6] The complexity $\mathcal{O}(\log n)$ only applies if we're computing $f(n)$ modulo some small m; otherwise, even just writing $f(n)$ takes $\mathcal{O}(n)$ already. In general, you should not ignore the size of numbers when they can grow arbitrarily big.

Again, there are $\mathcal{O}(3 \cdot n) = \mathcal{O}(n)$ states, and the transition is $\mathcal{O}(1)$, so the running time of the DP is $\mathcal{O}(n)$. But can we improve this, similar to how we improved the computation of the previous example to fast exponentiation, 3^n?

It turns out that we can! However, instead of numbers, we're exponentiating something else. To derive it, let's consider what happens above. We have three values

$$\begin{bmatrix} f(\texttt{N}, i) & f(\texttt{O}, i) & f(\texttt{I}, i) \end{bmatrix}$$

and somehow, we can determine them from only the values

$$\begin{bmatrix} f(\texttt{N}, i-1) & f(\texttt{O}, i-1) & f(\texttt{I}, i-1) \end{bmatrix}$$

using the formulas above. But note that the equations above are *linear* (think systems of linear equations), which means that we can encode the three formulas above compactly using the following matrix[7] identity

$$\begin{bmatrix} f(\texttt{N}, i) \\ f(\texttt{O}, i) \\ f(\texttt{I}, i) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(\texttt{N}, i-1) \\ f(\texttt{O}, i-1) \\ f(\texttt{I}, i-1) \end{bmatrix}$$

and this is true for every $n > 0$.

Exercise 3.2. Verify the above identity.

By expanding this iteratively, we see that

$$\begin{aligned}
\begin{bmatrix} f(\texttt{N}, n) \\ f(\texttt{O}, n) \\ f(\texttt{I}, n) \end{bmatrix} &= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(\texttt{N}, n-1) \\ f(\texttt{O}, n-1) \\ f(\texttt{I}, n-1) \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \left(\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(\texttt{N}, n-2) \\ f(\texttt{O}, n-2) \\ f(\texttt{I}, n-2) \end{bmatrix} \right) \\
&= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}^2 \cdot \begin{bmatrix} f(\texttt{N}, n-2) \\ f(\texttt{O}, n-2) \\ f(\texttt{I}, n-2) \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}^2 \cdot \left(\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(\texttt{N}, n-3) \\ f(\texttt{O}, n-3) \\ f(\texttt{I}, n-3) \end{bmatrix} \right) \\
&= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}^3 \cdot \begin{bmatrix} f(\texttt{N}, n-3) \\ f(\texttt{O}, n-3) \\ f(\texttt{I}, n-3) \end{bmatrix} \\
&\vdots \\
\begin{bmatrix} f(\texttt{N}, n) \\ f(\texttt{O}, n) \\ f(\texttt{I}, n) \end{bmatrix} &= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} f(\texttt{N}, 0) \\ f(\texttt{O}, 0) \\ f(\texttt{I}, 0) \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}
\end{aligned}$$

This means that we can compute $f(\texttt{N}, n)$ (and as a bonus, also $f(\texttt{O}, n)$ and $f(\texttt{I}, n)$) in $\mathcal{O}(\log n)$ time by fast-exponentiating the 3×3 matrix above!

[7] Please feel free to ask in Discord if you are not familiar with matrices. Also, try watching this amazing series on linear algebra (where matrices come from) by 3Blue1Brown.

Exercise 3.3. Show that we can reduce the matrix size to just 2×2 by using the state (*last char is O*, i), where "*last char is O*" is a boolean representing whether the last character we chose is an `O` or not. Explain the $\mathcal{O}(\log n)$ algorithm.

Note that there's nothing special with `OO`; the same approach will work if, say, we want to find the number of strings that don't contain the substring `IOI`. This time, we need to remember the last two letters, giving us a 9×9 matrix, for the 9 values:

$$[f_{\texttt{NN}}(i) \quad f_{\texttt{NO}}(i) \quad f_{\texttt{NI}}(i) \quad f_{\texttt{ON}}(i) \quad f_{\texttt{OO}}(i) \quad f_{\texttt{OI}}(i) \quad f_{\texttt{IN}}(i) \quad f_{\texttt{IO}}(i) \quad f_{\texttt{II}}(i)]\,.$$

(I write $f(\texttt{X}, i)$ as $f_{\texttt{X}}(i)$ for brevity.)

Exercise 3.4. Reduce the size of the 9×9 transition matrix for the `IOI`-problem by minimizing the number of distinct states we remember as much as possible. What is the smallest you can make the matrix?

Exercise 3.5. Let's generalize the problem: Suppose we have an alphabet of size α, and suppose we are given a word w and an integer n. Find an algorithm that computes the number of words that don't contain w as a substring:

1. Suppose our state is simply the last $|w| - 1$ letters we've selected. Show that this leads to an $\mathcal{O}(\alpha^{3|w|} \log n)$ algorithm. What is the size of the matrix?
2. Improve this algorithm by reducing the size of the matrix as much as possible. What is the smallest matrix you can get? What is the time complexity?

Hint: Multiplying two $k \times k$ matrices naively takes $\mathcal{O}(k^3)$.

Exercise 3.6. Suppose that we want to compute the number of `NOI`-strings of length n that *do* contain `OO` as a substring. Devise a matrix-exponentiation algorithm that computes this in $\mathcal{O}(\log n)$ time.

For this exercise, you need to solve the *directly*, i.e., by coming up with a proper state and transition matrix, instead of computing it as

$$3^n - (\text{number of strings that don't contain } \texttt{OO}).$$

Note that there's nothing wrong with the latter solution; we just want to train you with the state and transition approach.

Hint: You only need a 3×3 matrix.

Exercise 3.7. Find a recurrence for the number of tilings[a] of a $3 \times 2n$ grid with dominoes.

[a] In a tiling, each cell must be covered by exactly one tile.

Exercise 3.8. Use a 3×3 transition matrix to solve Exercise 3.7 quickly. Explain your states.

3.1.3 String counting 3

Let's move on to a slightly different problem.

Problem 3.3. Our alphabet is still `NOI`, but suppose that we give *weights* to the letters, say,

- The weight of `N` is 1,
- The weight of `O` is 2,
- The weight of `I` is 3.

Given n, how many words of *weight* n do not have the substring `OO`?

This time, the *length* of the string is not fixed anymore, but that's okay, let's just attempt to construct the string. If we use our constructive approach earlier, we find that:

- As before, our decision depends on the last letter we've chosen; the next letter can be anything, except if the previous one is `O` in which case it can't be `O`.
- We also need to keep track of the *remaining available weight* that we have. Selecting some letter decreases this by a particular amount.
- We stop when the remaining weight becomes 0.

Thus, we can have our state to be (c, i), where c is the last character we've chosen, and i is the remaining weight that we have. As before, we can derive the following recurrences:

$$\begin{aligned}
f_{\mathtt{N}}(i) &= f_{\mathtt{N}}(i-1) + f_{\mathtt{O}}(i-2) + f_{\mathtt{I}}(i-3)\\
f_{\mathtt{O}}(i) &= f_{\mathtt{N}}(i-1) + f_{\mathtt{I}}(i-3)\\
f_{\mathtt{I}}(i) &= f_{\mathtt{N}}(i-1) + f_{\mathtt{O}}(i-2) + f_{\mathtt{I}}(i-3).
\end{aligned}$$

Note that these are *linear* recurrences again, so you might think that the matrix approach would work again. Not so fast! We have a problem, which is that the ith vector

$$\begin{bmatrix} f_{\mathtt{N}}(i) & f_{\mathtt{O}}(i) & f_{\mathtt{I}}(i) \end{bmatrix}$$

is not completely described anymore by the previous vector

$$\begin{bmatrix} f_{\mathtt{N}}(i-1) & f_{\mathtt{O}}(i-1) & f_{\mathtt{I}}(i-1) \end{bmatrix},$$

which means we can't find a relation of this form

$$\begin{bmatrix} f_{\mathtt{N}}(i) \\ f_{\mathtt{O}}(i) \\ f_{\mathtt{I}}(i) \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \cdot \begin{bmatrix} f_{\mathtt{N}}(i-1) \\ f_{\mathtt{O}}(i-1) \\ f_{\mathtt{I}}(i-1) \end{bmatrix}.$$

Sad!

But let's not give up yet. Maybe we just need to look at the recurrences above and just *add* everything we need, say:

$$\begin{bmatrix} f_{\mathtt{N}}(i) \\ f_{\mathtt{O}}(i) \\ f_{\mathtt{I}}(i) \\ ? \\ ? \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \end{bmatrix} \cdot \begin{bmatrix} f_{\mathtt{N}}(i-1) \\ f_{\mathtt{O}}(i-1) \\ f_{\mathtt{I}}(i-1) \\ \color{blue}{f_{\mathtt{O}}(i-2)} \\ \color{blue}{f_{\mathtt{I}}(i-3)} \end{bmatrix} \begin{matrix} \\ \\ \\ \color{blue}{\leftarrow} \\ \color{blue}{\leftarrow} \end{matrix}$$

Now, remember that we want our matrix to move everything "one step forward", which means that we must put the corresponding *next* values on the left side, like

$$\begin{array}{c} \\ \\ \\ \rightarrow\\ \rightarrow \end{array}\begin{bmatrix} f_{\mathtt{N}}(i)\\ f_{\mathtt{O}}(i)\\ f_{\mathtt{I}}(i)\\ f_{\mathtt{O}}(i-1)\\ f_{\mathtt{I}}(i-2)\end{bmatrix} \overset{?}{=} \begin{bmatrix} 1 & 0 & 0 & 1 & 1\\ 1 & 0 & 0 & 0 & 1\\ 1 & 0 & 0 & 1 & 1\\ ? & ? & ? & ? & ?\\ ? & ? & ? & ? & ? \end{bmatrix} \cdot \begin{bmatrix} f_{\mathtt{N}}(i-1)\\ f_{\mathtt{O}}(i-1)\\ f_{\mathtt{I}}(i-1)\\ f_{\mathtt{O}}(i-2)\\ f_{\mathtt{I}}(i-3)\end{bmatrix}.$$

Note that we've also filled in the values of the matrix that we should already know.

But now we have a problem: we have these new terms on the left, $f_{\mathtt{O}}(i-1)$, which also need to be computed. And if we use the recurrence, we see that $f_{\mathtt{O}}(i-1)$ is dependent on $f_{\mathtt{I}}(i-4)$, which is not on the righthand vector yet!

So, you say, why not just add again the new things that we need on our vector? Well, we can consider it, but this will again introduce new terms on the left that need to be computed, and then they will in turn need a few terms we still don't have, and then we'd never stop adding new terms in the vectors!

But not all hope is lost. In fact, we don't have to keep adding more and more things in our vector if we just use *the correct state*. For example, consider the following:

$$\begin{bmatrix} f_{\mathtt{N}}(i)\\ f_{\mathtt{O}}(i)\\ f_{\mathtt{O}}(i-1)\\ f_{\mathtt{I}}(i)\\ f_{\mathtt{I}}(i-1)\\ f_{\mathtt{I}}(i-2)\end{bmatrix} \overset{?}{=} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1\\ 1 & 0 & 0 & 0 & 0 & 1\\ ? & ? & ? & ? & ? & ?\\ 1 & 0 & 1 & 0 & 0 & 1\\ ? & ? & ? & ? & ? & ?\\ ? & ? & ? & ? & ? & ? \end{bmatrix} \cdot \begin{bmatrix} f_{\mathtt{N}}(i-1)\\ f_{\mathtt{O}}(i-1)\\ f_{\mathtt{O}}(i-2)\\ f_{\mathtt{I}}(i-1)\\ f_{\mathtt{I}}(i-2)\\ f_{\mathtt{I}}(i-3)\end{bmatrix}$$

Again, we've filled in the rows of the matrix corresponding to the three original values, using our three recurrences. But what about the rest? It turns out that we don't have to do much, since they're already there, on the right! We just have to pick them out, like this:

$$\begin{bmatrix} f_{\mathtt{N}}(i)\\ f_{\mathtt{O}}(i)\\ f_{\mathtt{O}}(i-1)\\ f_{\mathtt{I}}(i)\\ f_{\mathtt{I}}(i-1)\\ f_{\mathtt{I}}(i-2)\end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1\\ 1 & 0 & 0 & 0 & 0 & 1\\ 0 & 1 & 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 0 & 0 & 1\\ 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_{\mathtt{N}}(i-1)\\ f_{\mathtt{O}}(i-1)\\ f_{\mathtt{O}}(i-2)\\ f_{\mathtt{I}}(i-1)\\ f_{\mathtt{I}}(i-2)\\ f_{\mathtt{I}}(i-3)\end{bmatrix}$$

We now have a complete 6×6 matrix that we can fast-exponentiate to solve the problem in $\mathcal{O}(6^3 \log n) = \mathcal{O}(\log n)$ time![8]

The lesson here is that: if you somehow have a recurrence where you need the value $g(i-k)$ somehow (for some fixed k), then you need to remember all of

$$g(i-1), g(i-2), \ldots, g(i-k)$$

in your state.

The key that makes matrix exponentiation techniques work are the following:

- All our recurrences are linear, i.e., everything depends on previous values with only constant coefficients. (Note that they can be negative.)

[8] We simply "ignored" 6^3 here since it's just a constant that can be absorbed by the $\mathcal{O}$-notation, but note that for larger matrices $d \times d$, you can't ignore the d^3 factor in the complexity; even $6^3 = 216$ is already significant enough to be noticeable, so we really shouldn't have ignored it!

- The transition is "time-blind" or "index-blind", i.e., independent of the current index i, and
- We only need to remember a finite window of the states.

Exercise 3.9. Find a 5×5 matrix that solves the same problem.

Exercise 3.10. Find a transition matrix to compute $f(n)$ in $\mathcal{O}(\log n)$ time, where f is defined as follows:

$$\begin{aligned} f(n) &= f(n-2) + 2f(n-3) + 4g(n-2) + 5g(n-3) \\ g(n) &= 5f(n-2) - 4f(n-3) - g(n-2) + 9g(n-3) \end{aligned}$$

with the first few values of f and g given.

Exercise 3.11. Find a transition matrix to compute $f(n)$ in $\mathcal{O}(\log n)$ time, where f is defined as follows:

$$\begin{aligned} f(n) &= f(n-1) + 2f(n-2) + 4g(n-1) + 5g(n-2) + 2019g(n) \\ g(n) &= 5f(n-1) - 4f(n-2) - g(n-1) + 9g(n-2) \end{aligned}$$

with the first few values of f and g given.

Bonus: Find a 4×4 transition matrix.

3.1.4 Small height-balanced trees

An AVL tree is a data structure that's designed to support set operations (insert, lookup and delete) in $\mathcal{O}(\log n)$. We don't need most details on how it works; for now, we just need to know the fact that it uses 1-height-balanced trees.

A **1-height-balanced tree** is a binary tree such that for every node, its left and right subtrees differ in height[9] by at most one.

Consider the following problem:

Problem 3.4. Given h, what is the smallest 1-height-balanced tree of height h? In other nodes, compute the minimum number of nodes needed to construct a 1-height-balanced tree of height h. (Let's write this as $f(h)$.)

Let's be constructive: how would you construct a *small* 1-height-balanced tree of height h? Well, in order for its height to be h, one of its subtrees must be of height $h-1$. It doesn't matter which one, so let's say the left subtree. Now, what can be the height of the right subtree? Clearly, it has to be at most $h-1$. But it also has to be at least $h-2$ to satisfy the height-balancing property. Since we want to minimize the number nodes, we want to minimize the height as well, so it must be $h-2$.

[9] The *height* of a tree is the largest depth of any of its nodes. The *depth* of a node is the number of edges between it and the root, hence, the depth of the root is 0. The height of the empty tree is -1, and the height of a tree consisting of a single node is 0.

Exercise 3.12. Show that f is increasing, i.e., $f(h) < f(h+1)$.

So now, our minimal subtree of height h has subtrees of heights $h-1$ and $h-2$ respectively. But these subtrees have to be 1-height-balanced as well, so they have at least $f(h-1)$ and $f(h-2)$ nodes, respectively. This means that we have the bound

$$f(h) \geq f(h-1) + f(h-2) + 1$$

where the $+1$ is for the root. Also, we can easily construct a tree with *exactly* $f(h-1) + f(h-2) + 1$ nodes, so we actually have equality:

$$f(h) = f(h-1) + f(h-2) + 1.$$

The base cases are

$$\begin{aligned} f(0) &= 1 \\ f(-1) &= 0. \end{aligned}$$

(Why?)

Using DP, this gives us an $\mathcal{O}(h)$ algorithm. But can we do better?

In fact, you should suspect that we can do better, because the recurrence above is suspiciously similar to the Fibonacci recurrence! They're not identical because of the $+1$, but you should now realize that f should be growing exponentially, which means that the height of a 1-height-balanced tree grows logarithmically in the number of nodes, enough to prove the efficiency of AVL trees!

But how do we compute f exactly? In fact, it's pretty simple once you realize that 1 is really the linear recurrence $1, 1, 1, \ldots$!

To explain in more detail, think of 1 as a *function* $\mathbf{1}$ that gives $\mathbf{1}(n) = 1$ for any n. Then we have the system of linear recurrences:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) + \mathbf{1}(n-1) \\ \mathbf{1}(n) &= \mathbf{1}(n-1). \end{aligned}$$

with base cases

$$\begin{aligned} f(-1) &= 0 \\ f(0) &= 1 \\ \mathbf{1}(0) &= 1. \end{aligned}$$

This gives us the following recurrence:

$$\begin{bmatrix} f(n) \\ f(n-1) \\ \mathbf{1}(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(n-1) \\ f(n-2) \\ \mathbf{1}(n-1) \end{bmatrix}$$

which gives

$$\begin{bmatrix} f(n) \\ f(n-1) \\ \mathbf{1}(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

which then gives us an $\mathcal{O}(\log h)$ algorithm to compute $f(h)$, and that is cool.

3.2 Weeding out stuff from the recurrence

The above demonstrates that even if you have something that isn't exactly a linear recurrence yet, it might still be a linear recurrence in disguise if you look carefully at the "extra stuff". In the above case, the $+1$ isn't actually that much of an issue. In fact, it can be any constant $+c$, and we can use the same technique.

Exercise 3.13. Find a transition matrix to compute $f(n)$ in $\mathcal{O}(\log n)$ time, where f is defined as follows:

$$\begin{aligned} f(n) &= f(n-1) + 2f(n-2) + 3g(n-1) + 4g(n-2) + 2 \\ g(n) &= 5f(n-1) - 4f(n-2) - g(n-1) + 9g(n-2) + 3 \end{aligned}$$

with the first few values of f and g given.

Bonus: Find a 5×5 transition matrix.

What else can we "weed out" from almost linear recurrences? Well, it turns out that there are a lot!

Example 3.14. Consider the following recurrence:

$$f(n) = f(n-1) + f(n-2) + 2^n.$$

In this case, it turns out that the sequence 2^n, or $1, 2, 4, 8, \ldots$, is also a linear recurrence if we define the function $\mathbf{2}(n) = 2^n$; we have the recurrence $\mathbf{2}(n) = 2 \cdot \mathbf{2}(n-1)$, so we have

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) + \mathbf{2}(n) \\ \mathbf{2}(n) &= 2 \cdot \mathbf{2}(n-1). \end{aligned}$$

So we can solve that as well!

Example 3.15. Now, what about

$$f(n) = f(n-1) + f(n-2) + n?$$

This is a little bit trickier, but you should be able to see that $n = (n-1)+1$, which means that if we define $\iota(n) = n$ and $\mathbf{1}(n) = 1$, then we have:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) + \iota(n) \\ \iota(n) &= \iota(n-1) + \mathbf{1}(n) \\ \mathbf{1}(n) &= \mathbf{1}(n-1). \end{aligned}$$

Example 3.16. Now, what about

$$f(n) = f(n-1) + f(n-2) + n^2?$$

It's even trickier than before, but one can easily note that $n^2 = (n-1)^2 + 2n - 1$, which means that we can define $\iota_2(n) = n^2$, and ι and $\mathbf{1}$ as before, and get:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) + \iota_2(n) \\ \iota_2(n) &= \iota_2(n-1) + 2\iota(n) - \mathbf{1}(n) \\ \iota(n) &= \iota(n-1) + \mathbf{1}(n) \\ \mathbf{1}(n) &= \mathbf{1}(n-1). \end{aligned}$$

These last few examples show that, in general, n^k can be "weeded out" by noticing that $n^k - (n-1)^k$ is a linear sum of lower powers of n, namely $1, n, n^2, \ldots, n^{k-1}$, so we can just remember all of $\mathbf{1}, \iota, \iota_2, \ldots, \iota_k$ in our state!

Example 3.17. As a final example, what about

$$f(n) = f(n-1) + f(n-2) + n^4 2^n?$$

This is a little bit tricky. We know that n^4 and 2^n are linear recurrences on their own, but is their product also a linear recurrence?

It turns out that yeah, it is![a] To see why, first we define $g(n) = n^4 2^n$. Then $g(n-1)$ is $(n-1)^4 2^{n-1}$, and $2g(n-1)$ is $(n-1)^4 2^n$. Now, what happens when we compute $g(n) - 2g(n-1)$? Well,

$$\begin{aligned} g(n) - 2g(n-1) &= (n^4 - (n-1)^4)2^n \\ &= (4n^3 - 6n^2 + 4n - 1)2^n \\ &= 4n^3 2^n - 6n^2 2^n + 4n2^n - 2^n. \end{aligned}$$

The big picture here is that we have expressed $g(n)$ as $2g(n-1)$ plus a bunch of $n^k 2^n$ terms for lower exponents $k < 4$. But by a similar derivation, 2^n, $n2^n$, $n^2 2^n$ and $n^3 2^n$ are also expressible in terms of lower powers! Hence, if we remember $2^n, n2^n, n^2 2^n, n^3 2^n$ and $n^4 2^n$ in our state, then we can solve this one as well!

[a] In fact, the natural generalization is true: if two functions $F(n)$ and $G(n)$ are defined by a finite linear recurrence, then $F(n)G(n)$ is also defined by a finite linear recurrence. Later on, we will develop techniques to prove statements like this.

The previous examples show that there are quite a lot of different kinds of "extra stuff" that we can remove from the recurrence. In its most general version, we can remove anything of the following form:

$$p_1(n)\alpha_1^n + p_2(n)\alpha_2^n + \ldots + p_k(n)\alpha_k^n$$

for some polynomials $p_1, \ldots, p_k$ and constants $\alpha_1, \ldots \alpha_k$. The proof is constructive and the construction is basically just the combination of the techniques in the previous examples.

Later on, once we learn the things called *generating functions*, we will also be able to prove that these are the *only* sorts of things that we can weed out. We can also prove that any function that can be computed with a matrix power, as before, is expressible in this form!

3.3 Recurrence shaking

In this section, we will describe a technique that can possibly turn an infinite recurrence to a finite one.

3.3.1 Weeding out, part 2

Let's revisit the "weeding out" techniques that we used earlier. In fact, there's an alternative way to weed out extra stuff.

Example 3.18. Let's go back to our earlier example:

$$f(n) = f(n-1) + f(n-2) + 1.$$

Instead of defining $\mathbf{1}$, let's see if we can remove 1 in another way. Replacing n by $n-1$ above, we get

$$f(n-1) = f(n-2) + f(n-3) + 1.$$

Then, if we subtract them, we get:

$$\begin{aligned} f(n) - f(n-1) &= f(n-1) - f(n-3) \\ f(n) &= 2f(n-1) - f(n-3). \end{aligned}$$

This gives us an order-3 linear recurrence for f; in effect, instead of introducing a separate function $\mathbf{1}$, we've weeded out $+1$ by increasing the order of the recurrence.

Note, though, that the transition matrix stays the same in size.

This is the essence of what I call *recurrence shaking*, a term I just invented: we use the recurrence itself, and "shake it a bit" (change indices) until we remove the extra stuff we don't want.

Exercise 3.19. Use recurrence shaking to find a pure linear recurrence for f defined as

$$f(n) = 2f(n-1) + 3f(n-2) + 2^n.$$

Exercise 3.20. Use recurrence shaking to find a pure linear recurrence for f defined as

$$f(n) = 2f(n-1) + 3f(n-2) + n2^n.$$

3.3.2 String counting (again!)

Recurrence shaking can do more than just weed out extra stuff. As explained in the intro to this section, it can also possibly turn an infinite recurrence to a finite one.

Let us go back to string problems.

Problem 3.5. Our alphabet is again `NOI`. We want to find the number of length-n strings that do not contain the substring `NO`. (We now forget about letter weights and return to length.) Let $f(n)$ be the answer to this question.

We already know how to answer this one, but let's ignore that for now.

Now, suppose that, as we construct the recurrence, you insist that we *must not* remember any of the letter(s) we have chosen so far. In other words, we want to find a direct recurrence of $f(n)$ in terms of its smaller values. (This is ill-advised, but please go with it for now :D)[10]

[10] Actually, this scenario isn't that far-fetched. It can happen in times when you're solving a DP recurrence,

Now, let's do the constructive approach again. Except this time, since we don't remember the previous letters we have chosen, we must choose enough of the string and only stop at the points in time when we know that the later choices are essentially the same as the current problem. Let's do it.

- If the first letter is `O`, then the rest of the string can be any string of length $n-1$ not containing the substring `NO`, so there are $f(n-1)$ such strings.
- If the first letter is `I`, then it's basically the same: we're safe, so there are $f(n-1)$ such strings.
- However, if the first letter is an `N`, then we can't just pass the work to $f(n-1)$ again, since we can't append the strings that begin with an `O`! So we must continue to choose the following letters.
  - If the next letter is an `I`, then we're safe. The remaining $n-2$ letters can be chosen in $f(n-2)$ ways.
  - If the next letter is an `O`, then we've failed, since we've just formed `NO`. So we drop this case.
  - If the next letter is an `N`, then we're back to the same issue; the next letter cannot be an `O`! So let's decide the next letter.
    - If the next letter is an `I`, then we're safe. The remaining $n-3$ letters can be chosen in $f(n-3)$ ways.
    - If the next letter is an `O`, then we've failed, since we've just formed `NO`. So we drop this case.
    - If the next letter is an `N`, then we're back to the same issue; the next letter cannot be an `O`! So let's decide the next letter.
      - ...

It is clear that this argument will not end, since the length of the initial sequence `NNNN...` can be arbitrarily long.

But actually, it is not arbitrarily long! The sequence is at most n in length, which means that this chain of reasoning will stop at some point. Therefore, this actually gives us a recurrence, although not a fixed-sized one:

$$f(n) = 2f(n-1) + f(n-2) + f(n-3) + f(n-4) + \ldots + f(1) + f(0).$$

Now, by itself, this yields an $\mathcal{O}(n^2)$ DP algorithm, because the transition is $\mathcal{O}(n)$. However, what if we use recurrence shaking here?

Replacing n by $n-1$ above, we get

$$f(n-1) = 2f(n-2) + f(n-3) + f(n-4) + f(n-5) + \ldots + f(1) + f(0).$$

If we subtract this from the previous one, we get:

$$f(n) - f(n-1) = 2f(n-1) - f(n-2).$$

Note how nicely everything cancels out! Thus, we get the finite linear recurrence

$$f(n) = 3f(n-1) - f(n-2)$$

via recurrence shaking!

but you're using an incomplete state! (Maybe you don't know that it's incomplete yet.) So you can use the techniques here to maybe still make the solution work.

Exercise 3.21. Find a combinatorial interpretation of the recurrence

$$f(n) = 3f(n-1) - f(n-2)$$

in terms of the string problem.

Note that we could have solved the problem directly with matrices, but in some cases, the first recurrence you might come up with may be of the infinite form like the above one. In such cases, having a technique like "recurrence shaking" can help you get down to a finite recurrence almost mechanically.

Now, we consider other kinds of "infinite" recurrence that we can shake to become finite.

Example 3.22. Consider the recurrence

$$f(n) = 3f(n-1) + f(n-2) + 2f(n-3) + f(n-4) + 2f(n-5) + \dots$$

where the tail of coefficients repeat as $1, 2, 1, 2, \dots$

Using recurrence shaking, we can subtract $f(n-2)$ from $f(n)$:

$$\begin{aligned} f(n) &= 3f(n-1) + f(n-2) + 2f(n-3) + f(n-4) + 2f(n-5) + \dots \\ f(n-2) &= 3f(n-3) + f(n-4) + 2f(n-5) + f(n-6) + 2f(n-7) + \dots \\ f(n) - f(n-2) &= 3f(n-1) + f(n-2) - f(n-3) \\ f(n) &= 3f(n-1) + 2f(n-2) - f(n-3), \end{aligned}$$

giving us a finite recurrence for $f(n)$.

Example 3.23. Recurrence shaking can also sometimes be used even if the tail of coefficients don't repeat. Consider the following:

$$f(n) = f(n-1) + 4f(n-2) + 9f(n-3) + 16f(n-4) + \dots + n^2 f(0).$$

Let's just do it. Subtract $f(n-1)$ from $f(n)$:

$$\begin{aligned} f(n) &= f(n-1) + 4f(n-2) + 9f(n-3) + 16f(n-4) \dots \\ f(n-1) &= f(n-2) + 4f(n-3) + 9f(n-4) + 16f(n-5) \dots \\ f(n) - f(n-1) &= f(n-1) + 3f(n-2) + 5f(n-3) + 7f(n-4) \dots \end{aligned}$$

We see that the sequence of coefficients are now linear-growing instead of quadratic. Let's shake again: subtract $f(n-1) - f(n-2)$ from $f(n) - f(n-1)$ to get:

$$\begin{aligned} f(n) - f(n-1) &= f(n-1) + 3f(n-2) + 5f(n-3) + 7f(n-4) \dots \\ f(n-1) - f(n-2) &= f(n-2) + 3f(n-3) + 5f(n-4) + 7f(n-5) \dots \\ f(n) - 2f(n-1) + f(n-2) &= f(n-1) + 2f(n-2) + 2f(n-3) + 2f(n-4) \dots \end{aligned}$$

Almost there! One more shake:

$$\begin{aligned}
f(n) - 2f(n-1) + f(n-2) &= f(n-1) + 2f(n-2) + 2f(n-3) \dots \\
f(n-1) - 2f(n-2) + f(n-3) &= f(n-2) + 2f(n-3) + 2f(n-4) \dots \\
f(n) - 3f(n-1) + 3f(n-2) - f(n-3) &= f(n-1) + f(n-2) \\
f(n) &= 4f(n-1) - 2f(n-2) + f(n-3)
\end{aligned}$$

Now, we have a finite recurrence. Whew!

Exercise 3.24. Use recurrence shaking to find a recurrence for $f(n)$ defined by

$$f(n) = f(n-1) + 2f(n-2) + 4f(n-3) + 8f(n-4) + \ldots + 2^n f(0).$$

Example 3.25. The geometric series

$$G_r(n) = 1 + r + r^2 + \ldots + r^{n-1}$$

can be seen in light of recurrences as well. Let's say $\mathbf{r}(n) = r^n$, so that we have the system of recurrences:

$$\begin{aligned}
\mathbf{r}(n) &= r \cdot \mathbf{r}(n-1) \\
G_r(n) &= \mathbf{r}(n-1) + \mathbf{r}(n-2) + \ldots + \mathbf{r}(0).
\end{aligned}$$

By shaking $G_r(n)$, we get

$$\begin{aligned}
G_r(n) &= \mathbf{r}(n-1) + \mathbf{r}(n-2) + \ldots + \mathbf{r}(0) \\
G_r(n-1) &= \mathbf{r}(n-2) + \mathbf{r}(n-3) + \ldots + \mathbf{r}(0) \\
G_r(n) - G_r(n-1) &= \mathbf{r}(n-1) \\
G_r(n) &= G_r(n-1) + r^{n-1}.
\end{aligned}$$

But we already know how to remove the extra stuff. In fact, we can just shake it away as well!

$$\begin{aligned}
G_r(n) &= G_r(n-1) + r^{n-1} \\
G_r(n-1) &= G_r(n-2) + r^{n-2} \\
rG_r(n-1) &= rG_r(n-2) + r^{n-1} \\
G_r(n) - rG_r(n-1) &= G_r(n-1) - rG_r(n-2) \\
G_r(n) &= (r+1)G_r(n-1) - rG_r(n-2).
\end{aligned}$$

This gives us a neat finite recurrence for the geometric series $G_r(n)$. In fact, from this and the base cases $G_r(0) = 0$ and $G_r(1) = 1$, you can now inductively prove that $G_r(n) = \frac{r^n-1}{r-1}$.

Once we learn generating functions, we will see linear recurrences like these in a different light, and also learn techniques on how to *derive* the formula $\frac{r^n-1}{r-1}$ from the recurrence, not just prove.

Shaking isn't limited to just reducing 1D recurrences. For example, consider the recurrence that we had for the partition number problem:

$$\begin{aligned} f(\ell, r) &= \sum_{c=\ell}^{r} f(c, r-c) \\ &= f(\ell, r-\ell) + f(\ell+1, r-\ell-1) + f(\ell+2, r-\ell-2) + \dots f(r, 0). \end{aligned}$$

We can similarly shake this; consider $f(\ell+1, r)$. By the above, we get

$$f(\ell+1, r) = f(\ell+1, r-\ell-1) + f(\ell+2, r-\ell-2) + \dots f(r, 0).$$

Now, we can subtract this from $f(\ell, r)$ to get:

$$\begin{aligned} f(\ell, r) - f(\ell+1, r) = f(\ell, r-\ell) \\ f(\ell, r) = f(\ell+1, r) + f(\ell, r-\ell). \end{aligned}$$

Here, we get an $\mathcal{O}(1)$ transition instead of $\mathcal{O}(n)$, hence, our DP algorithm improves to $\mathcal{O}(n^2) \cdot \mathcal{O}(1) = \mathcal{O}(n^2)$. That's awesome!

Exercise 3.26. Find a combinatorial interpretation for

$$f(\ell, r) = f(\ell+1, r) + f(\ell, r-\ell).$$

This example shows that even if you end up with a somewhat big recurrence/transition, shaking can sometimes help you speed it up.

4 Other DP patterns

4.1 Generalization of matrix techniques

The idea of optimizing DPs to logarithmic time is not exclusive to linear recurrences; at its heart, it is really just divide-and-conquer. (Fast exponentiation is just divide-and-conquer.)

4.1.1 String problem (not counting!)

To illustrate, let's consider the following problem.

Problem 4.1. Suppose you're constructing a binary string of length n, on the alphabet `OI`. Suppose that the strings `OO`, `OI`, `IO`, `II` all have *costs*, say, $c_{\texttt{OO}}$, $c_{\texttt{OI}}$, $c_{\texttt{IO}}$ and $c_{\texttt{II}}$, respectively. The cost of a string is the total cost of all its 2-letter substrings.

What is the minimum cost of any string of length n that begins and ends in an `O`?

Note that this problem is not that difficult and has an $\mathcal{O}(1)$ solution. However, let's attempt to solve it with DP and divide-and-conquer, since that technique is more flexible and generalizes to more contexts.

Exercise 4.1. Find an $\mathcal{O}(1)$ solution to this problem.

Let's think about how to construct such a string. Note that the first and last letters are fixed, so we only need to choose the middle $n-2$ letters. If we construct such a string from left to right, we find that there are only two things that affect our decisions:

- The previous letter, say p.
- The number of letters that remain to be chosen, say r.

In particular, we don't need to remember all our past choices! Thus, our state in this case could be (p, r).

Let's now define $f(p, r)$ as the minimum cost we incur starting from the state (p, r). Note that, in this case, we only count the cost contributions of all our *future* choices; we don't have to remember the total cost contributions of all our past choices since they don't really affect our decisions, and they will just blow up the size of our state.

By considering all cases, we end up with the following system of (non-linear) recurrences

$$\begin{aligned} f(\texttt{O}, r) &= \min(f(\texttt{O}, r-1) + c_{\texttt{OO}}, f(\texttt{I}, r-1) + c_{\texttt{OI}}) \\ f(\texttt{I}, r) &= \min(f(\texttt{O}, r-1) + c_{\texttt{IO}}, f(\texttt{I}, r-1) + c_{\texttt{II}}). \end{aligned}$$

The base cases are

$$\begin{aligned} f_{\texttt{O}}(0) &= c_{\texttt{OO}} \\ f_{\texttt{I}}(0) &= c_{\texttt{IO}}. \end{aligned}$$

Exercise 4.2. Show that we can alternatively use the base cases:

$$\begin{aligned} f_{\mathtt{0}}(-1) &= 0 \\ f_{\mathtt{1}}(-1) &= \infty. \end{aligned}$$

This gives us $\mathcal{O}(2n)$ states and a transition time of $\mathcal{O}(1)$, so the DP solution is $\mathcal{O}(n)$.

Now, this isn't logarithmic, but at this point we could hope that we can optimize it to $\mathcal{O}(\log n)$ as before, perhaps using something analogous to fast exponentiation, but we can't do that here because the recurrences are not linear!

But let's continue with our hope, and let's step back and determine how the previous optimization was even possible in the first place. Let's recall these conditions (that I mentioned earlier) on when we can use the fast exponentiation approach:

- The decisions we make are only dependent on a "narrow window" around where we're making it.
- The decisions are "time-blind"; the same decision process is made regardless of n.

These conditions are what allow us to use divide-and-conquer. To see this, consider first that our system has the states $S_0, S_1, \ldots$ and the transition functions are $\varphi_1, \varphi_2, \ldots$, where φ_n is the transition from the state S_{n-1} to the next state, S_n. Then:

- The "narrow window" property tells us that the states, the S_i's, are bounded in size.
- The "time-blindness" property tells us that all transition functions are the same regardless of which state we're transitioning. Hence, $\varphi_1, \varphi_2, \ldots$ are all the same, and we can denote then by the same symbol: φ.

Therefore, S_n is obtained by iteratively applying some fixed transition φ a total of n times, starting from the "base state", S_0, that is, $S_n = \varphi^n(S_0)$, where φ^n denotes applying φ n times, or in other words, φ^n is the function φ composed with itself n times. But composing functions is *associative*, which means the "divide-and-conquer" approach of fast exponentiation can be used! Specifically, the following rules of fast exponentiation also apply:

$$\begin{aligned} \varphi^0 &= \text{identity} \\ \varphi^{n+1} &= \varphi \circ \varphi^n \\ \varphi^{2n} &= (\varphi^n)^2. \end{aligned}$$

So linearity doesn't play a role in what makes divide-and-conquer possible. But if that's the case, then why do we require it—that is, why is it so important that the transition be expressible as a matrix?

The real answer is that, in general, the "transition" φ^n could be **large**. This means that, even though we could still use "fast" exponentiation, the total amount of work we perform may still be proportional to n. But if the transition function φ is a matrix, say of size $s \times s$, then φ^n stays of the same size regardless of n, which means that the divide-and-conquer approach actually reduces the amount of work! To be precise, it reduces the amount of work from $\mathcal{O}(s^2 n)$ (since it takes $\mathcal{O}(s^2)$ amount of work to "advance" a state) to $\mathcal{O}(s^3 \log n)$ (since it takes $\mathcal{O}(s^3)$ amount of work to compose—that is multiply—a linear transition matrix to another).

Thus, linearity simply keeps the size of the transition bounded!

Now, let's go back to the string problem. What is our transition function? It is the following:

$$\varphi\left(\begin{bmatrix} f(\mathtt{0}, r) \\ f(\mathtt{I}, r) \end{bmatrix}\right) = \begin{bmatrix} \min(f(\mathtt{0}, r-1) + c_{\mathtt{00}}, f(\mathtt{I}, r-1) + c_{\mathtt{0I}}) \\ \min(f(\mathtt{0}, r-1) + c_{\mathtt{I0}}, f(\mathtt{I}, r-1) + c_{\mathtt{II}}) \end{bmatrix}.$$

Now, let's see what happens if we compose two transition functions of this form, say φ and ψ:

$$\begin{aligned} \varphi\left(\begin{bmatrix} f(\mathtt{0}, r) \\ f(\mathtt{I}, r) \end{bmatrix}\right) &= \begin{bmatrix} \min(f(\mathtt{0}, r) + c_{\mathtt{00}}, f(\mathtt{I}, r) + c_{\mathtt{0I}}) \\ \min(f(\mathtt{0}, r) + c_{\mathtt{I0}}, f(\mathtt{I}, r) + c_{\mathtt{II}}) \end{bmatrix} \\ \psi\left(\begin{bmatrix} f(\mathtt{0}, r) \\ f(\mathtt{I}, r) \end{bmatrix}\right) &= \begin{bmatrix} \min(f(\mathtt{0}, r) + d_{\mathtt{00}}, f(\mathtt{I}, r) + d_{\mathtt{0I}}) \\ \min(f(\mathtt{0}, r) + d_{\mathtt{I0}}, f(\mathtt{I}, r) + d_{\mathtt{II}}) \end{bmatrix}. \end{aligned}$$

What does applying φ and then ψ do to a given state? Amazingly, it turns out that it looks like the following:

$$(\varphi \circ \psi)\left(\begin{bmatrix} f(\mathtt{0}, r) \\ f(\mathtt{I}, r) \end{bmatrix}\right) = \begin{bmatrix} \min(f(\mathtt{0}, r) + e_{\mathtt{00}}, f(\mathtt{I}, r) + e_{\mathtt{0I}}) \\ \min(f(\mathtt{0}, r) + e_{\mathtt{I0}}, f(\mathtt{I}, r) + e_{\mathtt{II}}) \end{bmatrix}. \tag{1}$$

where the e values can be computed as follows:

$$\begin{aligned} e_{\mathtt{00}} &= \min(c_{\mathtt{00}} + d_{\mathtt{00}}, c_{\mathtt{0I}} + d_{\mathtt{I0}}) \\ e_{\mathtt{0I}} &= \min(c_{\mathtt{00}} + d_{\mathtt{0I}}, c_{\mathtt{0I}} + d_{\mathtt{II}}) \\ e_{\mathtt{I0}} &= \min(c_{\mathtt{I0}} + d_{\mathtt{00}}, c_{\mathtt{II}} + d_{\mathtt{I0}}) \\ e_{\mathtt{II}} &= \min(c_{\mathtt{I0}} + d_{\mathtt{0I}}, c_{\mathtt{II}} + d_{\mathtt{II}}). \end{aligned}$$

This means that the transition function is bounded in size, and we can perform divide-and-conquer to get the faster running time $\mathcal{O}(\log n)$! (and that's really awesome!!)

4.1.2 Coming up with this solution

Now, I know you might say, "that's cool and all, but how am I supposed to come up with such a solution during contest!?" And yeah, this isn't how you usually come up with such a solution. For this, it's helpful to just consider the problem and then check if we can just use divide-and-conquer *directly*.

Let's try that in this problem. Here, the "divide" part amounts to splitting the array in half, say, by picking the middle letter and choosing it irst. So, let's say we choose the middle letter first. (Let's assume that n is odd for simplicity, so that there is a middle letter.) This amounts to choosing the letter $w_{\lceil n/2 \rceil - 1}$ first, assuming w is our word. What should it be? Well, there are two possibilities.

- If $w_{\lceil n/2 \rceil - 1}$ is an $\mathtt{0}$, then we minimize the cost by minimizing both sides of the string: $w_{0 \ldots \lceil n/2 \rceil - 1}$ and $w_{\lceil n/2 \rceil - 1 \ldots n-1}$. By definition, this is $f(\mathtt{00}, \lceil n/2 \rceil)$ on both sides, so the minimum cost is
$$f(\mathtt{00}, \lceil n/2 \rceil) + f(\mathtt{00}, \lceil n/2 \rceil).$$
- If $w_{\lceil n/2 \rceil - 1}$ is an $\mathtt{I}$, then we minimize the cost by minimizing both sides of the string, $w_{0 \ldots \lceil n/2 \rceil - 1}$ and $w_{\lceil n/2 \rceil - 1 \ldots n-1}$. By definition, this is $f(\mathtt{0I}, \lceil n/2 \rceil)$ on the left and $f(\mathtt{I0}, \lceil n/2 \rceil)$ on the right, so the minimum cost is
$$f(\mathtt{0I}, \lceil n/2 \rceil) + f(\mathtt{I0}, \lceil n/2 \rceil).$$

Important here is the fact that the choices we make on the left and right sides of the middle letter do not interfere with each other ("narrow window"), hence we can solve those subproblems separately. What's more, since both subproblems are basically the same ("time-blindness"), it amounts to only solving the subproblem once, i.e., we don't keep track of the indices of where we're constructing the string, only the length. This means that there's really only one subproblem, which is half the size, so overall we only take $\mathcal{O}(\log n)$ amount of work!

Exercise 4.3. Solve the same problem but on an alphabet of 4 letters. Specifically, suppose that each of the $4^2 = 16$ two-letter strings has a cost. What is the minimum cost of any length-n string that starts and end in some given letter?

Notice that generalizing the $\mathcal{O}(1)$ solution is much harder than generalizing the divide-and-conquer approach.

4.1.3 A tropical look

In this particular case, we can make the connection with matrix exponentiation more explicit by thinking of the operations "min" and "+" as the sort of "addition" and "multiplication" in a somewhat different number system. In fact, let's write them as $\oplus$ and $\otimes$, i.e.,

$$\begin{aligned} x \oplus y &= \min(x, y) \\ x \otimes y &= x + y. \end{aligned}$$

Then we can express our recurrences above as follows:

$$\begin{aligned} f(\mathtt{O}, r) &= c_{\mathtt{OO}} \otimes f(\mathtt{O}, r-1) \oplus c_{\mathtt{OI}} \otimes f(\mathtt{I}, r-1) \\ f(\mathtt{I}, r) &= c_{\mathtt{IO}} \otimes f(\mathtt{O}, r-1) \oplus c_{\mathtt{II}} \otimes f(\mathtt{I}, r-1). \end{aligned}$$

which we can also express compactly in "matrix" form:

$$\begin{bmatrix} f(\mathtt{O}, r) \\ f(\mathtt{I}, r) \end{bmatrix} = \begin{bmatrix} c_{\mathtt{OO}} & c_{\mathtt{OI}} \\ c_{\mathtt{IO}} & c_{\mathtt{II}} \end{bmatrix} \otimes \begin{bmatrix} f(\mathtt{O}, r-1) \\ f(\mathtt{I}, r-1) \end{bmatrix},$$

where "matrix multiplication" is performed with $\oplus$ and $\otimes$. Thus, we can express the divide-and-conquer using fast "matrix exponentiation" as follows:

$$\begin{bmatrix} f(\mathtt{O}, n) \\ f(\mathtt{I}, n) \end{bmatrix} = \begin{bmatrix} c_{\mathtt{OO}} & c_{\mathtt{OI}} \\ c_{\mathtt{IO}} & c_{\mathtt{II}} \end{bmatrix}^{\otimes n} \otimes \begin{bmatrix} f(\mathtt{O}, 0) \\ f(\mathtt{I}, 0) \end{bmatrix}.$$

I write the exponent as "$\otimes n$" to clarify that it is exponentiation with respect to $\otimes$.

This works because $\oplus$ and $\otimes$ behave very similarly to + and × and have a lot of their nice properties, e.g., commutativity, associativity, distributivity, etc.[11] More importantly, like normal exponentiation, $\otimes$-based exponentiation is also amenable to the same "fast exponentiation" techniques.

Writing things this way, the similarity with matrix multiplication is even more apparent! (However, again note that we aren't always able to do this; it just so happens that we can for this problem.)

Note: The operations $\oplus$ and $\otimes$ are called *tropical addition* and *tropical multiplication*, respectively, and together with the numbers, they form a mathematical structure called the *tropical semiring*.[12] (You don't have to worry about what those words mean!)[13]

[11] One difference is that $\oplus$ doesn't have inverses. However, $\oplus$ has an identity—what is it?

[12] `https://en.wikipedia.org/wiki/Tropical_semiring`

[13] Though you might ask, why "tropical"? You may ask in Discord; I'm sure someone will answer!

4.2 "Tree convolution" DP

The "tree convolution DP" is just my term for a particular kind of DP solution that occurs on rooted trees. It is a DP solution which naively looks like it runs in $\mathcal{O}(n^3)$ but in fact runs in $\mathcal{O}(n^2)$.

To illustrate it, let's consider the following problem.

Problem 4.2. Suppose we are given a rooted binary tree with $n \leq 5000$ nodes. Each node has a *score*; the score of node i is s_i. Your task is to find a subset of the nodes such that:

- it has at most k nodes (for some $k \leq n$),
- for every two nodes x and y in the set, x is neither an ancestor nor a descendant of y, and
- the total score across all nodes is maximized.

Note that everything here also works for all kinds of rooted trees, not just binary trees. We just choose to restrict ourselves to binary trees to make the explanations simpler.

4.2.1 The DP solution

How would we solve such a problem? You might think of greedy solutions like taking only the root, or maximizing the number of nodes we select, but these "solutions" are very easy to break. For example, for the greedy solution of maximizing the number of nodes, one can imagine the root to have a very large score that it completely dominates the sum of the rest of the nodes. So we can't just maximize the number of nodes.

Instead, let's be constructive again and figure out a way to construct such a subset—any subset. What would it look like? Where do we even begin?

The rooted tree structure helps us out here;[14] let's begin by considering the root.

- If we take the root, then we cannot take any other node since they are all descendants of the root. Thus, our score for this case is s_{root}.
- If we don't take the root, then we're free to ignore it and consider the two subtrees. We are supposed to take a few nodes (possibly zero) from one subtree and the rest from the other (possibly zero), but the total number of nodes must be at most k.

  To start with, we don't know how many we will need take from either subtree, so let's decide that first. Suppose we take t nodes from the left subtree. Then we can only take at most $k-t$ nodes from the right subtree. Note that t can be anything from 0 to k, though for a slight optimization, we can restrict t to only those where $t \leq \mathit{size}(\text{left subtree})$ and $k-t \leq \mathit{size}(\text{right subtree})$.

  We have now reduced the problem to two subproblems: the same problem on both subtrees. Thus, we can continue the same construction for each node, recursively...

Here, note that our decision-making only depends on two things: which subtree we are in, and the maximum number of nodes we can take. In particular, the choices we made in the other subtrees don't really matter. Thus, our DP state can be (i, m), where i is the current node, and m is the maximum number of nodes we can take. Note that we can assume that $m \leq \mathit{size}(i)$.

[14] and if it is not rooted, we could root it first!

Thus, we have the recurrence:

$$f(i,m) = \max\left(s_i, \max_{\substack{t \ge \max(0, m - \mathit{size}(r)) \\ t \le \min(m, \mathit{size}(\ell))}} f(\ell, t) + f(r, m-t) \right)$$

where ℓ and r are the left and right subtrees, respectively.

Exercise 4.4. Where do the bounds $\max(0, m - \mathit{size}(r))$ and $\min(m, \mathit{size}(\ell))$ come from?

The answer we are looking for is then just $f(\mathit{root}, k)$. For the base case, we have $f(i, 0) = f(\mathit{null}, m) = 0$ (verify), where "*null*" corresponds to an empty subtree.

Exercise 4.5. Why can we use $f(\ell, t)$ for the left subtree even though $f(\ell, t)$ means "at most t nodes", and in the explanation above, we said that we will take *exactly* t nodes from the left subtree?

Implementing this algorithm is straightforward. Here's an example implementation, without the memoization.

```
vector<int> s;     // score
vector<int> left;  // left child
vector<int> right; // right child
vector<int> size;  // subtree size

int f(int i, int m) {
    if (i == 0 || m == 0) return 0;  // base case

    int ans = s[i], l = left[i], r = right[i];
    for (int t = max(0, m - size[r]); t <= min(m, size[l]); t++) {
        ans = max(ans, f(l, t) + f(r, m - t));
    }
    return ans;
}
```

Note that:

- We index the nodes 1 to n, and use node 0 as the *null* node, and point nonexistent children to 0.
- The solution can now be computed by calling `f(root, k)`.
- We assume that the four vectors `s`, `left`, `right`, and `size` have been prepared beforehand.

4.2.2 The complexity

What is the time complexity of this DP approach? Let's try to compute it naively; we have $\mathcal{O}(nk)$ states "(i, m)", and in a transition, we loop across $\mathcal{O}(m)$ values, so it runs in $\mathcal{O}(m) = \mathcal{O}(k)$ time, so the running time seems to be $\mathcal{O}(nk) \cdot \mathcal{O}(k) = \mathcal{O}(nk^2)$. When k is close to n, this is basically $\mathcal{O}(n^3)$.

This is a valid bound. However, with $n = 5000$, it looks like it won't pass the time limit in the worst test cases.

But what are these "worst test cases"? Can you come up with one that triggers the $\mathcal{O}(n^3)$ "worst case"? You could perhaps consider typical *extreme* trees like line trees or perfectly balanced trees. But if we analyze the running time of our algorithm for each of these, we get:

- For line trees, it seems that our algorithm runs in $\mathcal{O}(n^2)$ time because one of the subtrees is always empty, hence the transition is really $\mathcal{O}(1)$.
- For perfectly balanced trees, we can argue recursively. Suppose $T(n)$ is the complexity of the algorithm for a balanced tree with n nodes. Then the amount of work done on the root is $\mathcal{O}(n^2)$, and the amount of work done on each subtree is $T(n/2)$, so we have the recurrence
$$T(n) = 2T(n/2) + \mathcal{O}(n^2)$$
which we can solve to be $\mathcal{O}(n^2)$ again.[15]

So if there are cases that trigger $\mathcal{O}(n^3)$, then they must be something else in between these two extremes. You could try experimenting with a few other types of trees, but no matter what you do, you might find it difficult to trigger something even slightly worse than $\Theta(n^2)$. Could it be that the algorithm is really $\mathcal{O}(n^2)$?

It turns out that it is! And the proof is rather clever.

The first step is to precisely pin down the amount of work done in some node i in terms of $\mathit{size}(\ell)$ and $\mathit{size}(r)$, where ℓ and r are the children of i.[16] Remember that for a node i, there are exactly $\mathit{size}(i) + 1$ states in it (since $0 \le m \le \mathit{size}(i)$). Now, consider some $f(i, m)$. To compute it, we need a bunch of terms like $f(\ell, t) + f(r, m - t)$ for several t. Rephrasing it slightly, for every $f(i, m)$, we need terms of the form $f(\ell, t) + f(r, t')$ for pairs of numbers (t, t') such that $t + t' = m$. The amount of work we perform for node i is simply proportional to the number of terms (t, t') we use, across all m (considering the number of times each pair is used).

But notice that:

- $0 \le t \le \mathit{size}(\ell)$,
- $0 \le t' \le \mathit{size}(r)$, and
- each pair (t, t') is used exactly once, namely for $m = t + t'$.

Therefore, the amount of work done is simply $\mathcal{O}(\mathit{size}(\ell) \cdot \mathit{size}(r))$!

This gives us the following recurrence, for the time complexity of the algorithm for a node i:
$$T(i) = T(\ell) + T(r) + \mathcal{O}(\mathit{size}(\ell) \cdot \mathit{size}(r)).$$

Exercise 4.6. What happens when either $\mathit{size}(\ell)$ or $\mathit{size}(r)$ is zero? Aren't we underestimating the time complexity?

Now, how do we solve the recurrence for $T(i)$? Note that we believe the answer to be $\mathcal{O}(\mathit{size}(i)^2)$, i.e., that $T(i) \le C \cdot \mathit{size}(i)^2$ for some constant C (for sufficiently large $\mathit{size}(i)$). What follows is a (not-so-formal) proof of this.[17]

[15] For example, recursively expanding $T(n/2)$, we get $T(n) = \mathcal{O}(n^2 + 2(n/2)^2 + 4(n/4)^2 + \ldots) = \mathcal{O}(n^2 + n^2/2 + n^2/4 + \ldots) = \mathcal{O}(2n^2) = \mathcal{O}(n^2)$.

[16] Note that $\mathit{size}(i) = \mathit{size}(\ell) + \mathit{size}(r) + 1$, so we can use $\mathit{size}(i)$ as well.

[17] The essential details are there; I'm sure you can fill in all the missing details, if you're so inclined. If you want the formal version, please consult Appendix A. :)

Proof. Instead of proving it directly, let's just solve the following simplified recurrence:

$$T(i) = T(\ell) + T(r) + (\mathit{size}(\ell) + 1) \cdot (\mathit{size}(r) + 1).$$

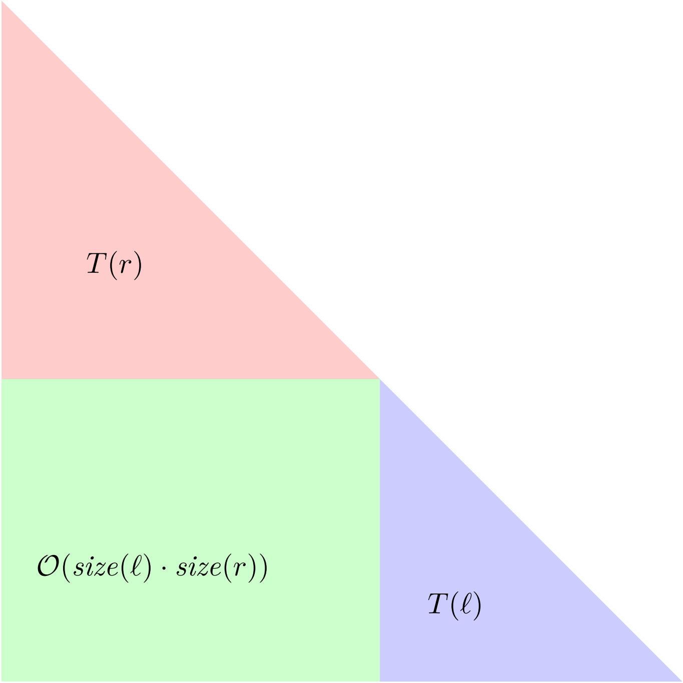
with the base case $T(\mathit{null}) = 1/2$.

In fact, $T(i) = \frac{1}{2}(\mathit{size}(i) + 1)^2$ solves this recurrence, and it is easily proven by induction! Thus, it is left as an exercise to the reader. □

Exercise 4.7. Carry out the inductive proof above.

Hint: $\mathit{size}(i) = \mathit{size}(\ell) + \mathit{size}(r) + 1$.

Note that the magic value $1/2$ is chosen specially to make the recurrence easy to solve. In fact, we can visualize the importance of $1/2$ geometrically, by viewing $\frac{1}{2}\mathit{size}^2$ as the area of a certain triangle:



Can you see how this image illustrates why $T(i) \approx \frac{1}{2}\mathit{size}(i)^2$?

Exercise 4.8. Modify the algorithm above to still work even for general rooted trees, not just binary rooted trees. Show that it still runs in $\mathcal{O}(n^2)$.

Hint: The state will now contain three numbers, (i, m, c), where

$$0 \leq c \leq \text{number of children of } i.$$

What could c possibly be?

Exercise 4.9. Note that to achieve the $\mathcal{O}(n^2)$ complexity, it is important that we restricted

> our loop on t to only those where $t \leq \mathit{size}(\ell)$ and $m - t \leq \mathit{size}(r)$.
>
> Without restricting t that way, what happens to the worst-case running time of the solution?

4.3 Sieves and DP

There's a kind of DP that has a distinct "sieve" feel. We'll motivate this with the following problem:

> **Problem 4.3.** Given a sequence S of positive integers and an integer m, how many good subsequences does it have? A sequence is *good* if it is nonempty, and the sum of the digits of its gcd is divisible by m.
>
> Two subsequences are considered distinct if there's an index i such that index i is in one subsequence but not in the other. In particular, the same sequence of values may be counted multiple times.
>
> Find the answer modulo $10^9 + 7$.
>
> **Constraints:** $|S| \leq 10^6$, $1 \leq S_i, m \leq 10^6$

Try it out first before proceeding.

The condition for being "good" is quite peculiar. The "sum of the digits of the gcd mod m" feels like a very random quantity. However, it has a crucial property: it only depends on the gcd of the sequence.

Next, observe that the values S_i are rather small, which means the gcds are also small: there are only 10^6 distinct gcds possible, and we can iterate through them. Combining this with the previous observation, we see that we can solve the problem if we can compute, for each gcd, the number of subsequences with that gcd. Thus, we have reduced the problem to the following: *For each g, $1 \leq g \leq 10^6$, how many subsequences of S have gcd g?* Let's call this number C_g.

Finding the number of subsequences with a given gcd feels hard. Not only do you have to ensure that everything is divisible by g, but you also have to ensure that there are no bigger common factors. So let's look at a simpler problem first: how many subsequences are there where everything is divisble by g? This version is easy: all we need to answer this is the number of elements of S divisible by g, because any subsequence of them works! So if there are M_g such elements, then there are exactly

$$2^{M_g} - 1$$

such nonempty subsequences.

However, this number is not quite C_g, since it counts sequences whose gcds are $> g$. However, since all elements are divisible by g, the gcd can only be $2g, 3g, 4g, \ldots$. Also, the number of sequences with these gcds can be obtained by recursion: they are just $C_{2g}, C_{3g}, C_{4g}, \ldots$. Thus, we have the following recurrence:

$$C_g = 2^{M_g} - 1 - C_{2g} - C_{3g} - C_{4g} - \ldots$$

or, in $\sum$ notation,

$$C_g = 2^{M_g} - 1 - \sum_{d \geq 2} C_{dg}. \tag{2}$$

Alternatively, we can rewrite it as follows:

$$\sum_{d \geq 1} C_{dg} = 2^{M_g} - 1. \tag{3}$$

These equations are equivalent, but the latter has a more direct double-counting proof.

Exercise 4.10. Prove Equation 3 via double-counting.

Anyway, for our purposes, Equation 2 is enough. We can use it for DP. Since every recursive call is for a *larger* g, we must iterate through the gs in decreasing order, and we get the following:[18]

```cpp
...
const ll MOD = 1'000'000'007;
const int V = 1'000'011; // maximum value
...
vector<ll> subseqs_with_gcd(const vector<int>& S) {
    vector<ll> C(V+1), M(V+1), p2(S.size()+1);
    for (int v : S) M[v]++;
    p2[0] = 1;
    for (int i = 1; i <= S.size(); i++) {
        p2[i] = p2[i-1] * 2 % MOD;
    }
    for (int g = V; g >= 1; g--) {
        C[g] = p2[M2[g]] - 1;
        for (int d = 2; d*g <= V; d++) {
            C[g] -= C[d*g];
        }
        C[g] %= MOD;
    }
    return C;
}
```

Unfortunately, this feels slow, because there are $\mathcal{O}(V)$ states (where V is the bound for the values, so $V = 10^6$), and the worst-case transition is $\mathcal{O}(V)$, e.g., for $g = 1$. So this suggests that the running time is $\mathcal{O}(V^2)$. However, this is not tight! The worst-case $\mathcal{O}(V)$ is only triggered by small g; the larger g is, the faster it gets. In the extreme cases like $g > V/2$, the transition running time is actually $\mathcal{O}(1)$ because g has only one multiple $\le V$.

We can express this more precisely: the number g has $\lfloor V/g \rfloor$ multiples $\le g$, so the transition running time is $\mathcal{O}(V/g)$, and the overall running time is

$$\mathcal{O}\left(\frac{V}{1} + \frac{V}{2} + \frac{V}{3} + \ldots + \frac{V}{V}\right).$$

We can factor out the V to get

$$\mathcal{O}\left(V \cdot \left(\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{V}\right)\right).$$

The inner sum is the Vth Harmonic number, which we've proved in an earlier math module to be $\mathcal{O}(\log V)$! Therefore, the running time is simply $\mathcal{O}(V \log V)$, which is fast, and solves the problem!

So this is what I usually think of as "Sieve DP"; it's still DP, but it has a sieve-like feel to it, which if you look at it closer, gives a $V \log V$-ish running time instead of V^2-ish. So be on the lookout if you're iterating through multiples or divisors!

[18] This code can be cleaned up a bit, e.g., by preallocating and reusing arrays, precomputing powers of two, making some variables **static**, etc.. I didn't do any of those to make the code easier to read.

Exercise 4.11. Sometimes, your DP transition iterates through the proper *divisors* of each number instead of its proper *multiples*. Show that the running time remains $\mathcal{O}(V \log V)$ in this case.

Hint: You can precompute the divisors of every number with a simple preprocessing step like this:

```cpp
vector<vector<int>> divs(V+1);
for (int d = 1; d <= V; d++) {
    for (int m = d; m <= V; m += d) [
        divs[m].push_back(d);
    }
}
```

5 Strategy

Dynamic programming is such a useful and widely applicable technique that is simple at its core but requires a lot of practice to get good at, that's why we're dedicating several modules for them. We hope, by doing so, that DP becomes natural for you and becomes one of the first things you consider after coming up with the brute-force/backtracking solution.

5.1 How do I come up with DP solutions?

Practice, practice, practice. Also, practice.

Also, as explained in the LCS example, sometimes, it really just boils down to coming up with the simplest way of "producing" any possible solution using a sequence of *actions*, so the problem reduces to finding the sequence of actions that produces the best result. (or their total, if it's a counting problem.)

So the idea is that you should think about how to *produce* or *construct* any possible solution systematically, while requiring as little memory as possible, i.e., remembering only the things you need to remember in order to continue the construction.

Also, practice! Why aren't you practicing!?

5.2 Good, better, and betterer DP

Not all DP solutions are equal. As we have seen before, different states/transitions yield DP solutions of differing efficiency.

Many times, the constraints in a problem are too large for some DP solutions; in such cases, the problem setter typically eyes a particular time complexity, and attempt to design the constraints and test data so that slower time complexities will not pass. This means that you will need to learn how to optimize DP solutions.

Improving the DP solution means one of the following:

- **Reducing the state**. This usually means removing unnecessary values in the state, for example, if they are actually not needed (like ℓ in the LCS example), or if they can be inferred from the other values. (for example, see **The Room:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/the-room).
- **Optimizing the transition**. Sometimes, this involves using *data structures* to speed up the transition. For example, we might have a recurrence like

  $$f(n) = v_n + \max(f(n-1), f(n-2), \ldots, f(n-k)),$$

  where we are taking some sort of range maximum, but the values of the $f(n)$ are dependent on it. Here, using data structures that support efficient queries and updates (like segment trees) can optimize the transition step. (see **Guardians of the Lunatics Vol. 2:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/guardians-lunatics-2)

  Also, sometimes, optimizing the transition involves finding additional nontrivial observations that you can exploit. For example, some slices of your DP data may be monotonic, so maybe you can speed up your transition with binary search. This is not always easy, but one thing that can help is to try to draw stuff on paper, and also to consider several examples. You can also print out the results of your DP and see if you can find patterns in the data.

- **Coming up with totally different states/transitions.** One can sometimes get stuck with a certain DP solution and fail to improve the running time. In fact, this is an inherent part of some particularly tricky problems; there are “dead-end” ideas which cannot be improved, but the real solution uses totally different ideas. (In the case of DP, totally different states and transitions.)

  Here, the advice is to try to think out of the box, and don’t be afraid to start from scratch once in a while, especially when your initial idea feels “ugly” or “too hacky”.

  This is usually not easy to do, and experience helps a lot. My main advice is to just skip a problem (especially if you’ve already spent too much time on it without making much progress), and maybe return to it later. This forces you to actually leave your idea for a while, and make it easier to go towards different directions later.

- **Striking a balance between state size and transition speed**, at least in some situations, because sometimes, reducing the state results in a slowdown in the transition, and vice versa. (This is just another manifestation of the space-time tradeoff.)

  Sometimes, trading off time with space, or vice versa, doesn’t change much, or just makes things worse. But in other occasions, there’s a *sweet spot*, in which there is a right balance of state size and transition speed that yields the fastest algorithm overall. (This is similar to the “sqrt” part of sqrt decomposition techniques!)

  This happens rarely, but it happens. So it’s good to keep in mind when solving problems with DP, just in case.

5.3 Feels like matrix exponentiation

You now have new tools in your arsenal, techniques which you’ve learned in this module. In addition to the above, you may also want to check if the DP you have can be optimized with *matrix exponentiation* (or its generalizations).

As mentioned before, the things that makes matrix exponentiation techniques work are the following:

- All our recurrences are *linear*, i.e., everything depends on previous values with only constant coefficients. (Note that they can be negative.)
- The transition is “time-blind” or “index-blind”, i.e., independent of the current index i, and
- We only need to remember a “finite window” of things to be able to proceed in our decision-making process.

There are sometimes telltale signs that matrix exponentiation (or a similar divide-and-conquer algorithm) may work in a given problem. For example, in a problem, you might notice that the grid is of size 10×10^{12}. In such a case, I’m usually convinced that some DP state is expected for the small “10” dimension. If the other dimension is large, this is even more of a giveaway that matrix exponentiation/divide-and-conquer is involved. If the “large” dimension is a bit smaller, like 10^5, then maybe you could even skip the matrix part since you can just perform regular DP.

But the other dimension doesn't even need to be large; even if a matrix exponentiation solution exists, the problem setter sometimes doesn't force it, especially if the recurrence is interesting enough on its own. Thus, if one of the dimensions is small, try to see if a state-based DP might help to solve it, even if *both* dimensions are small.

Watch out: Sometimes, the problem setter is sneaky with the constraints! For example, something like $rc \leq 80$ seems to be too much for a DP-based solution with exponentially many states, until you realize that, in fact, this implies that $\min(r, c) \leq 8$.

6 Problems

Some notes:

- Solve as many as you can!
- You may discuss the problems (and the exercises in the text) in Discord.
- I especially invite you to ask us if you get stuck at something, and you feel you shouldn't be.
- Don't be afraid to ask even if you feel your question is stupid!
- Ask me if anything is unclear.
- I'll try my best to give feedback to all submissions.
- Feel free to resubmit, or submit partial work!

6.1 Warmup problems

Here are some of the problems that are discussed or mentioned in the text. Please feel free to practice on them!

W1 Tri Tiling: UVa 10918

W2 Robbery 101: `https://www.codechef.com/problems/INF16B`

W3 The Room: `https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/the-room`

W4 Guardians of the Lunatics Vol. 2: `https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/guardians-lunatics-2`)

6.2 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine. But I suggest using more words to explain, when in doubt. Okay lang kahit Tagalog.

The ones in red are more important/fundamental than the others, so prioritize them.

Please solve at least [111★].

N1 A **walk** in a graph is like a path but where you are allowed to repeat nodes or edges. Formally, a walk of length ℓ is a sequence of $2\ell+1$ nodes and edges, alternating starting with a node, that is of the form

$$a_1, (a_1, a_2), a_2, (a_2, a_3), a_3, \ldots, (a_{\ell-1}, a_\ell), a_\ell.$$

We say that the **starting node** of the walk is a_1, and the **ending node** is a_ℓ.

We are given a simple unweighted directed graph with n nodes.

(a) [8★] Let $w(1, j, \ell)$ be the number of walks of length ℓ that begin at node 1 and end at node j. Find an $\mathcal{O}(n^3 \log \ell)$-time algorithm to compute $w(1, j, \ell)$ for all nodes j.

(b) [14★] Let $w(i, j, \ell)$ be the number of walks of length ℓ that begin at node i and end at node j. Find an $\mathcal{O}(n^3 \log \ell)$-time algorithm to compute $w(i, j, \ell)$ for all nodes i and j.

(c) [4★] Look at your transition matrix. How does it compare to the *adjacency* matrix? Can you explain the connection?

(d) [4★] Let $W^{(\ell)}$ be the $n \times n$ matrix representing the number of walks of length ℓ, that is, $W^{(\ell)}_{i,j}$ is the value $w(i, j, \ell)$. Express $W^{(\ell)}$ in terms of the $n \times n$ adjacency matrix of the graph A, where $A_{i,j} = 1$ if there is an edge (i, j), and 0 otherwise.

(e) [5★] What happens if we ask the same question on a multigraph[19]? Note that the choice of edge to take matters in a walk. How do we compute $W^{(\ell)}$ then? And how does it relate to the adjacency matrix?

(f) [14★] Now, suppose that our directed graph is now weighted. The *weight* of a walk is just the sum of the weights of all edges in it. Let $s(i, j, \ell)$ be the minimum weight of any walk from i to j with length ℓ. Find an $\mathcal{O}(n^3 \log \ell)$ algorithm to compute $s(i, j, \ell)$ for all nodes i and j.

(g) [8★] Let $S^{(\ell)}$ be the matrix of $s(i, j, \ell)$. Express $S^{(\ell)}$ in terms of the weight matrix C of the graph, where $C_{i,j}$ denotes the weight of the edge (i, j), and ∞ if there is no such edge.

N2 [11★] Use recurrence shaking to find a recurrence for $f(n)$ defined by

$$f(n) = F_1 f(n-1) + F_2 f(n-2) + F_3 f(n-3) + F_4 f(n-4) + \ldots + F_n f(0)$$

with base case $f(0)$, where F_n is the nth Fibonacci number: $F_n = F_{n-1} + F_{n-2}$ with base cases $F_0 = 0$ and $F_1 = 1$.

N3 You are given a row of n consecutive unit squares in a $1 \times n$ grid.

(a) [12★] Suppose that you want to tile[20] the cells with either 1×2 dominoes or 1×1 squares. Let $f(n)$ be the number of ways to do that. Find a recurrence for $f(n)$. Explain how to use this to solve the problem in $\mathcal{O}(\log n)$ time.

[19] a *multigraph* is a graph that allows multiple edges and loops.

[20] In a tiling, each cell must be covered by exactly one tile.

(b) [3★] Now, suppose that our only tiles are 1×2 dominoes and 1×3 triominoes. Let $g(n)$ be the number of ways to tile. (Note that $g(1) = 0$ because we don't have a 1×1 tile.) Find a similar recurrence for $g(n)$.

(c) [8★] Now, suppose that our only tiles are 1×2 dominoes and 1×3 triominoes, but triominoes come in two types: black and gold. Let $h(n)$ be the number of ways to tile. Find a similar recurrence for $h(n)$.

(d) [8★] Generalize the algorithm; assume we have k distinct tile types, where the ith tile has dimensions $1 \times d_i$. Find an algorithm that computes the number of tilings of the row of cells that runs in $\mathcal{O}(d_{\max}^3 \log n)$ time.

(e) [3★] Let $F(n)$ be the number of ways to tile a $2 \times n$ grid with (only) 1×2 dominoes. Dominoes may be rotated. Find a recurrence for $F(n)$.

(f) [2★] Explain the relationship between $F(n)$ and the $f(n)$ from (**N3**a).

N4 [16★] We are given k binary strings $s_1, s_2, \ldots s_k$. The *value* of string s_i is v_i. The score of a string is the sum of all values of every occurrence of s_i in it, across all i.

Devise an algorithm that computes the maximum score of any binary string of length n in $\mathcal{O}(64^m \log n)$ time, where m is the longest length of any s_i.

N5 [12★] Let $f(n) = 2f(n-1) + 3f(n-2) + 4f(n-3)$ and $g(n) = 5g(n-1) + 6g(n-2)$. Devise an algorithm that computes

$$f(0)g(0) + f(1)g(1) + \ldots + f(n)g(n)$$

for a given n in time logarithmic in n.[21]

N6 [18★] Suppose that we implement the solution to Problem 4.2 this way:

```
...

int f(int i, int m) {
    if (i == 0 || m == 0) return 0;  // base case

    int ans = s[i], l = left[i], r = right[i];
    for (int t = 0; t <= m; t++) {
        if (t <= size[l] && m - t <= size[r]) {
            ans = max(ans, f(l, t) + f(r, m - t));
        }
    }
    return ans;
}
```

What is the worst-case running time of the algorithm if we implement it this way, assuming it has been memoized? Also, please give at least one kind of tree that triggers this worst case.

N7 [12★] Exercise 2.9

N8 [8★] Exercise 3.3

N9 [8★] Exercise 3.5

[21] *Hint:* You only need a 7×7 matrix.

N10 [12★] Exercise 3.6

N11 [8★] Exercise 3.8

N12 [5★] Exercise 3.9

N13 [12★] Exercise 3.10

N14 [6★] Exercise 3.11

N15 [5★] Exercise 3.20

N16 [4★] Exercise 3.21

N17 [4★] Exercise 4.3

N18 [12★] Exercise 4.8

N19 [8★] Exercise 4.9

6.3 Coding problems

In your attachments, please ensure that it is clear just from the filename which **CX** problem it is meant for.

The ones in red are more important/fundamental than the others, so prioritize them.

Please solve at least [404★].

For Project Euler problems:

- Try to solve the problem for general cases, not just the given single case.
- Your code should run in at most 5 seconds for the given case.

C1 [17★] Implement the solution to Exercise 3.7. (Find only the answer modulo $10^9 + 2$.)

C2 [37★] Implement an algorithm that finds the number of tilings of a $4 \times n$ rectangle with dominoes, modulo $10^8 + 3$.

Hint: Verify that your answers are correct with brute force.

C3 [16★] A t-height-balanced tree is a binary tree in which for every node, its left and right subtrees differ in height by at most t. Implement an algorithm that computes the minimum and maximum number of nodes in a t-height-balanced tree of height h in time polynomial in t and $\log h$. Try to optimize the time complexity as much as you can. Explain your algorithm and running time in the comments.

C4 [25★] **Counting block combinations I:** `https://projecteuler.net/problem=114` (Solve it for general n, with running time logarithmic in n.)

C5 [10★] **Counting block combinations II:** `https://projecteuler.net/problem=115`

C6 [15★] **Red, green or blue tiles:** `https://projecteuler.net/problem=116` (Solve it for general n, with running time logarithmic in n.)

C7 [14★] **Red, green, and blue tiles:** `https://projecteuler.net/problem=117` (Solve it for general n, with running time logarithmic in n.)

C8 [42★] **Triominoes:** `https://projecteuler.net/problem=161`

C9 [37★] **Tri-colouring a triangular grid:** `https://projecteuler.net/problem=189`

C10 [32★] **Coloured Configurations:** `https://projecteuler.net/problem=194`

C11 [41★] **Building a tower:** `https://projecteuler.net/problem=324`

C12 [44★] **Generating polygons:** `https://projecteuler.net/problem=382`

C13 [43★] **Migrating Ants:** `https://projecteuler.net/problem=393`

C14 [38★] **A frog's trip:** `https://projecteuler.net/problem=416`

C15 [28★] **Polynomials of Fibonacci numbers:** `https://projecteuler.net/problem=435`

C16 [41★] **Permutations of Project:** `https://projecteuler.net/problem=458`

C17 [45★] **Jumping frog:** `https://projecteuler.net/problem=490`

C18 [70★] **Eulerian Cycles:** `https://projecteuler.net/problem=289`

S1 [16★] **Python Indentation:** https://codeforces.com/problemset/problem/909/C

S2 [16★] **Let's Go Rolling!:** https://codeforces.com/problemset/problem/38/E

S3 [16★] **Sum of gcd of Tuples:** https://atcoder.jp/contests/abc162/tasks/abc162_e

S4 [16★] **Winter is here:** https://codeforces.com/problemset/problem/839/D

S5 [37★] **Tiling Dominoes:** UVa 11270

S6 [34★] **Backbreaking Breakout:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/backbreaking-breakout

S7 [24★] **Illuminati:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2017-illuminati

S8 [40★] **Elevator:** https://codeforces.com/problemset/problem/983/C

S9 [34★] **Kuro and Topological Parity:** https://codeforces.com/problemset/problem/979/E

S10 [39★] **Robbery 101:** https://www.codechef.com/problems/INF16B

S11 [37★] **Karen and Supermarket:** https://codeforces.com/problemset/problem/815/C

S12 [50★] **Tim and BSTs:** https://www.codechef.com/problems/AMR16A

A Tree convolution DP runs in $\mathcal{O}(n^2)$ time; a (more) formal proof

In this appendix, we will prove the following theorem giving the complexity of the "tree convolution DP" we discussed in subsection 4.2. This also gives you a taste of how solutions to complexity recurrences are usually formally proven.

Theorem A.1. If T satisfies

$$T(i) = T(\ell) + T(r) + \mathcal{O}(\mathit{size}(\ell) \cdot \mathit{size}(r)),$$

then $T(i) = \mathcal{O}(\mathit{size}(i)^2)$.

Proof. By definition, a function $f(L, R)$ is in $\mathcal{O}(LR)$ if and only if there is a constant $c > 0$ such that $f(L, R) \leq c \cdot (L+1)(R+1)$ for any $L, R \geq 0$.[a]

With this, we can now rephrase the condition of the theorem as

$$T(i) \leq T(\ell) + T(r) + c \cdot (\mathit{size}(\ell)+1)(\mathit{size}(r)+1)$$

for some $c > 0$.

To prove the theorem, it is sufficient to show that, for all nodes i,

$$T(i) \leq d \cdot (\mathit{size}(i)+1)^2$$

for $d = \max(c, T(\mathit{null}))$. We prove this using structural induction (or induction on $\mathit{size}(i)$ if you like).

- For the base case, we have $T(\mathit{null}) \leq d = d \cdot (\mathit{size}(\mathit{null})+1)^2$, so it holds.
- For the inductive case, suppose that the hypothesis is true for the two children of ℓ and r. Then

$$\begin{aligned}
T(i) &\leq T(\ell) + T(r) + c \cdot (\mathit{size}(\ell)+1)(\mathit{size}(r)+1) \\
&\leq d \cdot (\mathit{size}(\ell)+1)^2 + d \cdot (\mathit{size}(r)+1)^2 + c \cdot (\mathit{size}(\ell)+1)(\mathit{size}(r)+1) \\
&= d \cdot \Big[(\mathit{size}(\ell)+1)^2 + (\mathit{size}(r)+1)^2 + c/d \cdot (\mathit{size}(\ell)+1)(\mathit{size}(r)+1)\Big] \\
&\leq d \cdot \Big[(\mathit{size}(\ell)+1)^2 + (\mathit{size}(r)+1)^2 + 2 \cdot (\mathit{size}(\ell)+1)(\mathit{size}(r)+1)\Big] \\
&= d \cdot [(\mathit{size}(\ell)+1) + (\mathit{size}(r)+1)]^2 \\
&= d \cdot [(\mathit{size}(\ell) + \mathit{size}(r) + 1) + 1]^2 \\
&= d \cdot (\mathit{size}(i)+1)^2,
\end{aligned}$$

  which proves the inductive hypothesis.

□

[a] You might contend that $\mathcal{O}$ only requires inequality for *sufficiently large* arguments, but we can easily prove that my statement is equivalent: There are only finitely many arguments that are *not* sufficiently large, so $f(L, R)$ must have a finite upper bound among them. And since $(L+1)(R+1) \geq 1$, we have $c \leq c \cdot (L+1)(R+1)$, so we can just choose c to be large enough to absorb this finite bound.