Summer SEMs

Subtask 1

Claim: The square-error minimizer of any non-empty sequence is just its **average** value.

Proof. Let $f(x)$ be the total square error of x with respect to some sequence $[v_1, v_2, \dots, v_n]$, so

$$f(x) = \sum_{i=1}^{n} (x - v_i)^2.$$

Let's do some manipulations on this summation, starting with expanding out the square in each summand:

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} (x - v_i)^2 \\ &= \sum_{i=1}^{n} (x^2 - 2xv_i + v_i^2) \\ &= x^2 \sum_{i=1}^{n} 1 - 2x \sum_{i=1}^{n} v_i + \sum_{i=1}^{n} v_i^2 \\ &= nx^2 - 2\left(\sum_{i=1}^{n} v_i\right) x + \sum_{i=1}^{n} v_i^2. \end{aligned}$$

Recall that the values $[v_1, v_2, \dots, v_n]$ are all constant with respect to f. Thus, f is actually a **quadratic polynomial** in terms of x.

We know that if $a > 0$, then the value of the quadratic $ax^2 + bx + c$ is minimized at its vertex, when $x = -b/2a$. Since $n > 0$, that tells us that the minimum of $f(x)$ is achieved at

$$x = \frac{-\left(-2\left(\sum_{i=1}^{n} v_i\right)\right)}{2n},$$

or,

$$x = \frac{\sum_{i=1}^{n} v_i}{n},$$

which, familiarly, is the average of the sequence. ■

Remark: This is actually the primary mathematical reason for why averages are so ubiquitous in probability and statistics! It being the square-error minimizer gives us a rigorous notion of what it *means* to "be in the center" of a set of points.

Note that this gives us our proof that $n! \times \mathrm{SumSEMs}(a)$ is always an integer. Each summand is an average, meaning it looks like (sum of integers)$/k$, where k is the length of a sublist. Since $1 \le k \le n$, we are guaranteed that $n!/k$ is an integer, and therefore all summands become integers after multiplying by $n!$.

It may be a bit tedious, but for the first subtask, you can (with the aid of a calculator) compute a_1, a_2, a_3, a_4, a_5, and then find the averages of each of its sublists. There are only $5(5-1)/2 = 10$ sublists, so this may be a bit tedious (especially since the numbers are big), but with a calculator, it's still doable to compute this by hand.

Subtask 2

The rest of this tutorial assumes a familiarity with working with multiplicative inverses modulo a prime, in order to handle terms like $1/k$ modulo 998244353. If you're not familiar, you can Google those keywords for resources; the author would like to suggest his writeup here.

For all remaining subsections, we will also assume that the values $[a_1, a_2, \dots, a_n]$ have just been directly computed by your computer using a simple loop.

For this subtask, our approach is to just directly implement our "get the average of all sublists" brute force so that a computer can do it for us. You could do something like this:

```
# pseudocode

n = 1600
MOD = 998244353

a = [...]              # generate a
n_fac = ...            # precompute n! mod MOD
mult_inverse = [...]  # precompute mod mult. inverses; can't just do 1/k, remember

ans = 0
for l in 1, 2, 3, ..., n:
    for r in l, l+1, ..., n:
        subtotal = 0
        for i in l, l+1, ..., r:
            subtotal += a[i]
        subtotal %= MOD

        ans += subtotal * mult_inverse[r-l+1] % MOD
        ans %= MOD
print(ans * n_fac % MOD)
```

How many operations are performed by this algorithm? It's a bit more nontrivial to analyze, but you could say something like this:

- It iterates over all sublists of the sequence
- To process a sublist whose length is some k, this solution perform $\sim k$ operations (to go over the elements one-by-one and add them add together)
- There are $n - k + 1$ sublists whose length is k
- Thus, the total number of operations being performed is roughly

$$\sum_{k=1}^{n} k(n-k+1) = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}.$$

Many computer scientists would be comfortable with just *bounding above* the efficiency of their program:

- Note that there are three nested loops, where each of ℓ and r and i could go from 1 to n (the inner loops are more restricted than this, but that's okay, we're just putting an upper bound). The innermost loop only does one addition and modulo operation.
- Thus, the total number of operations being performed is $< n \times n \times n = n^3$.

Computer scientists are typically happy with a bound like this, since it's good enough for rule-of-thumb estimates—both analyses give the running time as a cubic function of n, so the estimated amount of work should be roughly the same, up to a constant factor. More importantly, the *magnitudes* will be the same, so we can get a ballpark-estimate of how long this would take to run.

If $n = 1600$, then $n^3 \approx 4 \times 10^9$, so even with a slow language like Python, it seems likely that we are within the terminate-within-one-minute ballpark.

Subtask 3

Thinking like a computer scientist, our goal is to bring our running time down from a *cubic* function of n down to a *quadratic* function of n. That should dramatically speed up the efficiency of our program, enough that we could claim the points for the third subtask.

Our analysis revealed that the cubic running time comes from the fact that there are three nested loops. If we can shave off one of those loops, then we're good.

The key insight is to realize that there is a lot of *overlap* between the sums we're computing, so it's rather wasteful to start from scratch every time. Let `range_sum(l, r) = a[l] + a[l+1] + ... + a[r]`. Note that

```
range_sum(l, r) = range_sum(l, r-1) + a[r].
```

In words, if you already have a partial sum and want to include the next element r in the sum, then you *don't have to restart the entire sum from scratch*; you can just add `a[r]` to what you have so far.

This gives rise to the following solution:

```
# pseudocode

n = 16000
MOD = 998244353

a = [...]              # generate a
n_fac = ...            # precompute n! mod MOD
mult_inverse = [...]  # precompute mod mult. inverses; can't just do 1/k, remember

ans = 0
for l in 1, 2, 3, ..., n:
    subtotal = 0
    for r in l, l+1, ..., n:
        subtotal += a[r]
        subtotal %= MOD

        ans += subtotal * mult_inverse[r-l+1] % MOD
        ans %= MOD
print(ans * n_fac % MOD)
```

You can check that for each value of ℓ and r, the value of `subtotal` at that point in the loop is precisely `a[l] + a[l+1] + ... + a[r]` using the partial sums (or cumulative sums) idea.

By similar analysis as in subtask 2, this algorithm performs $\sim n^2/2 + n/2$ operations. Or, more simply, the number of operations is bounded above by $\sim n^2$. Either case gives us a quadratic function. Since $16000^2 = 256000000$, even a slow language like Python should finish within a minute.

Subtask 4

Again thinking like a computer scientist, our goal is to bring our running time down from a *quadratic* function of n down to some subquadratic function, such as one *linear* in n.

There are $\approx n^2/2$ sublists of a sequence of length n, which should inform our approach and way of thinking. Fundamentally, we **cannot** any more use solutions that examine each sublist of a one-by-one.

Instead, we have to start thinking about the individual elements of a. For example, maybe this works: for each a_i, what is its contribution to the final sum? Let's try writing out an example with $n = 6$:

$$\begin{aligned}
\text{SumSEMs}(a) &= \frac{a_1}{1} + \frac{a_2}{1} + \frac{a_3}{1} + \frac{a_4}{1} + \frac{a_5}{1} + \frac{a_6}{1} \\
&+ \frac{a_1 + a_2}{2} + \frac{a_2 + a_3}{2} + \frac{a_3 + a_4}{2} + \frac{a_4 + a_5}{2} + \frac{a_5 + a_6}{2} \\
&+ \frac{a_1 + a_2 + a_3}{3} + \frac{a_2 + a_3 + a_4}{3} + \frac{a_3 + a_4 + a_5}{3} + \frac{a_4 + a_5 + a_6}{3} \\
&+ \frac{a_1 + a_2 + a_3 + a_4}{4} + \frac{a_2 + a_3 + a_4 + a_5}{4} + \frac{a_3 + a_4 + a_5 + a_6}{4} \\
&+ \frac{a_1 + a_2 + a_3 + a_4 + a_5}{5} + \frac{a_2 + a_3 + a_4 + a_5 + a_6}{5} \\
&+ \frac{a_1 + a_2 + a_3 + a_4 + a_5 + a_6}{6}
\end{aligned}$$

If we rearrange this sum so that we collect the like a_i terms:

$$\begin{aligned}
\text{SumSEMs}(a) &= \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}\right) a_1 \\
&+ \left(\frac{1}{1} + \frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \frac{1}{6}\right) a_2 \\
&+ \left(\frac{1}{1} + \frac{2}{2} + \frac{3}{3} + \frac{3}{4} + \frac{2}{5} + \frac{1}{6}\right) a_3 \\
&+ \left(\frac{1}{1} + \frac{2}{2} + \frac{3}{3} + \frac{3}{4} + \frac{2}{5} + \frac{1}{6}\right) a_4 \\
&+ \left(\frac{1}{1} + \frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \frac{1}{6}\right) a_5 \\
&+ \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}\right) a_6.
\end{aligned}$$

That looks like it has a stunning amount of structure that we can leverage! If we examine only the numerators, we get the following grid pattern:

$$\begin{matrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 2 & 2 & 2 & 1 \\
1 & 2 & 3 & 3 & 2 & 1 \\
1 & 2 & 3 & 3 & 2 & 1 \\
1 & 2 & 2 & 2 & 2 & 1 \\
1 & 1 & 1 & 1 & 1 & 1
\end{matrix}$$

Note that in the grid above:

- The top half is a mirror image of the bottom half
- For i such that $2 \le i \le n/2$, the next row can be found by taking the previous row and then "adding +1" to some contiguous band of values.

We'll leave rigorously proving this pattern as an exercise to you. The numerator in the ith row's something/k is given by *counting* the number of times that a_i appears in a sublist of length k.

More formally, let c_i be "the coefficient of a_i". For convenience, let $c_0 = 0$. Then, for $1 \le i \le n/2$:

$$c_i = c_{i-1} + \left(\frac{1}{i} + \frac{1}{i+1} + \cdots + \frac{1}{n-i+1}\right)$$

Finally, computing these sums of reciprocals can also be done using running sums. Let r_i be the value added to c_{i-1} at each step, so for $1 \le i \le n/2$:

$$r_i = \frac{1}{i} + \frac{1}{i+1} + \cdots + \frac{1}{n-i+1}.$$

Note that $r_1 = 1/1 + 1/2 + 1/3 + \cdots + 1/n$. Then, for $2 \le i \le n/2$:

$$r_i = r_{i-1} - \frac{1}{i-1} - \frac{1}{n-(i-1)+1}.$$

All in all, we get the following solution. Here, `coeff` and `row_delta` are variables such that for each i in the loop, `coeff` $= c_i$ and `row_delta` $= r_i$ is always maintained.

```
# pseudocode

n = 16 * 10**6
MOD = 998244353

a = [...]              # generate a
n_fac = ...            # precompute n! mod MOD
mult_inverse = [...]  # precompute mod mult. inverses; can't just do 1/k, remember

ans = 0
coeff = 0
row_delta = None
for i in 1, 2, 3, ..., n//2:
    if i == 1:
        row_delta = mult_inverse[1] + mult_inverse[2] + ... + mult_inverse[n]
    else:
        row_delta -= mult_inverse[i-1]
        row_delta -= mult_inverse[n-(i-1)+1]

    coeff += row_delta

    ans += a[i] * coeff % MOD
    ans += a[n-i+1] * coeff % MOD  # for the bottom half
    ans %= MOD

print(ans * n_fac % MOD)
```

There is now only a single loop from 1 to n that is doing a handful of operations at each iteration. Thus, the number of operations performed by this solution should be linear (i.e., directly proportional) with respect to n, and so it should be fast enough even when $n = 1.6 \times 10^7$.