IOI Squad May Contest, 2009 Solutions

IOI Squad 2009

1 Jelly

At a glance this problem is similar do the corridor-tiling problem in the second round 2008. But there's a catch: not only to you need to worry about segments of different colours, you also need to make sure that arrangements using different segments of the same colour are only counted once.

The 30% solution is simply brute force: Try every possible arrangement and store them in a set so that no arrangement is counted more than once. The number of elements in the set can then be counted and the number of palindromic elements can be found by iterating through the set.

However, unsurprisingly, the real solution uses dynamic programming like the tiling problem. Let $f(n)$ denote the number of designs with length n. The first recurrence that comes to mind is $f(n) = C \cdot (f(n-2) + f(n-3))$, but this counts arrangements like RRR | RRR and RR | RR | RR separately. So we subtract the number of arrangements that would be counted twice. These are the arrangements that have identical colours up to position $n-3$, but may use different segments to get there. This turns out to be the definition of $f(n-3)$ so that is what we subtract.

The final recurrence is $f(n) = C \cdot f(n-2) + (C-1) \cdot f(n-3)$. This alone will get you 50% and combining it with the brute-force for smaller N will get 65%.

Dealing with palindromic arrangements requires quite a bit of thought. It helps to think of a mirror being placed in the middle of the tube and reflecting the first half of the design. The cases where N is odd and even need to be considered separately, as when N is odd, the mirror lies on top of a jelly tot and when N is even, it lies between two jelly tots.

Let's first consider N being odd. The number of palindromic arrangements would be the designs up to the middle element, that can also be found if the middle element were removed. It turns out that every design can still be found if the middle element were removed. Therefore the number of arrangements is $f(\lceil N/2 \rceil)$.

For even N, it is more complicated. For odd N, we didn't need to consider segments that overlapped the half-way point as 3-jelly-tot segments became 2-jelly-tot segments and 2-jelly-tot segments could be declared part of the reflected side. For even N, we add the number of ways that 3-jelly-tot segments can overlap the half-way point. Since these start at $N/2 - 1$ and can be any colour

there are $f(N/2-1) \cdot C$ such arrangements. But again, we have to discard the designs that can be found in this way and by simply working with two separate halves, as these have been counted twice. The final number of palindromic arrangements for even N is $f(N/2) + f(N/2-1) \cdot (C-1)$.

All of this will get you 100%. Just be carefull of one cruel catch. When adding the total number and number of palidromic designs and dividing by two, you are dividing over a modulus. What if the modulus were 5 and the dividend were 6 mod 5? Dividing 1 by 2 gives 0, not three as expected. Instead multiply the dividend by the inverse of 2 mod 5. This turns out to be the modulus plus one over two. Then taking the modulus of the final result will yeild the final answer. Failing to do this gets 75%. Sorry :-(.

2 Tiling Mall

2.1 Using Matrices

Now the solution for this problem is DP and with the constraints at a rather low 5 000 000 you should be easily be able to run any one dimensional DP in time. Now, however, let's suppose the contraints we're set at something much higher like 10 000 000 or even worse. Here even our DP's $O(N)$ running time fails us. So we need another solution.

The solution is to use matrices.[1] Using matrices allow us to achieve a much faster running time of $O(\log N)$. This will easily run in time, but it takes a bit more effort to code up and come up with.

The first step to solving the problem using matrices is to solve it as you would normally solve a DP problem. But that is not the scope of this solution, so I'll simply present the solution here.

Let A_n represent the number of possibilities for the following shape, with n being the shape's length from the leftmost to rightmost point:

Similarly for B_n:

And C_n:

[1] See Graham's Lecture on Linear Recurrences at the 2nd Camp 2009: `http://olympiad.cs.uct.ac.za/presentations/camp2_2009/lin_recurrences.pdf`

And D_n:

And finally E_n:

Then you get the following recurrence relationships for the above terms:

- $A_n = B_n + C_n$
- $B_n = A_{n-1} + B_{n-1}$
- $C_n = E_n + D_{n-1}$
- $D_n = A_{n-1} + D_{n-2}$
- $E_n = A_{n-2} + B_{n-1}$

And you have the following base states:

- $A_0 = A_1 = 1$
- $B_0 = 0$ and $B_1 = 1$
- $C_0 = C_1 = 0$
- $D_0 = 0$ and $D_1 = 1$
- $E_0 = E_1 = 0$

So now we have all the information we need for the normal DP solution and just and just have to construct the matrices, but the recurrence relationships we currently have are in an unfavourable form for this, consider $A_n = B_n + C_n$. This makes setting up the matrices really difficult, because you can't calculate and use A_n, B_n and C_n in the same step, so you somehow have to calculate the latter two in the step before the current one and this can get messy. The solution to this is to simply play around with the equations a bit.

The first change we make is to get rid of the E_n state entirely, by substituting it into the only term it is used, C_n. We need to do this or else we will have to calculate both C_n and E_n in the same step or somehow have to calculate E_n in the previous step. Getting rid of a term is also a very minor optimisation and thus we killed 2 birds with 1 stone. So we get:

$$C_n = A_{n-2} + B_{n-1} + D_{n-1}$$

The next thing we can and have rewrite is A_n. So

$$\begin{aligned} A_n &= (A_{n-1} + B_{n-1}) + (A_{n-2} + B_{n-1} + D_{n-1}) \\ &= A_{n-1} + A_{n-2} + 2B_{n-1} + D_{n-1} \end{aligned}$$

What you should also note now is that you can throw away the C_n term as it is no longer being used in any of the other recurrence relationships, so you now only have A_n, B_n and D_n. But you don't have to throw it away as the only penalty is a few extra calculations and a bit more memory; your results will still be correct.

Now all your recurrence relationships rely only on previous states, so things are good. You shouldn't be worried by the fact that you have both A_{n-1} and A_{n-2} in your recurrence relationships, to get around that you just store both of them in your current state.

So now let's get to the format of our current state, S_n, which be in the form of a 5×1 matrix:

$$S_n = \begin{bmatrix} A_n \\ A_{n-1} \\ B_n \\ D_n \\ D_{n-1} \end{bmatrix}$$

Let's name our recurrence matrix M, note that it has to be of size 5×5 and has to satisfy:

$$S_n = M \times S_{n-1}$$

From the recurrence relationships and our definitions of the current state we can easily deduce the value of M. If you are struggling with this, take some time and consider the definition of matrix multiplication, playing around with it until it makes sense.

$$M = \begin{bmatrix} 1 & 1 & 2 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Now S_1 acts as our base case:

$$S_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

So we finally end up with:

$$S_n = M^{N-1} \times S_1$$

Then the answer to the original problem is A_n as defined in S_n.

Note: For your solution to run in $O(\log N)$ you need to use a power function that also runs in $O(\log N)$.

2.1.1 Implementation Problems and Tips

This section details some tips for writing up the solution and some problems you might encounter while writing up the solution. This section is focussed mainly on C++, but some of the advice might be useful in other languages as well.

Tips:

- Instead of inputting your values for S_1 and M matrix directly into the matrices using their accessors, rather store them in a native array that's visually formatted to look like a matrix in maths. This way you can easily check if the values you inputted are correct and can also easily correct them.
- If $N = 1$ the power function you use or implemented might not work correctly, as it would need a 5×5 identity matrix. Rather than constructing an identity matrix you can just explictly test for this case and just use the S_1 as is.
- Use the power function in <ext/numeric> as it's simple to use and runs in $O(\log N)$.

Potential Problems:

- If you use a destructor, as you'll likely use dynamically allocated memory, remember to write a copy constructor and assignment operator.
- As $1\,000\,007^2$ is bigger than a normal int, remember to either store your matrix values as long longs or convert them to long longs before doing your multiplication, modulus by $1\,000\,007$ and convert them back to normal ints.
- Remember to modulus your values by $1\,000\,007$ after every calculation to prevent overflows and to get the correct answer.