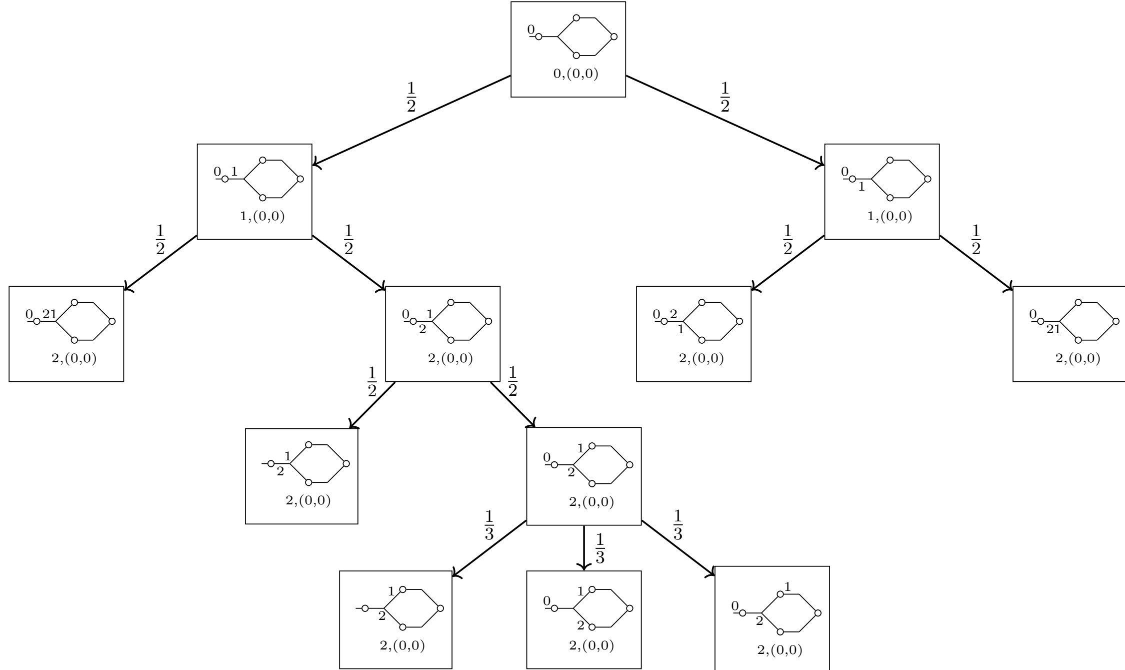
Problem 1: State Graph We use an ad-hoc visualization of the network state. We only show an induced subgraph on 12 nodes of the full state graph. (Note that the full state graph is *not* a tree, but a more general DAG.) We use packet id 0 to refer to the packet that is initially in H0's input queue. The state of the local variables

$$\texttt{pkt_count@H0}, (\texttt{pkt_count@H1}, \texttt{swapped@H1})$$

is given below the illustration of the states of the input and output queues.



Problem 2: Probabilistic Inference using Dynamic Programming The probability that a given path through the state graph is taken is equal to the product of all transition probabilities along the path. One way to compute the probability that some end state is reached from the initial state is to compute a sum over all paths from the starting state to the end state. However, this can be very costly, as there can be exponentially many such paths.

Instead, we can compute the probability of being reached for each node in turn, in topological order according to the DAG. The initial node has probability 1 of being reached, and all other such probabilities can be computed recursively as follows:

$$\Pr[\text{reach state } s] = \sum_{s' \in \Gamma^-(s)} \Pr[\text{reach state } s'] \cdot p_{s' \to s}.$$

Here, $\Gamma^-(s)$ denotes the set of direct predecessors of state s and $p_{s' \to s}$ is the transition probability on the edge (s', s).

Problem 3: Encoding in PSI See `sol3.psi` and `sol3_observe.psi`.

Invoking PSI gives:

```
1 $ psi --noboundscheck --dp --expectation sol3.psi
2 𝔼[swapped] = 1/8
3 $ psi --noboundscheck --dp --expectation sol3_observe.psi
4 𝔼[swapped] = 1/4
```

If we observe that both packets take different paths, the probability that they will be reordered increases. This is because it is impossible for the packets to be reordered if they are forwarded along the same path.

Problem 4: State Graph Size Note that packets can move around the network arbitrarily, so the goal is to compute the number of ways to put the k packets into the $2n$ input and output queues of capacity c. For subtasks 1.–4., we have $c = 2$.

1. Number of packets $k = 1$. There are $2n$ queues that might hold it. Each of them can store packets associated with $(n-1)$ ports. There are $2n \cdot (n-1)$ ways to store the packet.
2. Number of packets $k = 2$. Two cases:
   a) Both packets in different queues: $2n \cdot (2n-1) \cdot (n-1)^2$ ways.
   b) Both packets in the same queue: $2n \cdot 2 \cdot (n-1)^2$. ways. (Factor 2, because we need to order the packets.)

   Total: $2n \cdot (2n+1) \cdot (n-1)^2$ ways.
3. Number of packets $k = 3$. Two cases for three packets:
   a) All packets are in different queues: $2n \cdot (2n-1) \cdot (2n-2) \cdot (n-1)^3$ ways.
   b) Two packets share the same queue: $3 \cdot 2n \cdot 2 \cdot (2n-1) \cdot (n-1)^3$ ways. (3 ways to pick the two packets that are in the same queue.)

   We can reach all states with 2 or 3 packets (due to packet dropping on congestion).
   Total: $2n \cdot (2n+1) \cdot (n-1)^2 + 2n \cdot (2n-1) \cdot (2n+3) \cdot (n-1)^3$ ways.
4. Number of packets $k = 4$. Three cases for four packets:
   a) All packets are in different queues: $2n \cdot (2n-1) \cdot (2n-2) \cdot (2n-3) \cdot (n-1)^4$ ways.
   b) Two packets share the same queue: $6 \cdot 2n \cdot 2 \cdot (2n-1) \cdot (2n-2) \cdot (n-1)^4$ ways. (6 ways to pick the two packets that are in the same queue.)
   c) Two pairs of packets each share the same queue: $6 \cdot 2n \cdot 2 \cdot 2 \cdot (2n-1) \cdot (n-1)^4$ ways. (6 ways to choose an ordered pair of disjoint sets of 2 packets.)

We can reach all states with 2, 3 or 4 packets (due to packet dropping on congestion).
Total: $2n \cdot (2n+1) \cdot (n-1)^2 + 2n \cdot (2n-1) \cdot (2n+3) \cdot (n-1)^3 + (2n \cdot (2n-1) \cdot (2n-2) \cdot (2n-3) + 6 \cdot 2n \cdot 2 \cdot (2n-1) \cdot (2n-2) + 6 \cdot 2n \cdot 2 \cdot 2 \cdot (2n-1)) \cdot (n-1)^4$ ways.

5. There is a simple recurrence for the number that we need. Let $f_c(q,p)$ be the number of ways to fill q queues of capacity c with p packets. We have

$$f_c(q,p) = \begin{cases} 0, \text{ if } q < 0 \text{ or } p < 0, \\ 1, \text{ if } q \geq 0 \text{ and } p = 0, \text{ and} \\ \sum_{i=0}^{c} \binom{p}{i} \cdot i! \cdot f_c(q-1, p-i) \text{ otherwise.} \end{cases}$$

The recurrence makes a case distinction on the number of packets i that are placed into the last queue. It then selects which packets to put into the last queue (factor $\binom{p}{i}$), and orders them within the queue factor $i!$, then, it places the remaining $p-i$ packets into the remaining $q-i$ queues. The answer is $\sum_i [\min(c,k) \leq i \leq k] f_c(2n,i) \cdot (n-1)^i$. Using DP, this solution runs in time $O(n \cdot k \cdot c)$.
Note: *You are not expected to be familiar with the material below at the exam.*

Faster solutions with FFT. Alternatively, we can also first compute the number of ways to choose occupied spots in queues, and only then decide in which order we put the packets into occupied spots.
We will need to count the number of ways to partition k into $2n$ (ordered) numbers between 0 and c.
This number is the coefficient of x^k (written $[x^k]p(x)$) in the following polynomial:

$$p(x) = \left(\sum_{i=0}^{c} x^i \right)^{2n} = \left(\frac{x^{c+1} - 1}{x - 1} \right)^{2n}.$$

We can evaluate the polynomial at a point in time $O(\log(cn))$ using fast exponentiation. The degree of the polynomial is $2cn$. We can extract $[x^k]p(x)$ in time $O(cn \cdot \log(cn))$ by evaluating the polynomial at $2cn+1$ complex roots of unity:

$$[x^k]p(x) = \frac{1}{2cn+1} \sum_{j=0}^{2cn} e^{\frac{-2\pi ijk}{2cn+1}} \cdot p(e^{\frac{2\pi ij}{2cn+1}}).$$

Intuitively, this is a Fourier transform (on an implicit polynomial, which we can compute at points efficiently using exponentiation by squaring), followed by a single component of an inverse Fourier transform (explicitly recovering the k-th coefficient of the implicit input polynomial).

(For those not familiar with Fourier transforms: the Fourier transform evaluates a given degree-($\leq N$) polynomial on all N-th complex roots of unity (in our case, $N = 2cn+1$), and the inverse Fourier transform computes the polynomial given those values. Here, the composite linear transformation $f_k(p) = \frac{1}{2cn+1}\sum_{i=0}^{2cn} e^{-\frac{2\pi ik}{2cn+1}} \cdot p(e^{\frac{2\pi i}{2cn+1}})$ from degree-($\leq 2cn+1$) polynomials to real numbers acts the following way on the standard basis polynomial $b_l(x) = x^l$ for $l \in 0,\ldots,2cn+1$:

$$f(b_l) = \frac{1}{2cn+1}\sum_{j=0}^{2cn} e^{\frac{-2\pi ijk}{2cn+1}} \cdot (e^{\frac{2\pi ij}{2cn+1}})^l = \frac{1}{2cn+1}\sum_{j=0}^{2cn} e^{\frac{-2\pi ijk}{2cn+1}} \cdot e^{\frac{2\pi ijl}{2cn+1}}$$

$$= \frac{1}{2cn+1}\sum_{j=0}^{2cn} e^{\frac{2\pi ij(l-k)}{2cn+1}} = [k = l].$$

Therefore, for $p(x) = \sum_{l=0}^{2cn} a_l \cdot x^i$, we have

$$f_k(p) = f_k(\lambda x. \sum_{l=0}^{2cn} a_l \cdot x^l) = \sum_{l=0}^{2cn} a_l \cdot f_k(\lambda x. x^l) = \sum_{l=0}^{2cn} a_l \cdot [k = l] = a_k.)$$

The result is $\sum_i [\min(c,k) \leq i \leq k][x^i]p(x) \cdot i! \cdot (n-1)^i$, accounting for packet dropping and all ways to order the packets within the occupied spots.
The running time of this solution is $O((k - \min(c,k)+1) \cdot cn \cdot \log(cn))$, but we can easily optimize it to $O(cn \cdot \log(cn))$ by using the fast Fourier transform to compute simultaneously all coefficients $[x^i]p(x)$ that we need to sum, in time $O(cn \log cn)$. This is however a bit wasteful for large c, as we are interested in at most the first $k+1$ coefficients of the degree-(cn) polynomial. Note that we can multiply two degree-k polynomials p and q in time $O(k \log k)$ by applying the fast Fourier transform:

$$[x^l]p(x)q(x) = \frac{1}{2k+1}\sum_{j=0}^{2k} e^{\frac{-2\pi ijl}{2k+1}} \cdot p(e^{\frac{2\pi ij}{2k+1}}) \cdot q(e^{\frac{2\pi ij}{2k+1}}).$$

This is a degree-($2k$) polynomial. We can obtain a degree-k polynomial r by truncating:

$$r(x) = \sum_{j=0}^{k} x^j \cdot [x^j]p(x)q(x).$$

Taking one such truncated product therefore takes time $O(k \log k)$ using FFT. Recall that we need to compute the first $k+1$ coefficients of the polynomial

$$p(x) = \left(\sum_{i=0}^{c} x^i\right)^{2n}.$$

Note that instead of taking products during exponentiation, we can take truncated products at each step, as coefficients of higher powers of x never flow back into coefficients of lower powers of x. Therefore, using exponentiation by squaring using truncated products, we can obtain all relevant coefficients by computing $O(\log n)$ truncated products. The running time of the entire approach is $O(k \log k \log n)$.