Problem Title Description. Problem Name: Denethor's Decryption of Dequeue Permutations Author's Name: Dequeue Permutations Problem Code: DQPERMS Alphabet: F

Problem: The problem statement itself (https://www.spoj.com/problems/AMR12F/), despite being long, gives a clear understanding of the problem description (having examples etc.). For completeness, we reiterate the problem description here:

Define a dequeue permutaiton of N as follows. Begin with a double-ended queue ("dequeue") having the numbers 1, 2, 3, ..., N, in this order. Now, start emptying the dequeue, either from the front or the back. Any such resulting sequence is a "dequeue permutation of N". Examples: 1-2-3-...-N. N-1-2-3-...N-1. etc. are some examples.

Also, we consider the lexicographic ordering of these dequeue permutations. Lexicographic ordering is given 2 dequeue permutations A and B, A ¡ B iff at the first position (pos) where they differ, A[pos] ¡ B[pos]. So you have: (1-2-3-...-N) < (1-2-3-...-N-$N-1$) < ... < (N-1-2-...-$N-1$) < ... < (N-$N-1$-...-3-2-1).

The question now is, given a (possible) partial dequeue permutation, determine the least and largest possible (0-based) index of the partial dequeue permutation in the lexicographic ordering of all possible such permutations. Also, this partial information only comes in the first half of the permutation (and you will see why this is important). Also, this information is given incrementally, and you must determine your answer at each point of time.

(Recommended) Thought Process: The first thing to do when looking at this problem, is trying to gather some understanding of the *structure* of a dequeue permutation of N, as well as the lexicographic ordering. Try some of these intuitive questions: 1. How many dequeue permutations of N are there? 2. By choosing a larger number than a smaller one, how many dequeue permutations am I skipping in the lexicographic ordering? 3. What information am I getting about the permutation when I say "The i-th position has value j"?

Note that the 2nd and 3rd question described above are truly intuitive and indeed necessary to solving the problem (since this is what the problem is asking). The 1st question helps you answer question 2, by noticing that it asks "how many dequeue permutations are there having the smaller number chosen at this point" and that the remaining numbers can be cast into the range (1 to X), where X is the number of numbers remaining after making this "choice".

Now, lets proceed to answer the questions one by one.

Q. How many Dequeue Permutations of N are there? Apart from the formal calculation/proof described later, you could try to fill in dequeue

permutations for small N, say N upto 4. This is what you get: $N = 1$: (1) $N = 2$: (1-2) and (2-1) $N = 3$ (eg. from statement): (1-2-3), (1-3-2), (3-1-2), (3-2-1) $N = 4$: (1-2-3-4), (1-2-4-3), (1-4-2-3), (1-4-3-2), (4-1-2-3), (4-1-3-2), (4-3-1-2), (4-3-2-1). *Note that the above have been given in lexicographic ordering as well.* So the number you get, as function of N, is as follows: $N = 1$: 1 $N = 2$: 2 $N = 3$: 4 $N = 4$: 8 and so on.

At this point, you could probably guess that the answer to the question required is 2^{N-1}. This is indeed the answer, and while it is possible to *guess* this by enumeration, *proving* it would give you much idea of how to *count* dequeue permutations lexicographically.

Proof: Note that, at each point of time, *as long as* the dequeue has *more than one* element, choosing to pop_front() would give a different permutation as choosing to pop_back(). At the end, when you have just one element, pop_front() and pop_back() does not give you anything different. So, for each sequence of first $N - 1$ choices of pop_front()/pop_back(), you get a different dequeue permutation, hence the total number $= 2^{N-1}$.

A useful thing at this point, would be to establish a bijection between Front/Back pops' sequences of length $N - 1$ to dequeue permutations of N. i.e. 1-2-3-...-N maps to FFFF...F, 1-2-3-...-N-$N - 1$ maps to FFFF...FBF. N-1-2-3-...-$N - 1$ maps to BFFF...F. N-$N - 1$-...3-2-1 maps to BBB...B.

Q. By choosing a larger number than a smaller one, how many dequeue permutations am I skipping in the lexicographic ordering? This question basically says, suppose I have done some 'k' operations, (i.e., my F/B-string looks like some k length F/B string followed by $N - 1 - k$???s), now how many dequeue permutations am I skipping by choosing the larger number instead of the smaller number? After seeing the proof of the earlier fact, we easily see that : After putting an F instead of a B, I will still have an $(N - 1 - k - 1)$-length string to be filled in. So the answer is 2^{N-k-2}. Further, it is clear that putting a "B" in the string is worse than putting an "F" in it, and it leads to the following interesting bijection. *Map a pop_back() to a '1' bit and a pop_front() to a '0' bit. Now the $N-1$-length bit sequence you get, when looked at an $N-1$ bit number, is in fact the index in the 0-based lexicographic ordering of the given dequeue permutation*!!!

Just imagine: Had the question not brought in the nitty-gritties of "partial information", and had instead been *"Given a permutation of 1 to N, determine if it is a dequeue permutation; and if it is, output the index at which it would occur among all possible dequeue permutations of N"*, we would have been more or less done!

And this brings us to the 3rd question: **What information am I getting about the permutation when I say "The i-th position has value j"?** For example, if I said : "The second position had value 2", you would

know that the first position has to have value 1. If I said : "The second position had value 1", you would know that the first position had value N.

Lets look at it in terms of our $N-1$-bit numbers. "2nd position has value 2" means: by the 2nd choice, the number 2 was at one end (was exposed), and that that end was then chosen. Here, for large N, we know 2 was popped from the front, hence **the 2nd bit is 0**. But it was also exposed, which means, **the first (2-1) bits had exactly one 0-bit**. Since this is saying that the first 1 bit had exactly one 0, we get that the bit-sequence is 00??? which is as expected : "The first element is 1, and the second element is 2".

In general, when "value at i-th position is j", we know that (a) by the time we reached the i-th position, j was exposed from either right or left (b) it was then popped using that exposed side.

Here, it is that we need that it is in the beginning half of the permutation (i.e. i ¡= floor(N/2). This means that just depending on whether j ¿ N/2 or not, we can determine whether j was popped from the front or from the back.

Qualitatively, the proof goes as follows. Claim: if j ¡= N/2, then j was popped from the front. Because if it were popped from the back, that means that N, N-1, ..., [N/2] + 1 were all popped from the back too, and this is already floor(N/2) elements!! Hence j had to have been popped from the front. The exact same reasoning works with j ¿ N/2.

Quantitatively, the proof is as follows. If j was popped from the front, that means: in the first i bits, there were j 0s; and $i-j$ 1s. If j was popped from the back, that means: in the first i bits, there were $N-j+1$ 1s, and $i-(N-j+1)$ 0s. When $i \leq N/2$, the above two cannot both simultaneously hold (either $i-j$ goes negative, or $i-(N-j+1)$ goes negative).

In short, the information we get from the above is: **In the first $i-1$ bits, how many 0s and 1s are present**, as well as **what is the bit-value of the i-th bit**. With this information, we need to ask, **What is the minimum possible and maximum possible $N-1$ bit numbers satisfying constraints as given till now**.

Let us now try to understand how our minimum-possible (m) and maximum-possible (M) are affected by these changes. For this, let us take an example: $N = 100$, and our knowledge is got in the following (i, j) pairs: (3, 2), (11, 5), (5, 3), (9, 4), ...

Also, we shall keep our knowledge about number of 1s/0s till here, current bit-value in the form of a table (STL map for example). Initially, the table would look like this:

I	#0s seen	#1s seen	bit value
0	0	0	-

Currently, $m = (000000000000...00)_2$ $M = (111111111111...11)_2$ where I have specified the first 12 bits, and the others are 0s/1s depending on whether you're considering m or M.

Now, we see the pair (3, 2). This means, there are 2 0s and 1 1 upto bit3, with the 3rd bit = 0. Our table is modified now to

I	#0s seen	#1s seen	bit value
0	0	0	-
3	2	1	0

So here, the third bit of both m and M is fixed to 0. Also, in the first 2 bits, there's 1 0 and 1 1, so it is ideal for m to be of the form 01 while M is of the form 10 Thus, we now have $m = (010\ 000000000...00)_2$ $M = (100\ 111111111...11)_2$

Next, comes the pair (11, 5). This means that: In the first 11 bits, there are 5 0s 6 1s, with the 11th bit being a 0. Our table now becomes:

I	#0s seen	#1s seen	bit value
0	0	0	-
3	2	1	0
11	5	6	0

So here, after the first 3 bits of m and M are set to 010 and 100 respectively, we have to add 5 more 1s and 3 more 0s (one of which is at the 11th position). For m, it would be best arranged as ...00111110, whereas for M, it would be best arranged as ...11111000... Thus, we now have $m = (010\ 00111110\ 0...00)_2$ $M = (100\ 11111000\ 1...11)_2$

Next, we see (5, 3). This means that: In the first 5 bits, there are 3 0s and 2 1s, with the 5th bit being a 0. Our table now becomes

I	#0s seen	#1s seen	bit value
0	0	0	-
3	2	1	0
5	3	2	0
11	5	6	0

Since we have inserted the row 5-3-2-0 between the rows consisting of i=3 and i=11, we need to reorder things between the 4th and the 10th bit with this new information. Our information now becomes that, we have 4th-5th bit having one 0 and one 1, with a 0 at the 5th posn (hence it is 10 for both m and M) and in the 6th-11th bits, we have two 0s and four 1s, (with a 0 at 11th bit) where m would rather arrange it as 011110 and M would rather arrange it as 111100. Thus, we now have $m = (010\ 10\ 011110\ 0...0)_2$ $M = (100\ 10\ 111100\ 1...1)_2$

Finally, after seeing (9, 4), our table should become

I	#0s seen	#1s seen	bit value
0	0	0	-
3	2	1	0
5	3	2	0
9	4	5	0
11	5	6	0

And after making relevant changes to bits 6-11, you would get $m = (010\ 10\ 1110\ 10\ 0...0)_2$ $M = (100\ 10\ 1110\ 10\ 1...1)_2$

Also note that these tables can help detect for infeasibility also! Suppose instead of (9, 4) you instead had (9, 6). Your table would look like

I	#0s seen	#1s seen	bit value
0	0	0	-
3	2	1	0
5	3	2	0
9	6	3	0
11	5	6	0

showing you that there need to be "-1" 0s (and one of which is at bit 11!!) and 3 1s among the 10th and 11th bits!!!!

Finally, note that the values of m and M need only be updated within the range of bits from i_1 to i_2 where i_1 and i_2 are the elements just below and just above element i that is being inserted into the table. Further, m will always look like 0s followed by 1s, and M will always look like 1s followed by 0s.

Which brings us to our algorithm.

Algorithm:

Initialize table with 0-0-0–
(can be either an underlysing STL set plus an array, or just STL maps)
Also mark Feasible
for each (i, j) pair, do the following
if(!Feasible) output -1 and continue;
if (j > N/2)
bitval =1;
ones = N-j+1;
zeros = i - (N-j+1);
else
bitval = 0;
zeros = j;
ones = i-j;

Find the position where i is inserted: [i1, i2]
Consider 2 cases:
Case 1: i2 is the end of the set/map
reset bits of M to '0' from i1+1 to i
check for feasibility: are there non-negative #0s and #1s
between i1+1 and i-1 positions.
if infeasible, mark Feasible = false, output -1 and continue;
else set appropriate bits of m and M to 1, output and continue.
Case 2: i2 is not the end
reset bits of both m and M to '0' from (i1+1) to (i2-1)
check for feasibility: are there non-negative #0s and #1s
between i1+1 and i-1 positions and between i+1
and i2-1 positions.
if infeasible, mark Feasible = false, output -1 ad continue;
else set appropriate bits of m and M to 1, output and continue.

Finally, the above does not specify how to "set" and "reset" bits, but (for example) assuming you know m modulo 1E9+7 and which bits you want to reset (bits x to y), and if you had precomputed all powers of 2 modulo 1E9+7 beforehand, then you need to just "subtract" from m, the value pow2((N-1)-y)* (pow2(y-x)-1): since pow2(y-x)-1 gives you $y - x$ 1s, and multiplying it by pow2((N-1)-y) is equivalent to left-shifting these 1's to the appropriate position (i.e. y).

The above is modified for setting bits to 1 also.

Thus the overall time complexity is O(NlogN) using STL set/map.