CMIMC 2022
TCS Round

INSTRUCTIONS

1. Do not look at the test before the proctor starts the round.
2. This test consists of several problems, which require proofs, to be solved within a time frame of **90 minutes**. There are **300 points** total.
3. Answers should be written and clearly labeled on sheets of blank paper. Each numbered problem should be *on its own sheet*. If you have multiple pages, number them as well (e.g. 1/3, 2/3).
4. Write your team ID on the upper-right corner and the problem and page number of the problem whose solution you are writing on the upper-left corner on each page you submit. Papers missing these will not be graded. Problems with more than one submission will not be graded.
5. Write legibly. Illegible handwriting will not be graded.

Mysterious Cards

You are given a deck of cards, all face down, numbered from 1 to n, where $n = 2^{100} \approx 1.27 \cdot 10^{30}$. You are not allowed to look at the numbers on any of the cards, but, on each step, you may place the cards (face down) in a row from left to right, and you will be told the number of triples of cards whose values are ordered from least to greatest, going from left to right. For instance, if $n = 6$, and we laid down cards with values $4, 1, 5, 2, 3, 6$, from left to right, we would get a result of 6 triples (corresponding to $456, 156, 123, 126, 136, 236$).

Find an algorithm that, in at most k steps, will always allow you to determine the number written on every card. You do not have to be told that all triples in a placement are in order, but you need to have enough information to be certain that you know the number on each card.

Scoring

An algorithm that completes in **at most** k steps will be awarded:

- 1 pt for $k > 10^{100}$
- 10 pts for $k = 10^{100}$
- 30 pts for $k = 10^{61}$
- 50 pts for $k = 10^{33}$
- 80 pts for $k = 2.6 \cdot 10^{30}$
- 100 pts for $k = 1.3 \cdot 10^{30}$

You are allowed to prove multiple bounds, and will receive points for the best bound that is correctly proven. **Please state which bound you are trying to prove above any proof, or you may not be awarded points for that bound.**

Partial points may be awarded for progress towards the next bound, and partial points may be deducted for logical flaws or lack of rigor in proofs. To get full points, you must demonstrate that your algorithm always produces a correct result, and always runs in at most the stated number of moves.

Solutions

Solution for $k = 10^{61}$ (30 pts). We claim that we can determine in 4 steps which of two cards has a greater number. Let the cards that we will compare be C_1, C_2, and the rest of the cards are $C_3, C_4, \dots, C_n$. Suppose the cards in some arbitrary order $C_1, C_2, C_3, \dots, C_n$, which has t triples in order. Then place the cards in the order $C_2, C_1, C_3, \dots, C_n$, which has t' triples in order. If $t > t'$, then $C_1 < C_2$, because the first ordering counts triples that are not in the second ordering, while the second ordering counts only triples that are in the first ordering. Likewise, if $t < t'$, then $C_2 < C_1$.

Lastly, we must consider the case $t = t'$. This case will happen if there are no triples of the form (C_1, C_2, C_i) and no triples of the form (C_2, C_1, C_i). This happens if $C_1 = n$ or $C_2 = n$. In this case, we pull in a third card, C_3, which is guaranteed not to be the card with number n. We perform a similar comparison on C_1 and C_3, by placing the cards in the order $C_1, C_3, C_2, C_4, C_5, \dots, C_n$, and in the order $C_3, C_1, C_2, C_4, C_5, \dots, C_n$. Suppose that the first ordering has u triples in order, and the second ordering has u' triples in order. If $u \neq u'$, then neither C_1 nor C_3 have the number n, so $C_2 = n$.

If $u = u'$, then C_1 must have the number n, since we found earlier that either C_1 or C_2 have the number n. If $C_1 = n$, then $C_2 < C_1$, and if $C_2 = n$, then $C_1 < C_2$. In all cases, we can determine whether $C_1 < C_2$ or $C_2 < C_1$ using at most 4 comparisons.

By comparing each pair of cards in the deck, we can determine which number goes on each card. There are $\binom{n}{2} = \frac{n^2-n}{2}$ pairs of cards, so we make $4 \cdot \frac{n^2-n}{2} = 2n^2 - 2n < 2 \cdot (1.3 \cdot 10^{30})^2 < 10^{61}$ comparisons. Therefore this algorithm always determines the number on each card in at most 10^{61} comparisons.

Solution for $k = 10^{33}$ (50 pts). Now that we know how to compare any two cards in 2 steps, it is not hard to come up with a more efficient sorting algorithm. Let's say we at all times maintain the largest possible list of elements that we currently know to be in order (initially this list will have length 1). Then, we pick a card not yet in this list, and use comparisons to perform a binary search on this list to find where the card should go (this sorting algorithm is known as Insertion Sort). More specifically, we first compare our new card with the middle card in our list. If our card is larger, we only have to consider the sub-list of cards to the right of the middle card, and if it is smaller, we consider the sub-list of cards to the left of the middle card, and then repeat this process with the middle card of our new sub-list.

Each binary search will take at most $\log_2(n) = 100$ comparisons, and we have to process $n-1$ new cards before we know the proper ordering of all n cards, hence the total number of queries will end up being roughly $2 \cdot 100 \cdot (n-1) \approx 2.56 \cdot 10^{32} < 10^{33}$ (there will be a small number of extra steps involved in finding the card labelled n and then moving it to the end of the list, but this won't affect our bound here).

Solution for $k = 2.6 \cdot 10^{30}$ (80 pts). We modify our comparison method so that after 2 steps we can determine the number written on one of the cards. In particular, we can determine the higher of the numbers written on the two cards, and (usually) determine which card has that number. Recall that the ordering $C_1, C_2, C_3, \dots, C_n$ had t triples in order, and the ordering $C_2, C_1, C_3, \dots, C_n$ had t' triples in order.

We look at the difference of t and t'. If $t > t'$, then we count $t - t'$ triples that are counted in the first ordering but not the second ordering. This is equal to the number of triples of the form (C_1, C_2, C_i), with $C_2 < C_i$, which is equal to the number of C_i with $C_2 < C_i$. This makes $C_2 = n - (t - t')$. Similarly, if $t < t'$, then $C_1 > C_2$ and $C_1 = n - (t' - t)$. If $t = t'$, then as before we use 2 more comparisons to determine that either $C_1 < C_2 = n$ or $C_2 < C_1 = n$.

To determine the number on each card, note that after each comparison, we know the number on one of the cards. Then, as long as we have at least two unknown cards, we can compare those cards to learn the value of one of them. We stop once there is only one unknown card left, at which point we can determine the number on that card by process of elimination. This requires $n - 1$ comparisons, with all but one comparison requiring 2 steps, and one comparison requiring 4 steps. Our algorithm then terminates in $2(n - 2) + 4 = 2n < 2.6 \cdot 10^{30}$ steps.

Solution for $k = 1.3 \cdot 10^{30}$ (100 pts). Our solution for $k = 2n$ requires moving every card to the front each time, which seems somewhat inefficient, since at the end of the day we just want to be swapping pairs of adjacent cards, so let's see if we can avoid it. Recall that our swapping comparison from that solution gave us the value of the larger of the two cards (and, except when that value was n, told us which one it corresponded to). This means that one of our two queries looked something like $[K], X, C_3, C_4, \cdots$, where $[K]$ is a list of all cards with known values (currently one of C_1 or C_2), and $f(X)$ is strictly smaller than every value in $[K]$, since we only ever learn the value of the larger of the two cards we swapped.

Let's now query on $[K], C_3, X, C_4, \cdots$. Then, the difference $t - t'$ between this query and the other will again tell us how many valid triples contain C_3, X in some order. Furthermore, since everything to the left of our two cards is larger than X, no triple can contain one of those values and also contain X, so $t - t'$ will just tell us the number of cards to the right of C_3, X that are larger than both C_3 and X, along with which of C_3 and X is larger. This is enough to uniquely determine the larger of C_3, X, call it Y, since $f(Y)$ must just be the $(|t - t'| + 1)$th largest number in our list of not yet known values (every value not in $[K]$).

Now, set our new card X to once again be the smaller of C_3 and our old X, and consider the ordering $[K], Y, X, C_4, \cdots$ that we have already queried. Since we know the value of Y, we can merge Y into $[K]$, so once again we have obtained an ordering of the form $[K], X, C_i, \cdots$, where $f(X)$ is less than every value in $[K]$. We can now repeat this process indefinitely, each time making 1 new query, and ending with some ordering $[K], X, \cdots$ that we have queried before, so eventually $[K]$ will contain $n - 1$ values, and $f(X)$ will just be the remaining value (which has to be 1, since it is smaller than every value in $[K]$). After 2 queries, $[K]$ has length 1, so after n queries, $[K]$ will be of length $n - 1$, and we will be done.

Note also that if we run into the value n at some point midway along our process, it is not actually an issue, since we know that our current value of $f(X)$ will always be less than n, since it is less than something in $[K]$, which means the card not equal to X is the one with value n. Therefore, the only edge case we have to consider is when the very first two queries give equal results, since then we don't actually know which card to pick as X, and we can't begin our process. In this case, just move both C_1 and C_2 to the end of the list, and try again with C_3, C_4 (which is guaranteed to work). This costs two extra moves, so in total this process takes at most $k = n + 2 < 1.3 \cdot 10^{30}$ steps.

A Dungeon Journey

You have been placed in a dungeon with $n = 2^{100} \approx 1.27 \cdot 10^{30}$ rooms, numbered from 1 to n, each of which is connected to some other rooms by a two-way hallway. You are in room 1, but you don't know what the dungeon looks like. Even worse, going through a hallway (almost) completely wipes your memory! Your task is to gather a complete map of every room and hallway. (A complete map is a list of every pair of rooms connected by a hallway)

Each room has a large blackboard where you can write down anything you want. The next time you visit the room, you'll be able to see what you wrote, despite your memory being wiped. When walking between rooms, you may carry k bits of information (or alternatively, an integer from 0 to $2^k - 1$, inclusive), but all other memory will be forgotten.

When in your current room, you can see:

- The number of hallways leading out of the room (but the hallways are indistinguishable, and you cannot mark them)
- Any information written on the blackboard from previous visits.
- The k bits of information you chose in the previous room.
- The room number (an integer between 1 and n, where room 1 is your starting room)

You must then choose:

- What to write on the blackboard. (there is no limit to how much you can write)
- k bits of information to bring to the next room.

Since the hallways leaving your room are indistinguishable, you pick one to walk through at random! You may assume:

1. Every room is reachable from every other room.
2. Despite choosing randomly, you are lucky enough that you traverse each hallway at least once (in both directions) every n^{n^n} moves.
3. No room has a hallway to itself, or multiple hallways to the same room.
4. Every hallway is two-way; if it goes from A to B then it goes from B to A.

Design an algorithm to eventually create a complete map of the dungeon, **that you know to be correct**, in some room, in a finite number of steps. You will also remember this algorithm when moving between rooms.

Scoring

An algorithm that can map the dungeon carrying at most k bits of information will be awarded:

- 1 pt for $k > 10^{63}$
- 5 pts for $k = 10^{63}$

- 10 pts for $k = 10^{60}$
- 40 pts for $k = 350$
- 50 pts for $k = 250$
- 60 pts for $k = 150$
- $100 - 2(k-1)$ pts for $1 \leq k \leq 20$

You are allowed to prove multiple bounds, and will receive points for the best bound that is correctly proven. **Please state which bound you are trying to prove above any proof, or you may not be awarded points for that bound.**

Partial points may be awarded for progress towards the next bound, and partial points may be deducted for logical flaws or lack of rigor in proofs. To get full points, you must demonstrate that your algorithm always produces a correct result.

Solutions

Solution for $k = n^2 + n$ (5 pts). A simple approach is to hold the whole map in your memory. There are n^2 pairs of rooms, so n^2 bits should suffice to hold each entry of the table, which is just a yes/no answer.

Each time you visit a room, you will just note the hallway you passed through. You keep track of the last room you visited using the diagonal of the table, which should be 0 by Assumption 3.

By Assumptions 1 and 2, you will eventually traverse every hallway. Once you are in a room and see that you have recorded every hallway, you can note this using an extra bit. This adds n bits.

Solution for $k = n^2/2$ (10 pts). A slightly improvement on the $k = n^2 + n$ solution is to take note of Assumption 4. This reduces the number of valid pairs to $n(n-1)/2$. You should add $2\log(n)$ bits to this. One $\log(n)$ is to keep track of where you just came from; the other $\log(n)$ is a counter of how many rooms have had all their hallways visited. (You can record information in each room to make sure you don't increment it twice for the same room.) Keep in mind that you can store a number from 1 to n inclusive in $\log(n)$ bits; log will always be the base-2 logarithm.

As for large n, $2\log(n) < n/2$, we have that $n^2/2$ bits is possible.

Solution for $k = 3\log(n) + 2$ (40 pts). We can improve on our solution by not carrying the entire map in our head at once.

In this solution, two things are carried from room to room. The first is the room most recently visited.

The second is a message. There are two kinds of messages: "Room x connects to Room y", of which there are $n(n-1)/2$ possible; and "Room x has d connecting Rooms", of which there are $n(n-1)$ possible (n rooms, connecting rooms can range from 1 to $n-1$ inclusive by Assumptions 1, 3, and 4.) So there are $1.5n^2$ messages.

So the total information sent is $\log(1.5n^3) < \log(2n^3) < 3\log(n) + 1$ bits.

The message is chosen randomly from the set of all message which a given room knows to be true. (Every room knows its own number of connecting rooms, so there is always 1 possible message.)

By the laws of probability and Assumption 2, with probability 1, all of the information about all of the rooms will be collected and disseminated to every room eventually. The information which is sent is sufficient to make a map of all rooms.

Solution for $k = 2\log(n)$ (50 pts). We adapt the $3\log(n) + 1$ solution by noting that we don't need to send all of the information at once; we can send where we came from **OR** an edge **OR** a degree. So there are $n + n(n-1)/2 + n(n-1) < 2n^2$ messages, which takes $2\log(n) + 1$ bits.

By the same argument, all information will eventually be collected and disseminated to all rooms.

Solution for $k = 3$ (96 pts). The key insight for constant-time solutions is to note that you can transmit information to all nodes, one bit at a time.

This solution uses 8 states: `INF-0`, `INF-1`, `ASK`, `KEY`, `ADJ`, `ADV`, `DONE`, and `NIL`.

Room 1 is the "control room", and directs everything.

The algorithm uses the concept of "signals", which are messages that are passed along unchanged until they reach their intended destination.

```
FOR EACH i from 2 through n:
    WHILE there are still adjacent rooms of i to be found:
        SEND the ASK signal from the Control Room.
        All rooms other than room $i$ will ignore it and send it on.
        When room $i$ receives it, it will send KEY.

        The vertex that receives KEY will mark itself as the "adjacent room"
        and send NIL, unless it has been adjacent room before for this vertex.
        The NIL will propagate until it returns to the Control Room.
        Else, it transforms the signal back into ASK.

        Once an adjacent room has been identified,
        (as indicated by the receipt of NIL at the Control Room)
        the Control Room will send out ADJ.

        This will cause the adjacent room to send out the bits of its ID number.
        The bits will be sent as INF signals to the Control Room.
        When all bits have been exhausted, the adjacent room notes this
        and ceases to be the adjacent room.

        When room i receives ADJ for the first time after each ASK, it will return NIL,
        and increment the discovered vertices counter by 1.

        When all bits of the adjacent room's ID have been received,
        and room i has returned a NIL, the loop continues from ASK.

    WHEN room i has noted all of its adjacent rooms have been found,
    it will send out an DONE signal in response to the next ASK.

    This will indicate that the Control Room should advance to the next room.

    The Control Room will send out the signal ADV.
    Each room that receives it for the first time will note this, and send NIL.
    When all n-1 other rooms have noted the ADV, the loop continues.
```

This algorithm works cleanly because each signal has a unique meaning: the `INF` signals carry information to the Control Room; the `ASK` and `ADJ` signals represent two kinds of "questions" asked to the rooms; the `DONE` and `ADV` are special markers for the Control and every other room, respectively to advance to the next vertex; `KEY` is the special marker of adjacency; `NIL` is used to indicate that a signal has been received (in the `ADV` phase) or that a transmission is done (in the `ADJ` phase).

The algorithm is guaranteed to find an answer by Assumption 2, which ensures that a signal will eventually reach its destination.

Long Knight's Tour

Consider a n by n chessboard with squares labelled $(1, 1)$ through (n, n). On this chessboard lies a long knight. The long knight can move from square (x, y) to (x', y') if one of the two following conditions hold:

- $|x - x'| = 3$ and $|y - y'| = 1$
- $|x - x'| = 1$ and $|y - y'| = 3$

In essence, it is a normal chess knight, but longer.

A 'tour' of the chessboard is a sequence of squares $S_1, S_2, S_3, \ldots S_n$ such that for all $1 \le i \le n - 1$ the move from S_i to S_{i+1} is a valid move for a long knight. Such a tour is considered 'complete' if and only if the tour visits each row and column of the chessboard **exactly** once.

			4	
1				
		5		3
	2			

The above picture is an example of a tour on a 5 by 5 grid, but the tour is not 'complete' as Row 4 is visited twice, and Row 3 is visited zero times.

For each positive integer n, determine whether it is possible for a complete tour of an n by n chessboard to exist.

Scoring

- $2k$ pts for resolving the problem for $1 \le k \le 15$ values of n
- 35 pts for resolving the problem for infinitely many values of n
- 65 pts for resolving the problem for all but finitely many values of n
- $100 - 2k$ pts for resolving the problem for all but $1 \le k \le 15$ values of n
- 100 pts for resolving the problem for all $n \ge 1$

Please state which bound you are trying to prove above any proof, or you may not be awarded points for that bound.

Partial points may be awarded for progress towards the next bound, and partial points may be deducted for logical flaws or lack of rigor in proofs. To get full points, you must rigorously show proofs for n for which the problem is impossible, and have clear and reasonable ways to show a long knight's tour exists for the n for which it is possible (including but not limited to explicit constructions).

Solutions

Solution for all n (100 pts). First we show there are no complete tours for $n = 2, 3, 4$.
For $n = 2, 3$ there are no valid moves as no two coordinates differ by 3. Thus no tours are possible, let alone any complete tours.

For $n = 4$, we know S_2, S_3 must be adjacent (by a long knight's move) to at least two other squares in the 4 by 4 board. The only such squares are the corners, implying that S_2, S_3 must both be corners. However, no two corners are adjacent. Thus no complete long knights tours exist for $n = 4$.

Now for constructions. For $n = 1$ we take the trivial construction $S_1 = (1, 1)$.
For $n = 5, 6$ we provide the following constructions.

				3
	4			
1				
			2	
		5		

					3
		4			
	1				
				2	
			5		
6					

To reduce the search for $n = 5$, we can note that if we color the chessboard in with normal black and white square, all of the long knight's moves are on the same color. For there to be three visited squares in rows 1, 3, and 5 we must use the three columns 1, 3, and 5. This leaves the only place for squares on the 2nd and 4th rows to be on the opposite squares in the 2nd and 4th columns.

We can also note that the construction for $n = 6$ has the last square in the tour at the bottom left. We call any complete tour with this property 'nice'.

We also provide the following 'extenders':

									n
						n+1			
	n+6								
				n+5					
							n+2		
		n+7							
					n+4				
								n+3	
			n+8						
n+9									

								n
					n+5			
		n+6						
							n+1	
				n+4				
	n+7							
						n+2		
			n+3					
n+8								

Let's call these the 9- and 8-extenders respectively. We can use these extenders to turn to a nice complete tour of size n into a nice complete tour of size $n+9$ or $n+8$.

One final operation we can do is to take a nice complete tour of size n and 'cap' it with a $180°$ rotation of the $n = 6$ complete tour to get a $n+5$-size complete tour

We can now construct the following complete tours:

- $n = 7$ (Take $n+1$ to $n+7$ in the 8-extender)
- $n = 8$ (Take $n+1$ to $n+8$ in the 8-extender) - **nice**
- $n = 9$ (Take n to $n+8$ in the 8-extender) - **nice**
- $n = 10$ (Take n to $n+9$ in the 9-extender) - **nice**
- $n = 11$ (Cap the $n = 6$ nice complete tour)

We can also construct the following infinite families

- $n = 6+8k, k \geq 1$ (Add k copies of the 8-extender to to the 6 nice complete tour) - **nice**
- $n = 5+8k, k \geq 1$ (Chop off the last thing from the family above)
- $n = 7+8k, k \geq 1$ (Add $k-1$ copies of the 8-ext and 1 copy of the 9-ext to to the 6-tour) - **nice**
- $n = 8+8k, k \geq 1$ (Add k copies of the 8-extender to to the 8 nice complete tour) - **nice**
- $n = 9+8k, k \geq 1$ (Add k copies of the 8-extender to to the 9 nice complete tour) - **nice**
- $n = 10+8k, k \geq 1$ (Add k copies of the 8-extender to to the 10 nice complete tour) - **nice**
- $n = 11+8k, k \geq 1$ (Cap the $n = 6+8k$ nice family)
- $n = 12+8k, k \geq 1$ (Cap the $n = 7+8k$ nice family)

This covers every positive integer n except for $n = 12$. The following construction works (and is also a cap!):

									8		
						9					
			12								
										7	
					10						
		11									
											6
								5			
	2										
				3							
							4				
1											

Solution for infinitely many n (35 pts). If we find any extender we can do this. For example, if we have the 8-extender, conjoining k copies of this extender will give us a solution for $n = 1 + 8k$, which resolves infinitely many values as k ranges over the positive integers.

Solution for cofinitely many n (65 pts). One way to do this is by using the Chicken McNugget Theorem.

Consider the 8- and 9-extenders that were found before. By starting with the $n = 1$ tour (which is nice) and adding a copies of the 8-extender and b copies of the 9-extender, we can generate a complete tour of size $n = 1 + 8a + 9b$. By the Chicken McNugget Theorem, this implies that we can generate a complete tour for any grid of size $n \geq 1 + (8 \cdot 9 - 8 - 9) = 56$, which is all but finitely many values of n.

Another way to do this is if one just does a similar case bash as to in the full solution, but just uses larger grids as base constructions. If would guess that most solutions that do this would mostly be just manually showing the Chicken McNugget theorem, but using more detail to squeeze out a higher score.