Alternating Current

Subtask 1

There are only 2^M wirings, so when $M \leq 15$ we can simply try all of them and check if any wiring satisfies the constraints. To check a wiring, we can go through all wires in each direction separately and for each wire mark the segments that are covered by that wire. This yields an algorithm which runs in time $\mathcal{O}(2^M MN)$.

Subtask 4

No wire passes through the separator between segments 1 and N, so there is a natural order on wires, namely by starting point. We'll sort the wires according to this measure, and go through them one by one and assign their directions greedily.

When we see a wire, the choice is between assigning it clockwise and counter-clockwise direction. If the maximum ending point assigned to clockwise wires is greater than the maximum ending point assigned to counter-clockwise wires, then we are strictly better off assigning it a counter-clockwise direction, since clockwise and counter-clockwise directions are symmetrical. (Note that this symmetry comes from our common starting point.) Otherwise, we may assign it a clockwise wire. If we get any gaps, the answer is impossible, otherwise this greedy solution will work.

Subtask 5

The main observation is that if one wire w covers only a subset of the segments covered by another wire w', then we can without loss of generality assign the direction of w to be opposite the direction of w', because assigning the direction of w to be the same as the direction of w' would accomplish nothing. Let us therefore assign *parents* to as many of the wires as possible, in such a way that:

- If w' is the parent of w then w covers a subset of the segments covered by w'.
- A wire which is itself a parent has no parent.

By doing this we have partitioned the wires into two disjoint sets: The set of parents and the set of nodes that have a parent. From now on we will focus on the set of parents, which we denote by $\mathcal{W}$.

No wire w in $\mathcal{W}$ covers only a subset of the segments covered by any other wire w' in $\mathcal{W}$, because if that were the case we could have let w' be the parent of w. Instead, the wires in $\mathcal{W}$ partially overlap or are completely disjoint.

Let us now sort the wires in $\mathcal{W}$ by increasing start segment, and let $w_1, \ldots, w_k$ be the sorted list of wires in $\mathcal{W}$. Let us now prove the following lemma:

Lemma 1. *Suppose that k is even and that there is a valid wiring. Then we get a valid wiring by assigning the wires $w_1, w_3, w_5, \ldots, w_{k-1}$ in one direction and the wires $w_2, w_4, w_6, \ldots, w_k$ in the other direction.*

Proof. Suppose, for the sake of contradiction, that there is some segment s which is not covered in both directions when the wires $w_1, w_3, w_5, \ldots, w_{k-1}$ are assigned in one direction and the wires $w_2, w_4, w_6, \ldots, w_k$ are assigned in the other. Since there exists a valid wiring, all segments must be covered by at least one wire in $\mathcal{W}$, so let us assume that w_i covers s. If w_{i-1} or w_{i+1} covered s then s would be covered by wires in both directions, and therefore w_i must be the only wire in $\mathcal{W}$ that covers s. This means that all other wires (not in $\mathcal{W}$) covering s must be children of w_i, but then we would assign their directions to be opposite the direction of w_i, so s is again covered by wires in both directions. □

Lemma 1 implies that in the case when k is even, we can simply use an alternating assignment to the wires $w_1, \ldots, w_k$ and then check whether that assignment is a valid one.

The case when k is odd is slightly harder. For this case we use the following lemma:

Lemma 2. *Suppose that k is odd and that there is a valid wiring. Then there is an i such that we get a valid wiring if we assign the wires $w_i, w_{i+2}, w_{i+4}, \ldots, w_{i-1}$ in one direction and the wires $w_{i+1}, w_{i+3}, w_{i+5}, \ldots, w_{i-2}$ in the other direction, where all indices are taken modulo k.*

Proof. Consider a valid wiring. Since k is odd there exists an i such that the valid wiring assigns the same direction to w_{i-1} and w_i. We claim that we get a valid wiring if we assign the wires $w_i, w_{i+2}, w_{i+4}, \ldots, w_{i-1}$ in one direction and the wires $w_{i+1}, w_{i+3}, w_{i+5}, \ldots, w_{i-2}$ in the other direction. Suppose for the sake of contradiction that there is some segment s which is not covered in both directions by this wiring. Since there exists a valid wiring, s must be covered by at least one wire w_j in $\mathcal{W}$.

Suppose that s is covered by w_{i-1} or w_i. If s is also covered by w_{i-2} or w_{i+1} then s is covered by wires in both directions, so we can assume that s is not covered by w_{i-2} or w_{i+1}. This means that all other wires containing s (not in $\mathcal{W}$) must be children of either w_{i-1} or w_i, but then we would assign their directions to be opposite the directions we assign to w_{i-1} and w_i, so s is again covered by wires in both directions.

Suppose that s is covered by some wire $w_j \in \mathcal{W}$, for $j \notin \{i-1, i\}$. If w_{j-1} or w_{j+1} also covered s then s would be covered by wires in both directions, so let's assume that w_j is the only wire in $\mathcal{W}$ that covers s. Then all other wires covering s must be children of w_j, which means that they are assigned directions opposite to the direction assigned to w_j, so s is covered by wires in both directions. □

For subtask 3 it is sufficient to iterate through all possible values of i and check the resulting wiring for each. For subtask 5 we need to be more clever. A

wiring that assigns the wires $w_i, w_{i+2}, \ldots, w_{i-1}$ in one direction and the wires $w_{i+1}, w_{i+3}, \ldots, w_{i-2}$ in the other direction is valid if and only if

1. All segments are covered by at least one wire.
2. For all $j \notin \{i-1, i\}$, the wires w_{j-1}, w_j, w_{j+1} and all their children together cover, in both directions, all the segments covered by w_j.
3. The wires $w_{i-1}, w_i, w_{i+1}, w_{i+2}$ and all their children together cover, in both directions, all the segments covered by w_i and w_{i+1}.

The requirements 1 and 2 above are almost independent of the choice of i (except that we require $j \notin \{i-1, i\}$), so when we change the value of i we only need to check if requirement 3 is satisfied, which can be done in amortized constant time over all choices of i.

The time complexity of the algorithm is dominated by the process of finding parents of nodes and sorting the wires in $\mathcal{W}$, which takes time $\mathcal{O}(M \log M)$.

Genetics

This problem had a simple cubic solution, lots of potential for optimization, and a nice probabilistic quadratic solution. We'll describe the latter.

Let's start by solving the problem for the binary alphabet case. In this case, we can make the math slightly nicer: instead of having the letters `A` and `C`, we use the numbers 1 and -1. Computing the difference between two DNA strings a and b then becomes isomorphic to taking a dot product between two rows of a matrix A, i.e., a sum of $A_{i,j} \cdot A_{i',j}$ for $j = 1 \ldots M$ (up to some constant factor and linear rescaling – instead of wanting sums to equal K, we want them to equal $K' = M - 2K$).

Now, what we want to check is that the dot products are K' for all rows $b = A_{i' \neq i}$. Rather doing this individually for all b, we will check the sum against every other row at once. There are a bunch of different approaches for this, but one nice way is to pick random values w_i for each row, and then check that the dot product against the combined sum $\sum_{i'} w_{i'} A_{i'}$ equals $(\sum_{i'} w_{i'} - w_i) \cdot K'$ (where i is the row we're checking).

In less abstract mathematical terms, and generalizing to larger alphabets, we pick random w_i for each row, and then for each column j and letter $c \in \{A, C, G, T\}$, compute D_c as the sum of w_i for each row which has the letter c in the j'th column. Then, we can check a row a_i against every other by computing the sum $\sum_j \sum_{d \neq A_{i,j}} D_d$. If this row is the answer, this equals K times the sum of the other rows' w's, since it differs in exactly K positions from each other row $A_{i'}$, and the sum thus includes $w_{i'}$ K times.

If the row isn't the answer, it highly likely does not equal that. Changing any w of a row that differed in something else than K positions would results in a changed sum, and if we say do all the arithmetic modulo 2^64 we have a probability of accidentally passing the test on the order of 2^{-50}. Hence, the solution passes with very close to 100% probability.

Interesting notes:

- Test data generation for this task was pretty tricky. For the naive solution not to pass, we want *almost* all distances between rows to be K, but all but one row should also have some other row with distance not K. The only matrices that the authors are aware of where distances are all equal are identity matrices (with $N = M, K = 2$), constant matrices ($K = 0$), and generalized *Hadamard matrices* ($N = M$, $K = N(1 - 1/A)$), and combinations of the three. Here A denotes the alphabet size. For a binary alphabet, Hadamard matrices are defined to be matrices of $\{1, 0\}$ with $N = M$ such that all rows differ in exactly $N/2$ positions. They are well studied, and a simple construction of them is a recursive one: start with the matrix

  $$H = \begin{bmatrix} 1 \end{bmatrix},$$

  and repeatedly replace H by

  $$\begin{bmatrix} H & H \\ H & H \oplus 1 \end{bmatrix}$$

  where $H \oplus 1$ means H with all entries of H XORed by 1. This results in a Hadamard matrix of any size which is a power of 2.

  For alphabets of size 4 we can do something more complicated, with instead replacing H by

  $$\begin{bmatrix} H & H & H & H \\ H & H \oplus 1 & H \oplus 2 & H \oplus 3 \\ H & H \oplus 2 & H \oplus 3 & H \oplus 1 \\ H & H \oplus 3 & H \oplus 1 & H \oplus 2 \end{bmatrix}$$

  Proof that this works is left as an exercise for the reader (the construction is derived from the multiplication table for $GF(4)$, for the mathematically inclined).

  Given matrices with all pairwise distances K, we can perturb the matrix in various way to make the answer unique, e.g. duplicating rows or changing bits of the matrix.

  These complex constructions partly explain the constraints section – it is difficult to construct Hadamard matrices for sizes that are not powers of A for alphabet size A.

- There is also a fun sub-cubic solution: with the formulation that values are in $\{-1, 1\}$, we can think of the problem simply as asking for a matrix product $A \cdot A^T$, from where we can check which values are K. Matrix multiplication can in theory be computed in $O(n^{2.373})$ time, although in

practice the algorithms that do this are very non-trivial to implement and have too high constant factors.

Paths

Subtask 1

To get the points for Subtask 1, it's sufficient to write a bruteforce solution that naively counts all possible valid paths. (See full solution below, but remove the parts regarding memoization and dynamic programming to make it exponentially slow.)

Subtask 2

Since the number of colors was at most 3, the length of the path was also at most 3; and thus either 2 or 3. Paths of length 2 are simple to count: they are just the number of edges between nodes with different colors. Paths of length 3 require a bit more thought. One way we can handle them is by looping over which node is in the middle of the path, and which colors the nodes at the start and end of the path have. Then the number of paths that have this node in the middle is the product of the number of neighbors of the first color and the number of neighbors of the second color.

Subtask 3

This is a natural extension to subtask 2: instead of looping over which node is in the middle of the path, we loop over which *edge* is in the middle. The rest works exactly the same.

Subtask 4

Assume there's a function f that gives the number of valid paths starting in a certain node. Then the answer to the problem is the sum of f over all nodes. The trick is to calculate f efficiently, as the input to the problem is quite large.

Let f take two parameters c (the current node) and C (a bitset indicating the colors we have used so far, initially 0). Since the number of colors in a path is at most 5, the number of possible such bitsets is $2^5 = 32$. This means that the number of possible combinations of parameters to $f(c, C)$ is small enough to make a lookup table: we can use dynamic programming.

Memoizing on parameters c and C, we implement $f(c, C)$ by summing $f(c', C')$ for all neighbours c' of c, and where C' is the same bitset as C but with the color bit of c marked. We make sure to not take any paths where we reuse a color, and we make sure to not calculate answers for states that we have previously calculated. Given this, the answer to the problem is $\sum f(i, 0)$ (make sure to not count paths of length 1!). Also remember to use 64-bit integer types.

This yields a time complexity of $\mathcal{O}(2^K N)$.

Interesting note: the problem was essentially about counting paths of length at most k, given that all nodes need to have different colors. This is a much simpler than counting *all* paths of length at most k! The naive time complexity of the latter is on the order of n^k rather than $2^k n$. This can be used as a general algorithmic technique to speed up various problems: pick colors randomly a bunch of times (say, $O((\log n)^k)$ times), then use an algorithm that restricts nodes to have different colors.