Code Jam 2020 - Round 2

Analysis: Incremental House of Pancakes

Test Set 1

Test Set 1 is small enough that we can simulate the process. Since each step removes at least one pancake from a stack, this takes at most **L** + **R** operations. One thing we can notice is that because the i-th step removes i pancakes, the first i steps together remove i × (i + 1) / 2 pancakes. This means that the number of steps is actually bounded by 2 × sqrt(**L** + **R**), which means the simulation algorithm is also O(sqrt(**L** + **R**)). It would work for limits that are much larger than the ones in this test set!

Test Set 2

Unfortunately, O(sqrt(**L** + **R**)) is still too slow for the 10^{18} limits in Test Set 2, especially with 1000 cases to go through!

We can simulate multiple steps at a time by noticing that there are two distinct phases of the process. The first phase uses a single stack: the one that starts out with more pancakes. The second phase begins when that stack has a number of pancakes remaining that is less than or equal to the number in the other stack. Notice that if the left stack is the one we serve from in phase 1, it is possible that it is also the first one to be used in phase 2. It is also possible that no customer is served in either phase.

We may serve a lot of customers in phase 1 depending on the difference in size between the stacks at the beginning. If we serve i pancakes from one stack, we remove i × (i + 1) / 2 pancakes from it, so we can efficiently calculate how many customers we can serve in phase 1 by finding the largest i_1 such that $i_1 \times (i_1 + 1) / 2$ is less than or equal to the difference in number of pancakes between the two stacks at opening time. We can calculate that either by solving a quadratic equation and rounding carefully, or by using binary search.

The second phase is where the magic happens. Let us say that when serving customer i, stack X is used and Y is not, but then we serve customer i + 1 from stack Y. Therefore, stack X lost i pancakes and stack Y lost i + 1 pancakes. Since X was used instead of Y for customer i, X must have had no fewer pancakes than Y had at that time. Since X lost fewer pancakes than Y, X must have had more pancakes than Y after we served customers i and i + 1. This means if we ever use two different stacks in the order (X, Y), we must use X next. And, using the same reasoning with the roles reversed, we have now have used (Y, X) most recently, so we will use Y next, and so on. So, once we have used both piles, we always go on alternating between them.

We can use this observation to efficiently determine what happens in phase 2. After updating the original totals by subtracting the pancakes served in phase 1, we know which stack is used first in phase 2. The first stack will be used to serve customers $i_1 + 1$, $i_1 + 3$, $i_1 + 5$, ... which means that if it is used for c_1 customers, a total of $(i_1 \times c_1) + (c_1)^2$ pancakes are served from it. At this point, we know the value of i_1, so once again, we can calculate c_1 by solving a quadratic equation or binary search. The other stack is similar, since it will be used to serve customers $i_1 + 2$, $i_1 + 4$, $i_1 + 6$, ... so if it is used for c_2 customers, a total of $((i_1 + 1) \times c_2) + (c_2)^2$ pancakes will be served from it. So the final number of customers served is $i_1 + c_1 + c_2$. The total numbers of pancakes served from each stack come from the quantities in phase 2, with the quantity from phase 1 added to whichever stack was first.

If we use binary searches, each phase requires O(log(**L** + **R**)) time. If we directly solve the quadratic equations, each phase is actually constant time. Either is fast enough for the limits of this problem.

Notice that solving quadratic equations may be harder than usual, since typically that involves computing some square roots. While most languages provide a way to do that, they do it with double precision floating point, which does not have enough precision for this problem and can lead to off-by-one errors. We should either compute square roots directly on integers (by binary searching for the answer, for example) or use the built-in function, and then check the returned value and other values in its vicinity to find the correct rounded result.

Code Jam 2020 - Round 2

Analysis: Security Update

Let R_i be the number of computers that receive the update before computer i, and T_i be the time between computer 1 and computer i receiving the update. For each i, the input gives us exactly one of these numbers. We can set $R_1 = T_1 = 0$ for convenience.

A simplified problem

Let us assume for now that we have all the T_is. If computers i and j share a direct connection and $T_i=T_j$, then any path that gets to computer i in time T_i does not go through computer j, and vice versa, because all latencies are positive. Therefore, we can assign any positive latency to all those connections. If computer i has a given T_i, it means that any connection coming from computer j with $T_j < T_i$ needs to have a latency of at least T_i-T_j, or otherwise the update could get to computer i in less than T_i time by getting to computer j in T_j time and then using that connection. In addition, for at least one j, the latency of the connection between i and j has to be exactly T_i-T_j, or otherwise the time for the update to reach computer i would be larger than T_i. One simple way to solve this problem is to make all connections between computers with different T values have a latency of exactly $|T_i-T_j|$; this takes O(**D**) time.

Notice that the algorithm above finds a valid assignment for any set of T_is. To solve the actual test sets, we are left with the problem: given some T_is and some other R_is, assign all of the non-given values in a way such that sorting the computers by T_i leaves them sorted by R_i, and vice versa. In particular, computers with equal T values should have equal R values, and vice versa.

Test Set 1

In this test set, we can solve the subproblem from the previous section by setting $T_i := R_i$.

Test Set 2

For Test Set 2, we again focus on solving the subproblem. We do that by first ordering the computers by what is going to be their final T_i value (or equivalently, by their final R_i value). We can partition the set of computers other than the source computer into two: those for which we know R_i (part R) and those for which we know T_i (part T). We can sort each of those in non-decreasing order of the known value. We now have 2 sets that are in the right relative order, and need to merge them as in the last step of Merge Sort. We assign the source computer first. Then we iterate through the remaining **C**-1 slots in order. Suppose we have already merged N computers, and let computer k be the last one of those. Let i and j be the first computers remaining in parts R and T, respectively. If $R_i \leq N$, we take computer i next and assign $T_i := T_k$ if $R_i = R_k$, and $T_i := T_k+1$ otherwise. If $R_i > N$, we take computer j next and assign $R_j := R_k$ if $T_j = T_k$ and $R_j := N$ otherwise.

We can prove that if the original set of values is consistent with at least one latency assignment (which the statement guarantees), this generates a valid order and assignment of missing values, and moreover, it generates one in which the T value of the last computer in the order is minimal. We do that by induction on the number of computers. For a single computer, this is trivially true. Suppose we have **C** > 1 computers. By our inductive hypothesis, the first **C**-1 computers in the order were ordered and assigned values in a consistent way, with a minimal T value for the last computer among all options. Let us say the last computer in the full order is

computer i, and the next-to-last computer is computer j. By definition of how we assign missing values, $R_i = R_j$ if and only if $T_i = T_j$. If indeed $R_i = R_j$ and $T_i = T_j$, then the condition for the final assignment is equivalent to the inductive hypothesis. If computers i and j come from the same part, then the ordering choice between them was fixed, and the assignment of T values if needed is clearly minimal. So consider further the case in which computer i comes from a different part than computer j, and their R and T values are different. We have two cases: either computer i was in part R, or in part T.

If computer i was in part R, then its assigned T value is by definition the largest among all computers, and it's the smallest possible for it to go after computer j, whose value is minimal by the inductive hypothesis. As for the order, $R_i \leq$ **C**-1 per the limits. Since computer j comes from part T and was chosen for position **C**-1 (when N was **C** - 2), that means $R_i >$ **C** - 2. Therefore, R_i=**C**-1, and the chosen position is correct.

If, on the other hand, computer i was in part T, then its T value is minimal because T_i is fixed. As for the order, notice that all computers have either a T value strictly less than T_i or an R value strictly less than **C**-1, so none of them could have been last. By the inductive hypothesis, T_j is minimal among all possible orders, which means, by the existence of a full assignment, it has to be $T_j < T_i$, which implies the consistency of the final order and value assignment.

Code Jam 2020 - Round 2

Analysis: Wormhole in One

Test Set 1

As the limits for Test Set 1 are very small, we should be able to apply a brute-force algorithm and check all possible directions, starting points, and links between holes. However, there are infinitely many directions and starting points. Let's find a way to work with a finite number of possibilities.

First, let us observe that in order for the ball to touch more than two holes, some pairs of holes must be on lines parallel to the chosen direction. Otherwise, the best we can do is to link together two holes; the ball will go through them and will not touch any other holes.

So, when choosing an initial hit direction, we only need to consider those that are parallel to lines that connect pairs of holes.

Also, the exact starting point is not important. We can decide which hole we want to enter first, and then, regardless of which hit direction we choose, we can position the ball such that it will enter that hole (before any others). For example, because the holes are always at unique integer coordinates, we can choose a starting distance of 0.1 from the hole, in the direction that is the opposite of our hit direction.

Now, we can try all possible starting holes and linking schemes — with a recursive backtracking algorithm, for example — and choose the combination that touches the largest number of holes. This should be enough to pass Test Set 1.

Test Set 2

Suppose for now that we have chosen the direction of the hit. Let's calculate the maximum possible answer given that decision.

Imagine lines parallel to the chosen direction and going through all of the holes. Notice that each hole is on at most one such line. We will call a line an *odd line* if it contains an odd number of holes (but more than one), and an *even line* if it contains an even number of holes (at least two). If there are lines which only contain one hole, we call these holes *standalone holes*.

Let's also consider the holes along each line to be ordered in the chosen direction.

Note that:

- We cannot touch more than two standalone holes: one at the beginning, and one at the very end of the ball's journey.
- In the best case, we would touch all non-standalone holes.

Let's say we have C_{odd} total holes on the odd lines, C_{even} total holes on the even lines, and C_1 standalone holes. Then the answer is not greater than $C_{odd} + C_{even} + \min(2, C_1)$.

To touch two standalone holes, we should touch an even number of holes between them. To understand why, note that first of the standalone holes must be an entry to a wormhole, and the second one must be an exit from another wormhole. All of the holes that the ball touches in between those starting and ending holes must be linked in pairs by wormholes, which means there should be an even number of holes. This also means that if the number of non-standalone

holes is odd, then we will not be able to touch two standalone holes; in such a case, the answer will not be greater than $C_{odd} + C_{even} + \min(1, C_1)$.

As we will see, these upper limits can actually be achieved, so the answer is:

- $C_{odd} + C_{even} + \min(1, C_1)$, if $C_{odd} + C_{even}$ is odd
- $C_{odd} + C_{even} + \min(2, C_1)$, if $C_{odd} + C_{even}$ is even

Note that the parity of $C_{odd} + C_{even}$ is the same as the parity of C_{odd}, as C_{even} is always even.

Let's construct the linking scheme for the case when C_{odd} is even and C_1 is greater than 1:

1. Connect one standalone hole to the first hole of any even line.
2. Connect the other holes on that line pairwise consecutively (the last hole will remain unconnected).
3. Connect the last hole on that line to the first hole of another even line, and repeat steps 2 and 3 until only odd lines are left untouched.
4. Connect the last hole of the last even line to the first hole of any odd line.
5. Connect the second hole of that odd line (A) to the second hole of another odd line (B).
6. Connect the last hole of line B to the first hole of line B.
7. Connect the last hole of line A to the first hole of another odd line.
8. Connect the remaining holes of line A in consecutive pairs.
9. Connect the remaining holes of line B in consecutive pairs.
10. Repeat steps 5-9 until all the odd lines are used. When there are no more odd lines, connect the last hole of the last odd line to an unused standalone hole.

This scheme can be easily modified for the other cases, when C_{odd} is odd, and/or C_1 is less than 2.

Summarizing this, we can make full use of all of the odd and even lines and up to two standalone holes.

To calculate the number of holes on each line for a given direction, we can iterate through all ordered pairs of the holes and find the equation of the line that connects them in the form $y = mx + y_0$. Now, for each m, we store how many times we see each y_0. This number will be equal to the number of pairs of holes on this line, which we can use to calculate the number of holes. Note that we only need to do this once, as we will get the counts for all directions we are interested in.

Now, we can iterate through all the directions and calculate the answer for each one as described earlier by iterating through all the lines parallel to the current direction. Our final answer will be the maximum over the answers for all directions.

Note that even though there are $O(\mathbf{N}^2)$ directions, and $O(\mathbf{N})$ lines parallel to a direction, the total number of lines for all directions is $O(\mathbf{N}^2)$, as every line can be parallel to two (opposite) directions. So the total time complexity of this solution implemented optimally is $O(\mathbf{N}^2)$, but slower implementations might also pass.

Code Jam 2020 - Round 2

Analysis: Emacs++

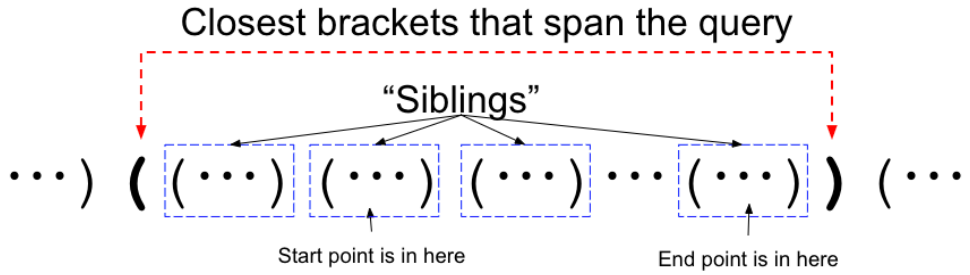
Test Set 1

TL;DR: Think of the brackets like a tree where a position's parent is the closest pair of brackets that contain that position. Go to the Lowest Common Ancestor, then back down the tree.

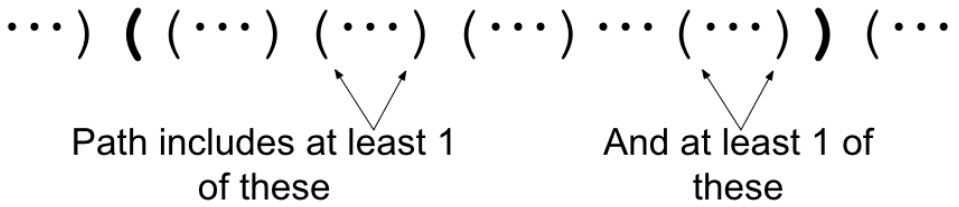
With this problem, our first thought may be to write a breadth first search for each query and add up the values. Unfortunately, this is much too slow for the bounds provided. Thankfully, we can make use of the structure of the Lisp++ program and the fact that all movement costs are 1 (going left, going right, or going to the matching bracket—we will call going left or right "walking" and going to the matching bracket "jumping" to make the explanation easier).

Two somewhat simple observations are needed before we can talk about the intended solution. The first observation needed is that we almost always want to jump instead of walking inside of a pair of brackets. If we are at a bracket endpoint, and jumping to the other endpoint doesn't make us "overshoot" the query's destination, then we should take that jump instead of walking there. The second observation needed is that if a pair of brackets contains both our current position and our destination, we should never move outside of that pair of brackets.

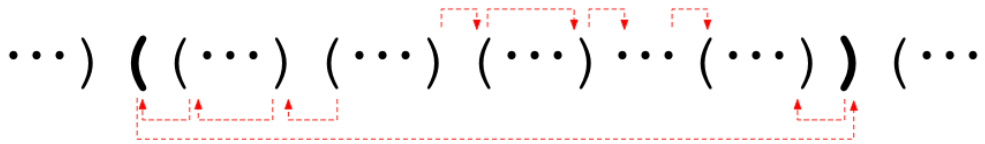
Using just these two observations, let's discuss what an optimal path looks like for a specific query. Consider the closest pair of brackets that contains both the start and end of the query:



The optimal path for this query must move "up" to this level:



One we are at this level, we jump along the top level of the siblings until we are at the brackets for the end query, then move "down" to the query answer. Note that we can jump either left or right (one of which will wrap around when we hit the bracket's endpoint) so we should check both and take the minimum.



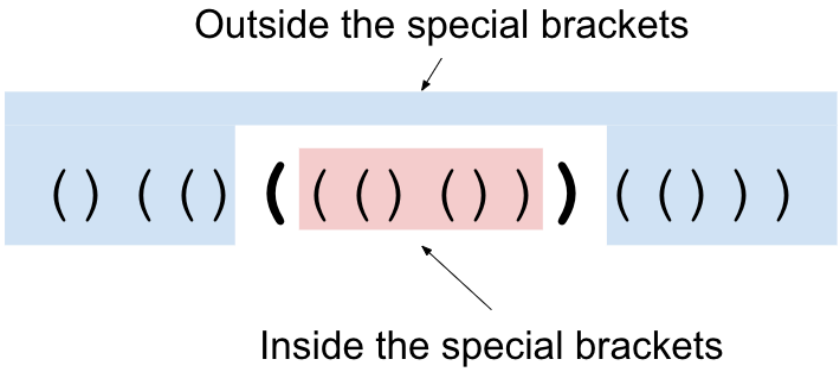
At this point, we treat the Lisp++ program as a tree with the parents in the tree being the closest pair of brackets that enclose us. Note that one of the parents is strictly closer than the other, so we will always go to that parent first on our path "up" or "down" the tree. The layer at the top is simply the lowest common ancestor in the tree, which can be computed in O(log **K**) time.

Test Set 2

TL;DR: (1) Find two pairs of brackets that partition the Lisp++ program into 4 disjoint sections. (2) Compute the shortest path from the break points to answer queries that go from one region to another. (3) Recurse on the 4 subregions.

For Test Set 2, some of the properties we used are no longer available to us. In particular, it may now be optimal to venture outside of the LCA described above. There are two main classes of solutions that are used to solve this test set. One is a modification of the LCA algorithm above. We will show the other here because it demonstrates an algorithm that is less traditional. Rather than answering the queries one at a time, we will employ a method where we can solve them in batches.

The key property we will be using is the ability to partition the Lisp++ program into multiple (almost) independent sections. First, let's start with a crucial observation. Consider a pair of matching brackets. The only way to get from inside the brackets to outside the brackets is to cross through the brackets themselves. These brackets that are used to split the input into the different sections will be called the *special brackets* throughout the explanation.



We can use this fact to answer queries differently. Rather than computing the distance from the starting point to the ending point of the query, we can compute the distance from each of the special brackets to the query's starting point and from the special brackets to the query's ending point. Since we know that any shortest path must go through one of the special brackets, we know that the sum of these two distances is the answer to the query.

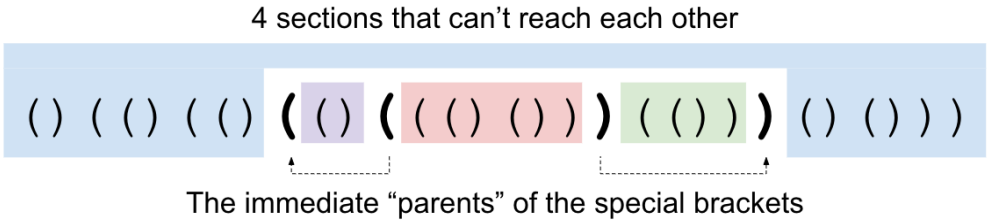
At first, it doesn't seem like this helps us. However, note that we can compute the answer to all queries that go from inside the brackets to outside the brackets at once! Just compute the distance from the special brackets to all locations (using Dijkstra's algorithm, for example) and use the method above.

Now what about all of the queries that start and end inside the special brackets (or outside the special brackets)? Well, the shortest path from the start to the end *might* go through the special brackets we chose, so we should make note of the potential answer if it does. Now, we're only interested in paths that do not use the special brackets.

Thus, we can split our problem up into two sub-problems: the "inside" part and the "outside" part. Split the queries up into their appropriate parts (inside and outside) and recursively solve each of these. Note that the special brackets can be removed completely since we know the

answers to any query involving them. There is one issue, though: this isn't necessarily fast enough. ☹ If we choose our special brackets poorly so that the "inside" is always a short string, then this algorithm will need $O(\mathbf{K}^2)$ time. In order to make this fast, we need to ensure that both subproblems are about half the size of the original problem. If we add in some more heuristics and choose our bracket pair randomly, we can make our code faster on average, but it is not guaranteed to pass the data. There is a slight tweak we can make described below which will save us!

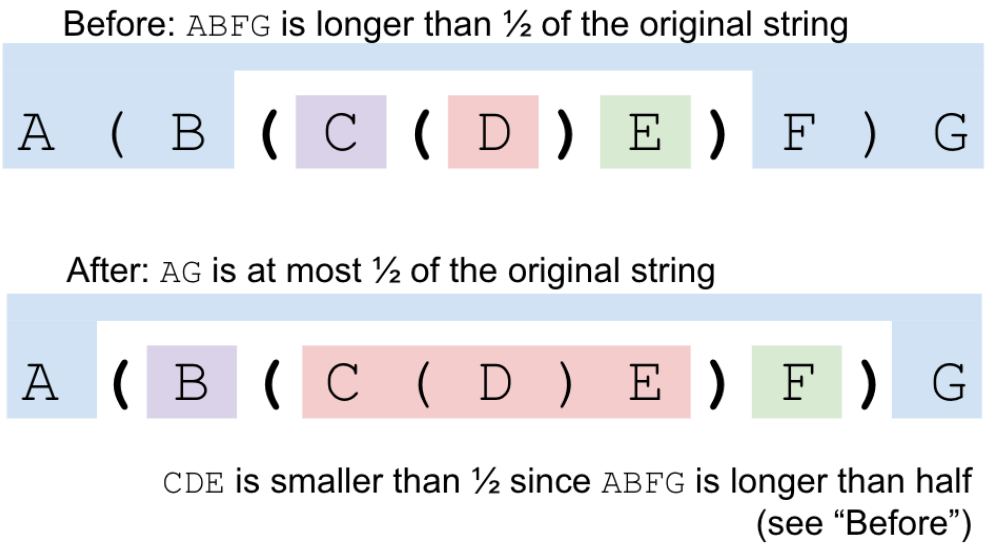
Instead of spliting the string into 2 sections, we will instead split the string into 4 sections. Consider a specific pair of brackets. Those brackets' *parents* are the closest pair of brackets that enclose them. If we split using a pair of brackets and their parents' brackets, we split our input into 4 sections. Note that it is impossible to get from those inner side pieces to the other side without crossing one of our two pairs of special brackets (since the parents are the closest brackets to our original brackets that would allow us to do that).



This small change looks like we just made things more complicated, but it solves our issue from above! First, let's add a pair of brackets to the outside of our string and set its $\mathbf{L_i}$, $\mathbf{R_i}$, and $\mathbf{P_i}$ to infinity. This way, all pairs of brackets have a parent, except for this new infinite pair we added.

Let's consider all the bracket pairs whose span (from starting bracket to closing bracket, inclusive) includes the middle bracket (there are two "middle" brackets; we can choose either). Call these brackets the "middle line brackets". The middle line brackets will form a chain in which each bracket pair nests under another middle line bracket, or is the outermost bracket that we added. Our middle line brackets have some nice properties that we can use.

If we consider the middle line brackets from outermost to innermost, we can observe that the spans of the brackets go from containing more than half the characters (the outermost bracket that we added spans the whole string) in our Lisp++ program, to containing at most half of the characters. This is because they are always getting smaller, and the innermost one spans at most half of all characters. Let's consider this "pivot point". It comprises a pair of consecutive middle line brackets in which one spans more than half, and it has a direct child that is also a middle line bracket that spans at most half. If we take these two brackets and cut them out of the string, we will have broken the string into 4 disjoint (possibly empty) parts, none of which contain more than half of the characters in the string.



Why? Let's say our brackets look like this `A ( B ( C ) D ) A`. We know that since the outer bracket pair spans more than half, the region `A` contains less then half of the characters. We know that the inner bracket either crosses or touches the middle line, so the regions `B` and `D` contain less than half of the characters. Finally, `C` has at most half (remember that we chose `C` specifically because it was the first middle line bracket that contained at most half of the characters). Note that we cannot get from one region to another without crossing a special bracket. In particular, we cannot get between `B` and `D` because the outer special brackets are the parent of the inner special brackets.

Thus, we can solve the problem by finding the two pairs of special brackets we are going to split on, using Dijkstra's algorithm to answer the queries that go between different regions (as well as compute potential answers for those queries that do not), and then recursing into the 4 sub-problems. Since each recursion cuts the length of the input string in half, we recurse at most O(log **K**) times. The sum of the strings at any particular depth of the recursion is at most the length of the original string. So the total work we needed to do at each layer is at most O(**K** log **K**) to run Dijkstra from our 4 special brackets. Also, each query is looked at at most once on each layer of the recursion, so the total complexity is O(**K** $\log^2$ **K** + **Q** log **K**).

Some Common Issues

Here is a list of common issues that might explain a Wrong Answer or Time Limit Exceeded verdict:

The edges are directed! This means that the distance from A to B is not necessarily the same as the distance from B to A. For the intended solution, this means we need to run Dijkstra's algorithm twice, not just once.
If we place values in our code about "infinity", those values must be large enough, but not so large as to cause overflows.
Picking a random edge and just breaking it into doing inside and outside is in general too slow. Even if we break the left and the right apart if they're not connected, we can run into issues. Consider the following. Let X=`(((((  ...  )))))`, be of length sqrt(N). If the input is XXXXX ... XXXX sqrt(N) times, then picking randomly is not very efficient. In order to split the input, we need to get lucky and hit one of the outer points of X in order to really cut down on the size of the input.