Code Jam 2017 - World Finals

Analysis: Dice Straight

Dice Straight: Analysis

Small dataset

One brute force approach to the problem is to examine all possible subsets of dice in all possible orders, and look for the longest straights. Since there may be as many as 100 dice in the Small dataset, we need a better strategy.

First, let's create the set of all integers that are present on at least one die, and sort that set in increasing order. Then we will create an interval that initially contains only the first number in that sorted list. We will expand and contract that interval according to the following strategy. As an invariant, the entire interval will always be a straight (a sequence of one or more consecutive numbers).

Check whether it is possible to build the straight using the available dice (we'll get to how to do that in a moment).

- If it is possible: Expand the interval to include the next value to the right.
    - If that value is not one greater than the previous rightmost value, then we no longer have a straight; contract the interval to remove all but that new rightmost value. Then we have a straight again.
- If it is not possible: Contract the interval by removing its leftmost value.

To check whether a possible straight can be built, we can find a bipartite matching from the required integers to the dice by using a flow algorithm such as Ford-Fulkerson. Since each die has exactly 6 faces, the number of edges in the graph is 6**N**, and the running time of one iteration of Ford-Fulkerson is O(**N**^{2}). We run it up to O(**N**) times as we adjust our interval, so this solution is O(**N**^{3}). Other polynomial solutions are possible.

Large dataset

To make the flow algorithm fast enough to solve the Large dataset, we need the additional insight that we do not need to start the algorithm from scratch every time. When expanding by adding a new number, we need to add all O(**N**) edges into that number (either by adding them outright or changing their flow capacity from 0 to 1), and then find and add a single augmenting path to the existing flow, which also takes O(**N**) time (since the total number of edges in the graph is at most 6**N**). So an expansion takes O(**N**) time overall. When contracting, we need to remove edges and flow from the path associated with that number; this takes O(**N**) time. Since we have O(**N**) expansions and contractions, they take O(**N**^{2}) time in total. Adding this to the O(**N**^{2}) from completely solving the flow problem the first time, the overall complexity is still O(**N**^{2}). This is fast enough for the Large dataset.

Code Jam 2017 - World Finals

Analysis: Operation

Operation: Analysis

Small dataset

Let us consider the Small dataset first. The number of operations is large enough that trying all possible orderings would time out, but a typical trick might work: turn that **N**! into a $2^{\mathbf{N}}$ with dynamic programming over subsets of cards. That is, try to reduce all possible orderings of a subset of the cards to only few possible results that are worth exploring further. The first option to try is to define a function f(C), for a subset C of the cards, as the best possible result of applying cards in C to the starting value **S**. It seems that defining f(C) as the maximum over all c in C of applying c to f(C - c) could be reasonable (with f(∅) being **S**). However, it doesn't quite work in general. For instance, suppose c is `* -1`. We are getting the maximum possible result out of f(C - c), only to flip the sign right after. It would have been better to get the minimum possible result for f(C - c) instead. Of course, if c is `* 2` instead, getting the maximum for f(C - c) seems like a good idea. It turns out that the best option is always either the minimum or the maximum for f(C - c), which we prove below. Therefore, let g(C, maximum) be the maximum possible result of applying cards in C to **S**, and g(C, minimum) be the minimum among those results. We can define g(C, m) recursively as the "m" (i.e., the maximum or the minimum depending on m) over each c and each m' in {minimum, maximum} of applying c to g(C - c, m'), with g(∅) = **S**. This definition of g formalizes our intuition that only the minimum and maximum possible values are needed from each subset. We prove its correctness below. The result is then g(C, maximum) for C = the entire input set. The function has a domain of size $2^{\mathbf{N}} \times 2$, and calculating each value involves an iteration over at most $2 \times \mathbf{N}$ possibilities, yielding $O(2^{\mathbf{N}} \times \mathbf{N})$ operations in total after memoizing the recursion. Of course, those operations are over large-ish integers. The number of digits of those integers is bounded by O(**N**D) where D is the number of digits of the input operands. That means the time complexity of each operation, which operates on a large integer and an input integer with up to D digits, is bounded by $O(\mathbf{N}D^2)$, which makes the overall running time of this algorithm $O(2^{\mathbf{N}} \times \mathbf{N}^2 \times D^2)$.

We can prove the definition of g is correct by complete induction. If C is the empty set, then g is correct immediately by definition. Otherwise, assume g is correct for all m' and all sets C' smaller than C, and let us prove that g(C, m) is correct. Let c be the last card used in an ordering of C that gives the "m" result when applied to **S**. If c is +v or -v, we can commute the operator "m" with the application of c. That is: let T be the result of applying all of the other operations in the optimal order. Then we know that T + v or T - v is "m" over the operations, so if the value of T is not the optimal g(C - c, m), then there is some other ordering that yields g(C - c, m) + v or g(C - c, m) - v, which is better. The same is true for c being a multiplication or division by a non-negative, since those also commute with maximum and minimum. If c is a multiplication or division by a negative, then it can commute with a maximum or minimum operator, but the operator is reversed, that is, max turns into min, and vice versa. Since we try both maximum and minimum in the definition of g, that poses no problem. Notice that this proof also shows that we do not even need to try both options for m'; we only need to check the one that actually works. Trying both is simpler, though, and it doesn't impact the running time in a significant way.

Large dataset

Of course, an exponential running time would not work for the Large dataset, so we need to reason further. As a first simplification, assume all additions and subtractions have positive operands by removing those with a zero operand, and flipping both the sign and the operand of those with a negative operand. This leaves an input with the same answer as the original one.

Suppose all cards are already ordered forming an expression E. We can distribute to "move" all additions and subtractions to the right end creating a new expression E' that contains multiplications and divisions first, and additions and subtractions later. In order to make the value of E the same as the value of E', we change the added or subtracted value on each term. The value of a given addition or subtraction card is going to be multiplied/divided by all multiplications/divisions that are to its right in E. For instance, if E = (((0 + 1) × 4) - 6) / 2, then E' = ((0 × 4) / 2) + 2 - 3. Notice "+ 1" turned into "+ 2" because it is multiplied by 4 and divided by 2. "- 6" turned into "- 3" because it is only divided by 2 (the multiplication in E does not affect it).

If we consider an initial fixed card "+**S**", we can even move that one to the end and always start with a 0, making multiplications and divisions in E effectively not needed in E'. The final result is then the sum over all additions minus the sum over all subtractions of the adjusted values (in the example above 4 is the adjusted value of the only addition and 3 is the adjusted value of the only subtraction). Notice that this shows the value of **S** always impacts the final result in the same way regardless of the order: it adds **S** times the product of all multiplications and divided by all divisions to the result.

Consider a fixed order for multiplications and divisions, and insert the additions and subtractions. As explained in the previous paragraph, inserting operation Z at a given position implies that Z will get multiplied by all multiplications that are to its right, and divided by all divisions that are to its right. Effectively, each position has a fraction F such that operations inserted there get multiplied by F. Given the view of the final result above, it follows that it's always best to insert additions at a position where F is maximal, and subtractions at a position where F is minimal. Even though there could be multiple places with the same value of F, this shows that it is never suboptimal to insert all additions in the same place A, and all subtractions in the same place B. Since additions and subtractions commute, this further shows that it is equivalent to have an input with a single addition and a single subtraction whose operand is the sum of the operands of the corresponding operation (after adjusting the signs to make them all positive).

Given the simplification, we can reverse the point of view. Consider the addition and subtraction fixed and insert multiplications and divisions. Since multiplication and division commute, we just need to decide between 3 places: a) before both addition and subtraction, b) in between, c) after both. What we insert in a) will multiply/divide only **S** in the final result, what we insert in b) will multiply/divide **S** and only additions or only subtractions depending on which is earlier, and what we insert in c) will multiply/divide everything. If we fix the positions of multiplications and divisions with negative operands, we can greedily place multiplications and divisions with a positive operand: we place multiplications to apply to the greatest of the 3 values mentioned above (after applying the fixed multiplications and divisions by a negative) and divisions to apply to the lesser of the 3 (after doing the same). This shows that it is never suboptimal to place all multiplications by a positive together in one place, and all divisions by a positive in one place.

To further simplify, divisions can't have a zero operand, but multiplications can. Notice that a "* 0" will nullify the entire thing, so only the placement of the rightmost "* 0" matters, so we can simplify them all into a single card (or no card if there is no "* 0" in the input). This leaves only multiplications and divisions by a negative. If a pair of multiplications by a negative are in the same place a), b) or c) as above, they multiply as a positive, so it is always better to move the pair to the optimal place to put multiplications, as long as we have pairs. If there is an even number of multiplications by a negative in a suboptimal place, then all of them get moved by this. If their number is odd, all but one are moved. Leaving behind the one with the smallest absolute value is optimal. A similar argument applies to divisions by a negative, although the optimal place to move to may of course be different than the optimal place to move

multiplications. This shows that we can group multiplications and divisions by a negative similarly to what we did with all other operations, but leaving out the two smallest absolute values of each type, as they may be needed in suboptimal places to perform a sign change (there are 3 places out of which 1 is optimal, leaving 2 suboptimal places).

After all the groupings, we are left with at most 11 cards: 1 addition, 1 subtraction, 1 multiplication by 0, 1 multiplication by a positive, 1 division by a positive, 3 multiplications by negatives and 3 divisions by negatives. This can be refined further, but there's no need for it. With just 11 cards, we can apply the Small solution and finish the problem. There are also other ways of reasoning among similar lines to get the number of cards low enough. Another possibility that doesn't require any dynamic programming is to notice that we can brute-force the order of the addition and subtraction (two options), and then brute-force which place a), b) or c) each multiplication and division (up to 8 cards after all simplifications) should go into. This requires exploring only 2×3^8 combinations in total, and exploring each one is a relatively fast process requiring only $O(\mathbf{N}^2)$ time (11 operations of quadratic time each, since the operands in the simplified cards can have linear size).

It is possible to reduce the set of cards further with more thought, and it's also possible to follow other lines of reasoning that will lead you to slightly higher card totals that are small enough to make the dynamic programming solution work.

Code Jam 2017 - World Finals

Analysis: Spanning Planning

Spanning Planning: Analysis

In general, for any value of X greater than 2, it is possible to construct a graph that has X spanning trees: connect X vertices to form a single cycle. Then there are X ways to delete a single edge, and each of these deletions leaves behind a different spanning tree. However, in this problem, we are only allowed up to 22 vertices, so we can only use that strategy for values of **K** up to 22.

The statement guarantees that an answer exists for every **K** in the range [3, 10000], so we just need to find those answers. Since there are so few possible testcases, we can prepare ourselves by finding one answer for each possible **K** before ever downloading a dataset.

A good way to start exploring the problem is to create random graphs and count their spanning trees. We can use the beautiful Kirchhoff matrix tree theorem to reduce the problem of counting spanning trees in a graph to the problem of finding the determinant of a matrix based on the graph. We can either carefully implement our own function for finding determinants, or use a library function, e.g., from SciPy. We must take care to avoid overflow; a complete 22-node graph has 22^{20} trees! Precision loss is also a potential concern if working outside of rational numbers, but one of our internal solutions successfully used Gaussian elimination and doubles.

It turns out that we cannot find all the answers we need via a purely random search, but we can find most of them, which provides some hope that we can reach the others with some tweaking. For example, we can change the graph size and the probability that each edge exists; empirically, 13 nodes and an existence probability of 1/4 work well.

Another strategy is to apply small modifications to a graph, hoping to converge on one of the desired numbers of trees. Specifically, we can remember what numbers of trees we still need to get, pick one of them as a "target", and then repeatedly add edges if we have fewer trees than the target or remove them (taking care not to disconnect the graph) if we have more trees than that target. Once we reach our goal, we pick another still-unreached number as the new goal, and so on. To make this finish quickly, we can mark all visited numbers as reached even if we reach them while in pursuit of a different goal.

In both of the above approaches, generating solutions for **K** = 13 and **K** = 22 can take a very long time, since those numbers apparently require very specific graph structures. However, since we're allowed 22 vertices, we can get the answers from those numbers by building a cycle with 13 or 22 vertices, as described earlier.

This problem is inherently experimental, and requires research on a computer; you cannot just write down a solution on paper. (If you do know of a tractable construction-based solution, we'd love to hear about it!) However, this sort of exploratory research skill is valuable in real-world programming, and we like to reward it in Code Jam, particularly in problems in which it is surprising that a randomized strategy works at all.

Code Jam 2017 - World Finals

Analysis: Omnicircumnavigation

Omnicircumnavigation: Analysis

The concept of omnicircumnavigation requires a given itinerary to touch every possible hemisphere. Pick any plane P that contains the origin. The plane splits the surface of the sphere into 3 parts — two open hemispheres and a circle between them. If the travel path lies entirely within one of the hemispheres, then it is not an omnicircumnavigation. The problem is to find such a *dividing plane* P or prove that one does not exist.

Continuing with that reasoning, the travel path touches a plane P if and only if one of the stops is on P, or there are stops on both hemispheres so that the connection between them passes through P. Therefore, another equivalent definition of dividing plane of a travel path is a plane that has all stops strictly on one of the hemispheres. This means the order of the input points is not important: the answer is the same for any valid permutation of a given set of points!

Notice that each actual stop S is given in the input by giving another point S' such that S is the normalized vector of S'. That means the origin, S and S' are collinear, which in turn implies that any plane P that contains the origin leaves both S and S' on the same hemisphere. Then, checking for the actual stops to be on a side of a plane is equivalent to checking the points S' given as input. And since points S' have the lovely property of having all integer coordinates, it is much better (and precise) to use them directly.

To summarize, the problem we are presented is equivalent to a simplified formulation: given a set X of points with integer coordinates, decide whether there exists a plane P that contains the origin such that all points in X lie strictly on the same side of P. Let us call a plane that goes through the origin and leaves all points strictly on one side a *dividing plane*.

Convex-hull based solutions

Notice that if there exists a dividing plane P, then P also has the convex hull of all points on one side. Moreover, by convexity, a dividing plane exists if and only if the convex hull does not contain the origin. So, one possible solution, that can even work for the Large, is to calculate the convex hull and check whether it contains the origin. If you do this using a library, it might be easier to calculate the convex hull of X plus the origin and check whether the origin is indeed a vertex of it. This solution, however, has two major drawbacks: 1. the algorithm to do convex hull in 3d is pretty hard, and 2. many implementations will have either precision issues, overflow problems, or get slowed down by handling increasingly big integers. This is because the needed plane can be really skewed, with angles within the 10^{-6} order of magnitude. Moreover, if the entire input is coplanar, then the convex hull might fail. One way to take care of both problems is to calculate the convex hull of X plus the origin plus F where F is a point really far away. F is not going to be coplanar, and it will also make the convex hull not have extremely narrow parts. Of course, the addition of F may make the convex hull contain the origin when the original did not. We can solve that with a second pass using the antipode of F, -F. If the original convex hull contained the origin, then both of the passes will. If the original convex hull didn't, then at least one of them won't (the one where F or -F is on the appropriate side of the dividing plane, since they are necessarily on different sides).

A simplified way to check for this is to notice that, in the same way there is a triangulation for any convex polygon, there is a tetrahedralization of any polyhedron. That means, we can avoid explicitly calculating the convex hull if we check all possible tetrahedra. This can't give false positives because all of them are included in the convex hull, and since some subset of those

tetrahedra will actually be a partition of the convex hull, their union is the entire convex hull, and one of them contains the origin. We can conclude that the convex hull of X contains the origin if and only if some tetrahedron with vertices in X does. The coplanar case can be simplified in this case: if the entire input is coplanar, we can check for any triangle to contain the origin. This solution, however, takes time $O(\mathbf{N}^4)$, which is definitely too slow for the Large dataset, and might even be slow for the Small, given that checking for each tetrahedron to contain the origin requires quite a few multiplications, which takes significant, though constant, time. The coplanar edge case of this solution can also be avoided by adding phantom points F and -F as above.

A speedup of the solution above that is certainly fast enough to pass the Small is to notice we can fix one of the vertices and try every possible combination of the other 3. This is because, for any vertex V, there is a tetrahedralization of the convex hull such that all tetrahedra in it have V as a vertex. This takes the running time down to $O(\mathbf{N}^3)$, which is definitely fast enough for the Small dataset, even with a large constant.

Solutions based on restricting the dividing planes

As usual in geometry problems, we can restrict the search space from the infinitely many possibilities to just a few. Suppose there is a dividing plane P. If we rotate P while touching the origin, we will eventually touch at least one point S from the input. If we rotate while around the line between S and the origin, we will eventually touch another point from the input. That means we can restrict planes P to those who touch two points from the input. Of course, the plane P is not the dividing plane (since it touches points from the input), but P represents planes epsilon away from those touching points. This means we need to take special care of inequalities to make sure that small rotation doesn't ruin the solution. In short, if there is another point touching P, we can't necessarily rotate P to make all 3 points on P to lie on one side. We need to take care of this coplanar case separately, with either solving the 2-dimensional version of the problem, or using phantom poins. Since 3 points (two points from the input and the origin) completely determine a plane, this restricts the number of planes to try to $O(\mathbf{N}^2)$ possibilities. For each one, we need another pass through the input to check on which side of the plane each point lies. This yields a solution that runs in time $O(\mathbf{N}^3)$, which is enough to pass the Small. Even in fast languages, this can be too slow for the Large, as the constant is, once again, very significant.

The solution above can be made run much faster, by randomizing the input, and thus, the order in which we check the points. For most planes, there will be several points on either side, and then when checking in a random order, the expected time to find one on each side (after which, we can stop) can be greatly reduced. Notice, however, that a case in which all the points are coplanar will not have its running time improved by this randomization, as no point will fall strictly on one side. For this to work well, we need to check for the all-coplanar case and special-case that one before launching the general case algorithm. This randomized version is indeed enough to pass the Large.

A bazooka-to-kill-a-fly solution

And finally, we can use linear programming. A plane that contains the origin is defined by an equation $Ax + By + Cz = 0$, for some A, B and C. A plane that has all points on one side has to satisfy either $AX + BY + CZ > 0$ for each point (X, Y, Z) or $AX + BY + CZ < 0$ for (X, Y, Z). Notice that if a triple (A, B, C) satisfies one of the conditions, then (-A, -B, -C) satisfies the other. So, we can restrict ourselves to one of the two cases, and then define a polytope with the set of inequalities $AX + BY + CZ > 0$ for each (X, Y, Z) in the input. The answer to the problem is whether that polytope is empty. Most LP algorithms figure that as an intermediate result towards optimization, and some libraries may provide functionality to check that directly. For others, you can just optimize the constant function 0 and see whether the answer is indeed 0 or "no solution".

As simple as the description of this solution is, it has a lot of issues to resolve. If using a library, it is highly likely that you run into similar precision / large number problems as the convex hull (and for the same reasons) that may make it either wrong or slow or both. If you want to implement your own algorithm, well, it's long and cumbersome to do it and avoid precision problems. There are tricks here, too. We can catch the possible problems and try to rotate the input to see if the library performs better. We can add additional restrictions like bound A, B and C to absolute values up to 10^6 to have a bounded polytope to begin with. That being said, judges tried 4 different LP libraries, and only one of them worked, after adding both additional restrictions (the library wouldn't handle unbounded spaces) and rotating the input a few times. Adding far-away phantom points can also help the LP, because it avoids the same problems as in the convex hull case. Of course, if you had a really robust prewritten algorithm or library, this option was best, even enabling a possible 3-line solution that passes the Small and the Large.

Code Jam 2017 - World Finals

Analysis: Stack Management

Stack Management: Analysis

This problem requires quite a lot of thinking, but relatively little code in the end!

Let S be the number of suits that appear on at least one card used in the problem. For each of these suits, we will call the card with the highest value the *ace* of that suit, and the second-highest card (if it exists) the *king*. We'll say that a card is *visible* if it is at the top of one of the stacks, and that a suit is *visible* if a card of that suit is visible.

Notice that once a suit becomes visible, it will stay visible throughout the rest of the game. At any point in the game, if there are **N** visible suits (recall that **N** is the number of stacks), then either we have won, or we cannot make any more moves and so we have lost. On the other hand, if there are fewer than **N** visible suits, then either we have won, or we are able to make a move (because either a suit has two visible cards, or there is an empty stack). In particular, this implies that if S < **N**, then we are guaranteed to win however we play. However, if S > **N**, we cannot win, because we can never remove the last card of any suit from the game. This means that S = **N** is the only interesting case, so we will assume from now on that S = **N**. In this case, a winning position is a position in which we have exactly one card in each stack, with each one representing a different suit.

Let us assume there is some way to win the game, and we will consider the very last move of that game. Before that move, we had not won, and yet we were able to make a move; this means there must have been fewer than **N** suits visible. So, the last move must have exposed a card of some suit that had never been visible. This means that this suit contained only one card (the card that now remains as the only card in its stack), and this card started the game at the bottom of its stack.

Also note that it is never disadvantageous to remove a card; the only real decisions made in the game are choosing which cards to move into an empty stack. Thus, as a pre-processing step, we can remove all the cards that can be removed from the initial position. If doing that is enough to win the game, we are done. If doing that leaves us with no empty stacks, we are also "done", because we have lost the game! So, let's assume that when we begin, there is at least one empty stack.

We will now aim to prove that the following condition is necessary and sufficient for a game to be winnable. Let us construct a graph in which vertices are the suits for which the ace begins the game at the bottom of some stack. We say that a vertex (suit) *s* is a *source* if the ace is the only card in this suit, and that *s* is a *target* if there is another ace (of a different suit) in the stack in which the ace of *s* is at the bottom. We add an edge from vertex s_1 to a different vertex s_2 if the king of s_2 is in the stack that has the ace of s_1 at the bottom.

Now, we claim the game is winnable if and only if there exists any path from some source vertex to some target vertex.

To understand this condition, consider a simple case in which there is a single edge from a source suit s_1 to a target suit s_2; i.e., suit s_1 has exactly one card (an ace), which is at the bottom of a stack *A*, and suit s_2's king is in *A*. To ensure that s_1 is a source but not a target, assume that there are no other aces in *A*, and to ensure that s_2 is a target, assume that its ace

is at the bottom of a different stack *B*, and there is a third suit s_3 that has an ace higher up in *B*, which we will assume is the ace nearest the bottom except for the bottmmost card in *B*.

The winning strategy in this case is as follows. First, make all legal moves until the only remaining legal move is moving the ace of s_3 to an empty pile. Since we won't uncover the ace of s_1, there will be fewer than **N** suits visible, so this state is always achievable. When we reach this state, all of the following are true:

- There is an empty stack (since s_1 isn't visible, we'd otherwise have two cards visible in one suit, and could remove the one with the lower value).
- Stacks *A* and *B* are the only stacks with more than one card. (Otherwise, we could move a card from from one of the other stacks into the empty space.)
- The other **N**-3 stacks (aside from *A*, *B* and the aforementioned empty stack) each contain an ace of one of the remaining **N**-3 colors. (We couldn't have removed the aces, and they are not in stack *A* or stack *B*).

At this point, we will move the ace of s_3 to the empty stack, and then try to remove cards from *B*, until we get down to the ace of s_2. If the top card of *B* isn't yet the ace of s_2, then it's either in suit s_2 (and lower than the king, so we can remove it, because the king is visible), or in some other suit (in which case the ace of that suit is visible, and we can remove it). Therefore, we can remove cards down to the ace of s_2, then remove the king of s_2, and then again dig down to the ace of s_1. We can do this because any card in *A* other than the ace of s_1 will be removable, since we now see all the aces other than s_1, and there are no cards but the ace in suit s_1.

The description above can be extended relatively easily to show how to win the game when a longer path exists. First, we clean up everything but the aces mentioned in the path, and then move the ace from the end of the path into the empty space, and remove all the remaining cards one by one. So, what remains is to prove that if the game is winnable, a path from a source to a target always exists in the graph we constructed.

At the end of a successful game, each of the **N** stacks will contain one of the **N** aces. Whenever we move an ace to the bottom of a stack, it will never again be covered. So, before the last move action, **N**-1 aces will be on the bottom of a stack, and the last move is necesarilly moving an ace to the empty spot. Some of the aces are visible before the last move. Nothing interesting will happen to cards in those suits - we might uncover a card in one of those suits, and then we will be able to immediately remove it, because the ace is visible. The more interesting suits are the ones in which the aces are not visible and are at the bottoms of stacks. Note that once uncovered, an ace cannot be covered again, so these aces had to be at the bottoms of their stacks from the beginning of the game, and the cards on top of them had to be there from the beginning of the game. So, it's enough to prove that we will see a source-target path in the position before the last move. This will mean that the path was there from the beginning of the game.

The cards on top of the other stacks with more than one card have to be in the same set of colors as the covered aces (or else they would have already been removed). We want to prove they are all kings. We will proceed by contradiction: assume that one of them is a lower card (say, a "queen of spades"). Since it is visible and not removed, the king of spades must be somewhere in one of the stacks, and not visible; it cannot have been removed yet, because in this case the ace would have to be visible, and the queen would have been removed as well.

Consider what happens if we move this queen into the empty space. We experience a sequence of removals, which cannot end with removing the queen (that would contradict the assumption that no more moves can be made before the winning one). Thus, it has to end in uncovering the ace of the suit that was not previously visible (causing us to lose the game) - let's call this suit "diamonds".

Now, consider the winning move instead. We also end up with a sequence of removals. After each removal except the last one, we see **N**-1 suits, and so we have exactly one choice what to remove. So, we deterministically remove cards until we, at some point, uncover the king or the ace of spades, whichever comes first, and that causes us to remove the queen of spades... and then execute the exact same deterministic sequence of removals that, in the end, caused us to uncover the ace of diamonds. Note that the other high spade card (whichever among the king and ace that we did not see) is not uncovered in this sequence. If it were, it would have removed the queen of spades in the previous scenario - so, we end up with at least one card that is not visible, which is a contradiction.

So, we have proven that kings are on top of our stacks with (non-visible) aces on the bottom. At this point, following the graph from the ace that was the source will eventually lead us to the target: the stack with two aces.

After establishing all this, the algorithm to check for the desired condition is fairly simple. After constructing our graph, we can start at sources and perform a depth-first search to see if there is a path from any of them to a target. This is considerably faster than running a backtracking algorithm on the set of moves itself, which works for the Small but not for the Large.

Code Jam 2017 - World Finals

Analysis: Teleporters

Teleporters: Analysis

This problem starts off with a strange twist, because instead of the regular Euclidean geometry (also known as L2 geometry), it uses L1 geometry. The reason, if you are curious, is that the distance between two points with integer coordinates is also an integer. A Euclidean geometry version of this had serious precision issues, and L1 geometry has all the properties that are needed. A concept that will come up later is the set of points that are at a particular distance from a center. In Euclidean geometry, that's just a sphere. In L1 geometry, though, is an octahedron with diagonals (segments that connect opposite vertices, passing through the center) parallel to the axis. Luckily, most intuitive properties of spheres in Euclidean geometry also work for spheres in L1 geometry, with the important exception of rotations, that are not used in this problem. If it helps you to visualize, for remainder of the text, you can think of the problem as working in L2 geometry. However, throughout the rest of this analysis, every time we say "distance", we are referring to L1 distance; we will write dist(x, y) for the L1 distance between points x and y. Every time we say "sphere", we are referring to L1 spheres (i.e., regular octahedra).

Small dataset

Let us start slowly: after 0 teleportations starting from a point P, the only reachable place is point P. After 1 teleportation, reachable places depend on which teleporter we used. If we use teleporter t, reachable points are the surface of the sphere with center t and radius dist(P, t). Let R_i be the set of points that are reachable in i teleportations using any set of teleporters. As we mentioned above, $R_0 = \{P\}$ and R_1 is a union of **N** sphere surfaces, one centered on each teleporter. What about R_2? If we use teleporter u for the second teleportation, we can land at any point that is at distance d from u, where d can take the value of any distance betwen u and a point in R_1. This implies R_2, and all other R_i, are also a union of sphere surfaces, possibly infinitely many.

Notice that R_1 is a connected continuous set because all the spheres' surfaces intersect at point P. Thus, the values for the distance d in the definition above form an interval, since the distance function is continuous. This implies that R_2 is actually a union of sphere differences: for each teleporter t, all points x such that $L_{t,2} \le dist(x, t) \le U_{t,2}$ are reachable. (Throughout these explanations, we use L to refer to a lower bound on a reachable range, and U for a corresponding upper bound.) Once again, all the sphere differences contain P, thus, they intersect and R_2 is a connected continuous set, which means the distances we use to calculate R_3 are intervals. This argument generalizes to prove by induction that each R_i is exactly the union over all teleporters t of all points x such that $L_{t,i} \le dist(x, t) \le U_{t,i}$.

This yields a dynamic programming algorithm that solves the Small dataset. Keep two arrays L and U representing the values $L_{t,i}$ and $U_{t,i}$ for a particular i and use them to calculate the next values $L_{t,i+1}$ and $U_{t,i+1}$. After each iteration, check whether $L_{t,i} \le dist(Q, t) \le U_{t,i}$ for some t, and if so, i is the answer.

By definition, $L_{t,i+1}$ and $U_{t,i+1}$ are the distances from t to its closest and farthest points in R_i, respectively. The farthest point in R_i from t is at a distance which is the maximum over all teleporters u of $dist(t, u) + U_{u,i}$ (this is the distance to the point on the surface of the sphere

centered at u with radius $U_{u,i}$ that is the opposite direction from t). The closest point is slightly more complicated. For each teleporter u we need to consider:

- dist(t, u) - $U_{u,i}$ if dist(t, u) > $U_{u,i}$ (t is outside the outer sphere centered at u),
- $L_{u,i}$ - dist(t, u) if dist(t, u) < $L_{u,i}$ (t is inside the inner sphere), or
- 0, in all other cases (t is in between, that is, it is itself a reachable point).

This means we can calculate each $L_{t,i}$ and $U_{t,i}$ in O(**N**) by comparing the values above for each teleporter u.

Notice that a point reachable in i teleportations is also reachable in i+1, i+2, ... etc teleportations, because you can use a teleporter to move from a point to itself. Thus, $U_{t,i}$ is non-decreasing with i, and $L_{t,i}$ is non-increasing with i. Additionally, since the distances dist(t, u) are positive, when **N** ≥ 2, the maximum over all t of $U_{t,i}$ is strictly increasing with i, and the minimum over all t of $L_{t,i}$ is strictly decreasing with i up to the first j where $L_{t,j} = 0$. That means, for **N** ≥ 2, the intervals grow until one of them represents a sphere covering the entire cube of values for Q within the limits. Moreover, since the values are integers, the increase and decrease is at least 1 per iteration, so the number of iterations needed to cover the entire region of valid Qs is bounded by 3M (M on each direction), where M is the number of valid coordinates, which is only 2001 in the Small dataset. This in particular means that for **N** ≥ 2 the answer is never impossible. For **N**=1, we can note that using the same teleporter twice in a row is never useful, so after 1 iteration, if Q is not reached, the answer is impossible.

The time complexity of the presented algorithm is O(M $\mathbf{N}^2$): up to 3M steps, each of which requires calculating O(**N**) values, and calculating each one requires an iteration over **N** other teleporters and constant-time math.

Large dataset

Of course, when M is bounded by $2 \times 10^{12} + 1$, a time complexity linear on M won't finish fast enough, so we have to do something else.

Let us focus first on deciding if it's possible to go from P to Q with a single teleportation. That means using a single teleporter t, and due to conservation of distance, it must be dist(P, t) = dist(Q, t). Moreover, this condition is sufficient and necessary for the answer to be 1. We can check for this condition initially with a loop over all teleporters in O(**N**) time.

As we saw on the Small, checking whether the answer is 1 is sufficient to fully solve cases with **N** = 1. We assume further that **N** ≥ 2.

Let us now consider the case where there exists two teleporters t and u such that dist(P, t) ≥ dist(Q, t) and dist(P, u) ≤ dist(Q, u). Consider the sphere A centered at t that passes through P, and the sphere B centered at u that passes through Q. By the assumed inequalities, A contains Q and B contains P, which means A and B intersect. Let x be any point at the intersection, for which dist(P, t) = dist(x, t) and dist(Q, u) = dist(x, u) hold. Then, x is a possible intermediate stop to go from P to Q in exactly 2 teleportations, so, if the inequalities hold, 2 is the answer. Notice there are other cases in which 2 is also the answer, which are covered below.

At this point, we can assume that either P is closer to any teleporter than Q, or vice versa (otherwise, we can choose two teleporters to fullfill the inequalities at the beginning of the previous paragraph). Since the problem is symmetric, swap P and Q if needed to make P the closest of P and Q to all teleporters.

Now recall the definitions of R, L and U from the Small solution. Since P is closest to all teleporters, dist(Q, t) > $U_{t,1}$ = dist(P, t) for all t. This means Q is outside the spheres centered in

all teleporters. Since $L_{t,i}$ is non-increasing with i, the inner sphere contracts with each step, which means Q is never inside the inner sphere, so as soon as Q is inside the outer sphere, we are guaranteed that Q is reachable. So, we only need to calculate the Us. By reading its definition above, we note that $U_{t,i}$ is equal to the longest path from P to t using teleporters as intermediate steps, where the length of each step is simply the distance between the two points.

We can calculate the length of the required longest paths for all t and a fixed i in $O(\mathbf{N}^3 \log i)$ time by using something similar to iterated squaring to calculate the matrix of largest distances from any teleporter to any other in i - 1 steps, and then combining that with the vector of distances from P to each teleporter. The "multiplication" here is not an actual matrix times matrix multiplication, but rather the use of the property that the longest path from t to u in i steps is the longest path from t to v in j steps plus the longest path from v to u in k - j steps, for some v. Taking j = k / 2 for even k shows how to do log k steps overall. This, combined with a binary search on the number of steps, gives an algorithm with overall time complexity $O(\mathbf{N}^3 \log^2 M)$. If you have a good implementation in a fast language, this runs in minutes, but it's enough to pass the Large.

It's possible to get rid of one the log factors for an overall time complexity of $O(\mathbf{N}^3 \log M)$, and a program that finishes the Large in a few seconds. This is achieved by starting the binary search on a range [min, max) whose size, that is, max - min, is a power of 2. Each step calculates mid as the averge of min and max, so mid - min and max - mid are also powers of 2, which proves by induction that the range size is a power of 2 in every step of the binary search. Then, since mid - min is also a power of 2 in every step, every distance matrix you need is of a number of steps that is itself a power of 2 (the range keeps being cut in half, so it remains of size a power of 2, so the delta between the min and the midpoint that we need to test is always a power of 2). That means we can precalculate all needed matrices in $O(\mathbf{N}^3 \log M)$ time, since the matrix for 2^{k+1} steps is the "square" of the matrix for 2^k steps. With the memoized matrices, each step of the binary search only takes $O(\mathbf{N}^2)$ time to "multiply" the matrix and the initial vector.