Problem Editorials

November 12, 2017

Forbidden Zero

A simple approach for solving this problem is:

```
n += 1
while n contains a zero:
  n += 1
print n
```

For many integers, this will run very quickly—in particular, for any integer that does not end in 9, we will never enter the while loop. In the worst case, n takes on the value 999999, in which case we have to manually loop over 111112 integers until we find one we can print out. This is a relatively small number of integers to loop over, so the above algorithm will be fine.

Note that determining whether a number contains a zero is most easily done by casting the number to a string and searching for the character zero, rather than attempting to perform arithmetic on the number.

Odd Palindrome

We present two algorithms—one that is easier to prove correct and one that is easier to code.

The most direct way to tackle the problem is to consider all even-length substrings and check if any of those are palindromes. There are $O(|s|^2)$ such substrings and it takes $O(|s|)$ to check if such a string is a palindrome, so this algorithm runs in $O(|s|^3)$ and passes in time.

The above algorithm is easy to prove correct because it directly checks the condition about whether the string is odd. The approach that is easier to code requires making the following observation about odd strings—a string is odd if and only if it has no palindromic substrings of length 2.

To see that this condition is equivalent to the original condition, assume that the string s has a palindrome of length $2n$. Remove the first $n-1$ and last $n-1$ characters of the substring—this leaves a palindrome of length 2 that is a substring of the original string. All strings which are not odd therefore have a palindromic substring of length 2.

To check if a string has a palindromic substring of length 2, it suffices to check all adjacent pairs of characters in the substring to see if any of them are equal. This algorithm runs in $O(|s|)$.

Unloaded Die

Because there are only six sides to the die, we can try to erase the number on every side and compute the correct number to put on the erased side, and then save the best one.

Let's say that we're erasing the 1, and we want to replace it with x_1. We have that $v_1 x_1 + 2v_2 + 3v_3 + 4v_4 + 5v_5 + 6v_6 = 3.5$. Therefore,

$$x_1 = \frac{3.5 - 2v_2 - 3v_3 - 4v_4 - 5v_5 - 6v_6}{v_1}.$$

By similar logic, we can compute the exact value of x_k if we erase k and want to replace it with x_k to make it so that the value is fair.

One thing to be careful about is division by zero. However, erasing a value that has probability zero of coming up will never affect the value, and we can always erase a value k and replace it with itself if the die is already fair.

Here's a puzzle about this problem: what's the greatest value that can be printed?

Star Arrangements

Because S is small, we can iterate over all possible counts of stars in the first row from 2 to $S-1$. There are two cases to check:

- The first row and the second row have the same number of stars. If the first row has r stars, then we simply check if S is divisible by r.
- The first row has r stars and the second row has $r-1$ stars. There are two possibilities here: either the last row has r stars or the last row has $r-1$ stars. In the first case, you want $(S-r)$ to be divisible by $2r-1$. In the second case, you want S to be divisible by $2r-1$.

```
for r from 2 to S-1:
  if the first row can have r stars and the second can have r-1 stars:
    print (r, r-1)
  if both the first and second rows can have r stars:
    print (r, r)
```

Halfway

Because n is large, we will not be able to directly simulate the number of comparisons for a given integer i and then keep incrementing i until we hit the halfway point.

What we can do, however, is perform a binary search for the point at which we are halfway done.

Imagine that we have an integer k that we have just printed out and we want to determine if we have already done enough comparisons. Note that the number of comparisons done for the integers from $k+1$ to n is $(n-k)(n-k-1)/2$, so the number of comparisons that have been performed thus far is $(n(n-1)-(n-k)(n-k-1))/2$. We can binary search for when this value exceeds $(n+1)/2$, which is the halfway point.

We can do it even faster than this with an analytic solution: the halfway point will be near $(1-\sqrt{1/2})n$. Either we can do the math carefully and determine a precise closed-form answer, or we can search around this point for the correct answer.

Note that because n is large, we must be careful about precision and overflow. In particular, if we ever want to reference the number of comparisons done, we should remember to use 64-bit integers, and if we want to return an analytic answer by leveraging square root, we should be careful since it's hard to predict how that value being close to a rounding boundary can interact with the final answer; this is why we recommend searching around the point returned from the calculation in order to verify it.

Purple Rain

We translate this problem as follows: we have an array where every element is either 1 or -1. We wish to compute the subarray where the absolute value of the sum of the elements is maximal.

To do this, we calculate a running sum of the values from the first to the last. We remember the earliest point this running sum hit its maximum, and the earliest point this running sum hit its minimum. The answer is the range between these two points.

Delayed Work

There are two approaches here to efficiently get the answer.

One uses calculus. Let $f(n)$ be the cost given that n painters are hired. We note that $f(n) = Xn+(KP)/n$. Note that f is decreasing and then increasing, as its derivative $f'(n) = X-(KP)/n^2$ is negative and then positive. We can compute where $f'(n)$ is zero and then check the two values around it.

Another way of doing the above but without calculus is using ternary search to compute the minimum cost.

Because of the above observation and the low bounds, it is also possible to iterate on the number of workers from 1 to infinity, stopping as soon as the cost starts increasing.

Latin Squares

This problem has two parts: figuring out if the given array is a Latin Square, and then figuring out if it is reduced.

To check if an array is a Latin Square, we want all the elements in every row and every column to be distinct. Because the given array is small, for any given row or column, we can loop over all pairs of elements and check for matches. A faster solution would be to store a set of characters and check if there are any duplicates inserted into the set. A third solution is to copy and sort the array and ensure there are no duplicates.

Given that the array is a Latin Square, the last check is to figure out if it is in reduced form. To do that, we can generate the natural order of the digits and then make sure that the first row and first column exactly match the natural order.

Security Badge

We start by solving a simpler problem: given a specific ID, is it possible to get from room 1 to room n? This problem can be solved with BFS in $O(n + m)$ time—rooms are vertices and doors are edges—only doors that admit the given ID can be used.

If we applied this algorithm directly for all IDs from 1 to k, we would have an $O((n + m)k)$ algorithm, which is far too slow because k is very large.

If we consider a very small set of rooms and doors and a large ID space, we would note that a lot of IDs are essentially equivalent since they can be used to pass through exactly the same sets of doors. In particular, the only time when IDs i and $i + 1$ do not necessarily give access to the same doors is if some door admits i and not $i + 1$, or some door admits $i + 1$ but not i.

If we go back to the algorithm that iterates on IDs from 1 to k and instead think about the sets of doors that can be accessed, the only IDs we need to consider are IDs where a door first becomes accessible, or when a door is first no longer accessible. These IDs are therefore all possible values of c_i and $d_i + 1$.

We can do the above BFS for a given value v and see if it is possible to reach room n. If so, then all IDs from v up to, but not including, the next highest value in the IDs worth considering, are valid.

This reduces the complexity of our algorithm from $O((n+m)k)$ to $O(mn+m^2)$, which will run in time.

Fear Factoring

Sometimes to solve problems like this it is easiest to break it down. One way to calculate the sum of factors of a single number is trial division:

```
for (int f = 1; f <= p; f++)
   if (p % f == 0)
      s += f;
```

This can be sped up by realizing that factors (other than the square root, for a square) come in pairs:

```
for (int f = 1; f*f <= p; f++)
   if (p % f == 0) {
      s += f;
      if (f*f != p)
         s += p/f;
   }
```

Here we separate factors into two groups, small factors (those less than or equal to the square root of p) and large factors (those larger than the square root of p).

The previous code tests factors for divisibility. Rather than simply check for a single number if it is divisible by a given factor, we can ask how many numbers in a range are divisible by that factor. For the range $1, \ldots, n$, the number of positive numbers divisible by a factor f is just $\lfloor n/f \rfloor$; each one contributes f to the final sum. If m is the largest small factor, then for the range $a \ldots b$ we can sum the small factors with

```
int m = sqrt(b);
for (int f = 1; f <= m; f++)
   s += f * (b/f - (a-1)/f)
```

For instance, for the range $7, \ldots, 10$ and for the factor 2, there are two numbers divisible by 2: 8, and 10, and each contributes 2 to the sum from the factor 2.

Each large factor (those larger than m) is paired with a small factor f. The multiples of f less than or equal to b have complementary factors of $1, 2, \ldots, \lfloor b/f \rfloor$. The sum of a sequence $1 \ldots k$ is just $k(k+1)/2$, the so-called triangle numbers. Factors less than or equal to m are already counted, so the final code looks like:

```
int m = sqrt(b);
for (int f = 1; f <= m; f++)
   s += f * (b/f - (a-1)/f) + triangle(max(m, (a-1)/f), max(m, b/f))
```

where `triangle`(q, r) calculates the sum of integers from $q + 1$ to r.

Straight Shot

Imagine that we point the toy robot directly at $(X, 0)$ and let it walk until it reached the line $x = X$. If it ends above $(X, 0)$, then we know we have to turn the robot clockwise into the 4th quadrant. If it ends below $(X, 0)$, then we know we have to turn the robot counterclockwise into the 1st quadrant.

This observation lets us use bisection to find the correct angle to send the toy robot in to arrive directly at $(X, 0)$. (Bisection is the continuous analogue to binary search; we terminate the search when the width of the search window falls below an appropriate epsilon.) Given that we set it off at an angle of θ, then the amount of time it takes to get there is $X/(v \cos\theta)$, and we can limit our search to the range where $\cos\theta \leq 1/2$, or $-\pi/3 \leq \theta \leq \pi/3$ (in practice, we widen the window slightly to guard against precision/rounding issues).

An even easier solution is to just calculate the average vertical speed of the robot on its journey, since the proportion of time we will spend on each sidewalk is simply the width of that sidewalk divided by X. If we know the average vertical speed, and the average total speed, we can use the Pythagorean theorem to calculate the average horizontal speed and from this the total time required for transit.

Unsatisfying

The given formula is a 2-SAT formula. There's a well-known method to turn a 2-SAT problem into a graph so graph theory techniques can be used.

Boolean logic tells us that any disjunction $a \vee b$ is equivalent to the implication $\neg a \to b$ and also the equivalent implication $\neg b \to a$. Thus, if we create a graph node for both the positive and negative forms of a proposition, we can turn each disjunction into a pair of edges on that graph. Then, a chain of implications is made equivalent to a path on that graph.

A 2-SAT instance is unsatisfiable if and only if there is a path from the true version of the variable to the false variable and vice versa. Both paths are required.

For the clauses that we can add, we can only add clauses with two positive variables. This corresponds to adding directed edges, one from a negative variable to a positive one.

For each variable, we'll find the minimum cost to make this happen.

For each node, we can find the set of nodes that are reachable from it. Let this set be $f(x)$. In addition, for each node, we can find the set of nodes that can reach it. Let this set be $r(x)$.

If $\neg x$ is in $f(x)$ and x is in $f(\neg x)$, then we're done, we don't need additional variables.

Otherwise, if we can check the intersection of $f(x)$ and $f(\neg x)$ to see if there is any negative variable, and the intersection of $r(x)$ and $r(\neg x)$ to see if there is any positive variable. If so, we can do this with one additional clause.

Otherwise, we check if $f(x)$ has a negative variable, $f(\neg x)$ has a negative variable, $r(x)$ has a positive variable, and $r(\neg x)$ has a positive variable. If these are all true, then we can do it with two additional clauses.

Otherwise, it is impossible.

We can take the minimum over all variables as our answer.

Distinct Distances

For two given points a, b, our new point q will be the same distance away from a, b if and only if q lies on the perpendicular bisector of a and b.

So, let's consider all $\binom{n}{2}$ pairs of points and their perpendicular bisectors. For every pair of perpendicular bisectors, we compute their intersection and then calculate the set of distances from this point to all other points.

In addition, we also need to consider the midpoint between every pair of points as a candidate because all the bisectors might be parallel.

This algorithm is $O(n^5)$ or $O(n^5 \log n)$ depending on how you compute the size of the set of distances, but with the low bounds this is fast enough.

One might be tempted to try to use floating point to calculate the distances, but comparing floating point values for equality can be very tricky due to round-off error. The coordinate limits in this problem were chosen carefully so exact rational arithmetic can be used. Indeed, since the only non-integral value is the candidate point being considered, we can easily multiply all values by the denominator of the candidate point under consideration and do all arithmetic with integers. With this approach careful consideration must be given to overflow.

Long Long Strings

Given two DNA editing programs, we would like to determine if they are equivalent. One way of approaching this is taking a program and reducing it into a canonical form, and then checking if two programs produce the same canonical form. This canonical form can be defined in a number of different ways; one way is to ensure all deletes come after all inserts, and that the inserts are in a particular order, and the deletes are in a particular order. Creating this canonical form entails sorting the given program to satisfy the constraints of that canonical form.

To sort, we can use a bubble sort. We need to ensure we know how to exchange any two commands in the sort order, preserving the semantics by updating the indices. In addition, we need to support the cancellation of an insert by a delete. There are quite a few cases to consider, and a single mistake can cause our program to misbehave.

A simpler solution is to just simulate the effect of each program using a data structure such as a rope or an interval list. If implemented carefully, this can easily run in time.

Spinning Up

If we think about making a palindrome out of the string, the first place to start would be the outermost characters. We can decide on how we want to arrange the outermost characters and then recursively handle the inner substring. The only caveat here is that if handling the inner substring forces a carry to happen on the most significant digit, that can affect the answer that is generated.

We can handle this by doing a DP with two functions. For string S, let $f(S)$ be the minimum number of steps needed to convert S into a palindrome where the leading digit does not force a carry on the digit above it, and let $g(S)$ be the minimum number of steps needed to convert S into a palindrome where the leading digit does force a carry to the digit above it.

The transitions for f and g are mutually dependent. Imagine for example that $S = 1s2$, where s is some other string. $f(S)$ is therefore less than or equal to $1 + f(s)$, but it is also less than or equal to $g(s)$.

There are $O(k)$ substrings to analyze here and there are 20 transitions to consider based on the surrounding digit and whether to transition to f or to g. In practice, at most 4 of these transitions are relevant. This gives an $O(k)$ algorithm.

Rainbow Roads

Consider a node v which has two edges incident to it that are the same color. Let these neighbors be $u_1, u_2, \ldots, u_k$. Then, we can safely eliminate all nodes in the subtree defined by the directed

edge from v to u_1.

Doing this naively will take quadratic time. One way we can see quadratic time is if we mark subtrees using search multiple times. To prevent this, we can mark each direction of each edge has having already been explored; if we encounter a marked direction on an edge in a subsequent exploration, we know we don't have to further explore that subtree. It is important to mark each direction on an edge separately.

Another way we can see quadratic behavior is if we have a star-like graph; enumerating the adjacent nodes from a central node many times can slow us down. To prevent this we can keep track of how many times we have traversed a particular node; if we see a node more than twice, we don't need to explore it again. Once is not sufficient as the edge we entered a node from is important. But twice will always suffice.

A third issue is determining when we have seen an edge color more than once from a particular node. If this is not done carefully it also can introduce a quadratic factor in our runtime.

Grid Coloring

If we look at any given column inside a valid grid, it must have k blue squares on the top followed by $m-k$ red squares on the bottom. Furthermore, given the constraint that all squares to the left of a blue square must be blue, if we enumerate the values of k for each column, those values must be non-increasing as we go from left to right across the columns.

This enables us to run a column-by-column DP on the number of blue squares at the top of a given column.

More formally, let $f(i, c)$ be the number of valid grids using just the first c columns where the last column has exactly i blue squares on the top.

To check if $f(i, c)$ is even a feasible state, we require that the highest red square is strictly below the ith square in the column, and that there are no blue squares below the ith square in the column. Finally, we can transfer from $f(j, c-1)$ to $f(i, c)$ if and only if $j \geq i$.

This DP algorithm can be implemented in $O(mn)$ with prefix sums, but due to the low bounds an $O(m^2 n)$ implementation would also pass.

Enlarging Enthusiasm

Let's consider an $O(n!)$ solution. First, for every order, we are interested in whether or not this is achievable.

For each person in order, we can check if we have enough points to assign to them to beat the current leader, and assign the smallest number of points so they can. Then, we can check if the total number of points assigned is at most x (if it is less than x, we can always give excess points to the first place person in the final rankings).

To speed this up, let's make the following transformations. Since the q_i's must be announced in increasing order, we can instead think about the judges choosing $t_1, t_2, \ldots, t_n$ to the singers, where $q_i = t_1 + \ldots + t_i$, and $n \cdot t_1 + (n-1) \cdot t_2 + \ldots + 1 \cdot t_n \leq x$. Now, for a given permutation $\sigma_1, \sigma_2, \ldots, \sigma_n$, we know that $t_i \geq \max\{0, p_{\sigma_{i+1}} + 1 - p_{\sigma_i}\}$. So, to check if a given permutation is possible, we can set t_i to be as low as possible, and check our cost doesn't exceed x.

Thus, we've simplified this to a DP state of current singers already announced, last singer announced, and points left to assign. We can transition by choosing the next singer, and making sure we have enough points (i.e., $(n-i) \cdot t_i$).

Avoiding Airports

Let's describe an $O(m^2)$ solution.

First, we sort the flights by arrival time and relabel them so that

$$e_1 < e_2 < \cdots < e_m.$$

Define $f(k)$ as the smallest frustration that we can possibly have incurred before getting on flight k. For each flight in order, we can check all previous flights that have could have landed before, and take the min over all cases. In mathematical notation,

$$f(k) = \min_{\substack{i<k \\ b_i=a_k}} \left\{ f(i) + (s_k - e_i)^2 \right\}.$$

Of course this is too slow. We'll show how to speed this up to linear time (after sorting).

We rewrite the righthand side as

$$f(k) = \min_{\substack{i<k \\ b_i=a_k}} \left\{ f(i) + s_k^2 - 2s_k e_i + e_i^2 \right\} = s_k^2 + \min_{\substack{i<k \\ b_i=a_k}} \left\{ -2e_i s_k + f(i) + e_i^2 \right\}.$$

Let $A_i = -2e_i$ and $B_i = f(i) + e_i^2$, and imagine lines on the plane given by $y = A_i x + B_i$. With this formulation, the righthand side of the equation above can be interpreted as finding the minimum value attained by these lines, at $x = s_k$.

This problem can be solved efficiently with a method known as the "convex hull trick". In essence, we can keep track of the lower envelope of all the lines deque, where intersection between the lines goes from left to right, and the slopes are decreasing from left to right. Then, when we do a query, we can pop off lines form the front of our deque. You can see the code for more details.