Problem Tutorial: "Abnormal Words"

First we should convert `a` into 0, `b` into 1, ... , `z` into 25. This can be done on every letter in the string with a for loop. Then we can add or subtract the shift and compute the number modulo 26. Afterwards, we can convert each number back into the corresponding letter of the alphabet. Take care when treating C/Java chars as integers: they are signed, so 'z' + 25 is actually a negative number!

Problem Tutorial: "Balanced Fighters"

We can simulate the fight between the fighters. Either both fighters deal no damage to one another, which we can easily check, or one of the fighters deals at least one damage. This means that the fight will last at most 10 000 rounds.

Alternatively, for every duel between A and B, we can calculate the number of rounds for A to get B's HP to zero and B to get A's HP to zero. Whichever number is smaller is the result of the duel; if the numbers are equal, it's a draw. We need to be careful not to divide by 0 here.

Afterwards, we can try all triplets to check if they are intransitive triples in $O(N^3)$. If we choose to simulate the fighters, we need to first precompute the result of all $O(N^2)$ possible duels between pairs of fighters before we check for triples or else we would TLE.

Problem Tutorial: "Unjob Search"

Note that querying `? a b c` is equivalent to asking the following question.

"If we root the tree at `a`, is `b` an ancestor of `c`?".

We can root the tree arbitrarily, say at node 1, and pick an arbitrary other city u to be a candidate for the terminal city. Then we can see if any other city v is a descendant of u. If we find no descendants, we would be done. Otherwise, if we find that v is a descendant of u, we can have v be our next terminal city candidate and continue. Note that cities that aren't descendants of u can't be descendants of v, so we can use the information from the previous queries.

This way, we can find a terminal city, and have a sequence of $N - 2$ queries to verify that it is a terminal city, 2 fewer than necessary.

Problem Tutorial: "Astrodirections"

Notice that at any point in time, the set of planets at which the festival might be held form a contiguous interval. Furthermore, this interval only changes when we visit some planet in this interval, at which point we'll be directly adjacent to the new interval.

This observation suggests dynamic programming with about $2N$ states: the size of the remaining interval, and one bit denoting which side of the interval we're on. However, we may need to take some intermediate jumps well outside of the interval.

Luckily, some pre-processing enables us to compute the minimum cost of a series of jumps that achieves any given total displacement, leading up to our next landing inside the interval under consideration. The constraint that we must stay between 1 and N appears troublesome. Fortunately, because $J \le N/2$, it can be shown that any sequence of jumps can be reordered such that we never move more than $N/2$ away from our starting point and always within the bounds of the planets.

The pre-processing can be accomplished with Dijkstra's algorithm in $O(J^2)$ by scanning through a distance array and visited array for minimum distance vertex. Alternatively, we can pre-process by noticing that each sequence of at most 2^i jumps can be composed of a pair of at most 2^{i-1} jumps. We can scan through the array $\log J$ times and compute all combination of pairs of jumps to reach each point. This takes $O(J^2 \log J)$.

Finally, the dynamic programming itself takes $O(NJ)$ time because we consider J possible total jumps from each of $O(N)$ states.

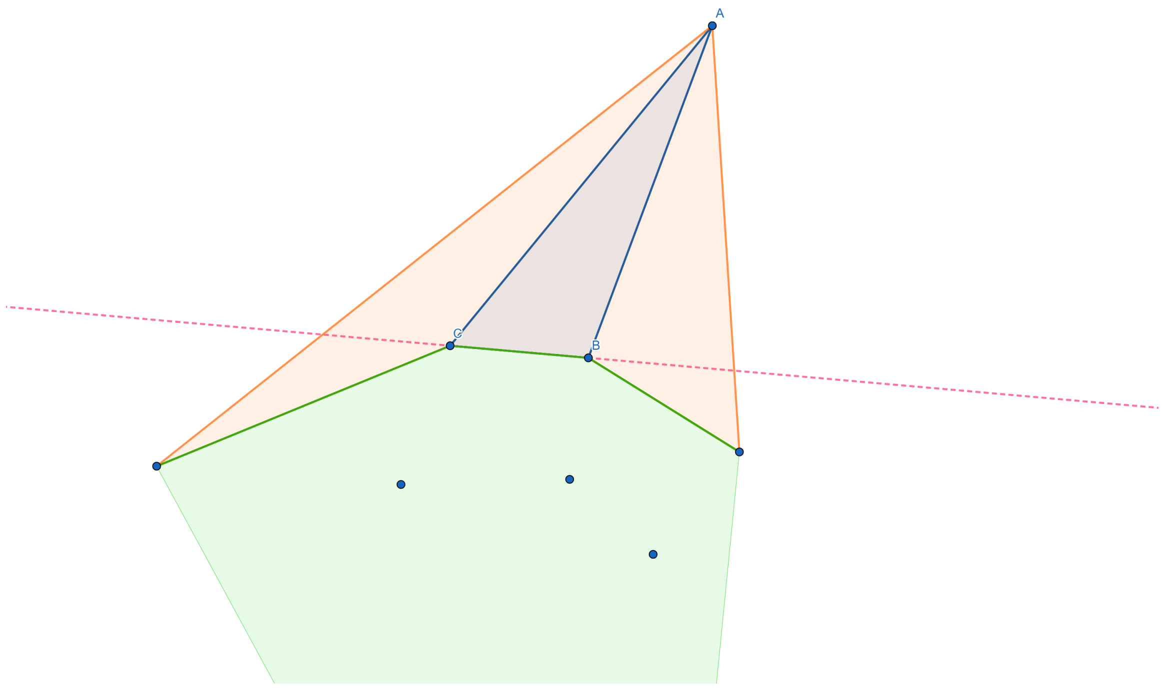
Problem Tutorial: “Exciting Acts”

It is always possible to partition the acts into K parts such that the K most exciting scenes are each in different acts. Hence we can output the sum of the K largest elements by sorting the list in $O(N \log N)$.

Using a counting sort we can solve this problem in $O(\max(a_i))$, or we can use quick select to solve the problem in expected $O(N)$ time.

Alternatively, divide and conquer DP can be employed to solve the problem in $O(NK \log N)$ and while it will pass, the solution is both slower and more complicated. $O(NK)$ DP is also possible.

Problem Tutorial: “Fair Distribution”



Let’s first solve the problem with no three points collinear. Consider any triangle of three points A, B, and C. This triangle contributes to A’s share iff A is chosen after both B and C, but before all other points on A’s side of BC. Let $v_1, v_2, \ldots, v_k$ be the other points on A’s side of BC. The valid permutations of $A, B, C, v_1, \ldots, v_k$ are precisely those which start with either B, C, A or C, B, A. Hence, there are $2 \cdot k!$ valid permutations out of a total of $(k+3)!$. The probability of ABC’s contribution is therefore $\frac{2 \cdot k!}{(k+3)!} = \frac{2}{(k+1)(k+2)(k+3)}$.
We can sum the triangles areas weighted by this probability to get our desired answer.

If we do this naively with collinear points, we may miss or double-count some areas due to ambiguity in assigning equivalent areas. There are multiple ways to canonically credit the area to some set of triangles. One such way is to forbid any points within the line segment BC when calculating the contribution of triangle ABC to A, by treating those points as on the “same” side as A, and all other points as on the opposite side. This effectively decomposes the large triangle into smaller triangles. Alternatively, we can forbid any outside points so we sum over the largest triangle.

We can fix B and C, count the number of points on each side, then update all points on either side with the area weighted by the probability. This solution would be $O(N^3)$ and is fast enough.

Alternatively, you can solve this with an angular sweep around every point B, and use the linearity of the area calculation to solve this in $O(N^2 \log N)$ or even $O(N^2)$ with a topological line sweep in the dual space.

Another reasonable-seeming approach is to slightly perturb all the points, so as to eliminate collinearity without appreciably changing the answer. However, this method requires higher precision than 64 bits.

This problem was inspired by this paper on Shapley values in the Plane by Timothy Chan which only considers the case when points are in general position: `https://arxiv.org/abs/1804.03894`.

Problem Tutorial: "Infinity Plus One"

Some easy facts:

- P(T) = T if, and only if, P contains no +'s. In this case, we say P is zero. Proof: by induction.
- For all programs P and Q with Q non-zero, P < PQ. Proof: $\mathrm{P}(T) \subsetneq \mathrm{Q}(\mathrm{P}(T)) = (\mathrm{PQ})(T)$.
- If we increase any subprogram in P, the resulting P' satisfies P ≤ P'. Proof: by induction, noting that the definitions are monotonic in subprograms.

The key result is that all counting programs are equivalent to a canonical form, which consists of a non-increasing sequence of "powers of infinity". For example: [[[+]]][[+]][+][+][+]+++

We'll prove this by induction on the context-free language. But first, we need one more lemma: if P is any power of infinity and Q is a larger power of infinity, then PQ = Q.

Proof: WLOG, increase P so that its power is only one less than that of Q. Then Q = [P], and:

$(\mathrm{PQ})(T) = (\mathrm{P[P]})(T) = [\mathrm{P}](\mathrm{P}(T)) = \mathrm{P}(\mathrm{P}(T)) \cup \mathrm{P}(\mathrm{P}(\mathrm{P}(T))) \cup ... = [\mathrm{P}](T) = \mathrm{Q}(T)$.

Now we do the induction. The base case is trivial. The concatenation case follows from the lemma, which lets us erase any powers that violate the non-increasing constraint.

All that remains is the infinite loop case [P]. If P is zero, then so is [P]. Otherwise, the canonical form of P has n copies of the leading term ∞^d and possibly some smaller powers. By the lemma, the k'th finite repetition of P is at least $kn\infty^d$ and at most $(kn+1)\infty^d$. Therefore, the union of all such finite repetitions equals the union of all finite repetitions of the leading term alone, and so [P] reduces to ∞^{d+1}.

Problem Tutorial: "Ancient Wisdom"

The answer for David's age is the product of all prime numbers with an odd power in C. So, simply factor out primes from C less than $\sqrt[3]{2^{63}} = 2^{21} < 3 \cdot 10^6$. This will ensure that either C is prime, or the product of at most two large primes. From here, we can check whether what remains is a square or not. If it is, we can ignore it, otherwise, we can write it is the product of two primes that appear exactly once, so we can just multiply our solution so far by it. This runs in $O(C^{1/3})$.

It is also possible to use more complicated $O(C^{1/4})$ factoring techniques, but that isn't necessary.