Problem Tutorial: "Fraction"

Group #1. ($n \leqslant 100$) For this group it's possible to look over all possible values of the numerator $1 \leqslant a \leqslant \lfloor \frac{n}{2} \rfloor$. For the current value of a we have to compute denominator as $b = n - a$, check that there are no common divisors (except 1) of a and b (it also could be done by a linear search of possible common divisors $2 \leqslant d \leqslant a$). The answer is the maximal fraction that was considered during the iteration over all possible values of the numerator and passed the irreducibility test.

Pseudocode of the described solution is presented below:

Set $a_{best} = 1$, $b_{best} = n - 1$;
for $a = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$ **do**
    Set $b = n - a$;
    Set *coprime* = `True`;
    for $d = 2, 3, \ldots, a$ **do**
        if a is divided by d and b is divided by d **then**
            Set *coprime* = `False`;
        end if
    end for
    if coprime and $\frac{a}{b} > \frac{a_{best}}{b_{best}}$ **then**
        Set $a_{best} = a$, $b_{best} = b$;
    end if
end for
Output a_{best}, b_{best};

This solution have $\mathcal{O}(n^2)$ time complexity.

Solutions for other groups use the following idea. It is easy to see that maximal fraction $\frac{a}{b}$ such that $a + b = n$ is $\frac{\lfloor \frac{n}{2} \rfloor}{\lceil \frac{n}{2} \rceil}$. Moreove, the closer numbers a and b are, the higher the value $\frac{a}{b}$ is. So, we can look over possible numerator values strating from the value of $\lfloor \frac{n}{2} \rfloor$ and decrease it by one at each step. We can stop our search on the first encountered value of a such that number a and $b = n - a$ are coprime.

This is the common part of solutions for groups 2 and 3:

Set $a = \lfloor \frac{n}{2} \rfloor$, $b = n - a$;
while Not `Coprime`(a, b) **do**
    Set $a = a - 1$, $b = b + 1$;
end while
Output a_{best}, b_{best};

To complete this algorithm we need to define the `Coprime` function that checks that numbers a and b have no common divisor except 1. It will be shown below that the "while"loop is guaranteed to stop after no more that 2 iterations. So, time complexity of the described solutions is determined by the time complexity of the `Coprime` function.

Group #2. ($n \leqslant 100\,000$) For this group we can use following implementation of the `Coprime` function:

function Coprime(a, b)
    for $d = 2, 3, \ldots, a$ **do**
        if a is divided by d and b is divided by d **then**
            Return `False`;
        end if
    end for
    Return `True`;
end function

The idea behind this pseudocode is again the linear search of common divisors of a and b. This implementation of the function have time complexity of $\mathcal{O}(n)$ and the solution that use this implementation have the same complexity.

Group #3. ($n \leqslant 10^9$) For this group we have to use enhanced implementation of the `Coprime` function. Let's use the following idea. To find common divisors of a and b we can look over all divisors of a and for each divisor d check whether d is also divisor of b or not. We can find all divisors of a in a efficient procedure with $\mathcal{O}(\sqrt{(n)})$ time complexity:

```
1: function Coprime(a, b)
2:     Set d = 2;
3:     while d * d ⩽ a do
4:         if a is divided by d then
5:             Set D = a/d;
6:             if b is divided by d or b is divided by D then
7:                 Return False;
8:             end if
9:         end if
10:        Set d = d + 1;
11:    end while
12:    Return True;
13: end function
```

This procedure is based on the fact that for each number d that is divisor of a there is a number $D = \frac{a}{d}$ that is also divisor of a. Since $d \times D = a$, one of these numbers is not grater than $\sqrt{a}$. So, we can only look over possible divisors of a that is not grater than $\sqrt{a}$ and given a divisor $d \leqslant \sqrt{a}$ we can compute paired divisor as $D = \frac{a}{d}$.

Group #4. ($n \leqslant 10^{18}$) For this group we can check that a and b using greatest common divisor (`gcd`). a and b are coprime if and only if $\texttt{gcd}(a, b) = 1$. If we use Euclidean algorithm to compute `gcd` then we get full solution for the problem.

However, we can find answer to the problem analytically in the following way:

```
1: if n is odd then
2:     a = ⌊n/2⌋, b = ⌈n/2⌉;
3: else
4:     if n is not divided by 4 then
5:         a = ⌊n/2⌋ − 1, b = ⌈n/2⌉ + 1;
6:     else
7:         a = ⌊n/2⌋ − 2, b = ⌈n/2⌉ + 2
8:     end if
9: end if
10: Output a, b.
```

Problem Tutorial: "Jury Meeting"

Group #1. Obviously, each member of the jury needs to buy exactly two tickets — to the capital and back. Go through all possible variants of assigning forward and return flights to jury members and fix availability period for each person. Then intersect them and check if their common availability period is at least k days long. Find minimum cost among these variants or find out that the problem has no solution. Expected complexity: any reasonable exponential complexity.

Group #2. Sort all flights by the day of departure. Now go through flights to the capital (forward flights) and one by one assume it is the last forward flight in answer (let's say it is scheduled on day d). Thus you are assuming that all forward flights are contained in some fixed prefix of flights. Make sure that there is at least one forward flight for every jury member in this prefix and find the cheapest forward flight among them for every person. All return flights we are interested in are contained in the suffix of flights list such that every flight's departure date is at least $d + k + 1$. Take similar steps: make sure that there is at least one return flight for every jury member in this suffix and find the cheapest return flight among them for every person as well. Select minimal cost among these variants or find out that the problem has

no solution. Expected complexity: $O(nm^2)$ or $O(m^2 + n)$.

Group #3. Just as the boundary of considered prefix moves right, the boundary of considered suffix moves right as well. This suggests that the problem could be solved by the two pointers method. Assume you are storing minimum forward flight's cost on current prefix (infinity if no flight exists) for every person, and you are storing multiset (ordered by cost) of all return flights on current suffix for each person as well. To proceed next prefix and conforming suffix do the following:

- Move prefix boundary to the next forward flight. If its cost c_i is less than minimum forward flight's cost fwd_{f_i} from that city, then you could improve total cost: decrease it by $c_i - fwd_{f_i}$ and set fwd_{f_i} to c_i since it's new minimal cost.
- Move suffix boundary to the next backward flight until there is such flight exists and it departure date difference with prefix boundary departure date is under $k + 1$.
- While moving suffix boundary, keep return flights multisets consistent: remove boundary flight right before moving that boundary to the next flight. Also check out if you are removing cheapest flight from multiset. If it is so, minimal flight cost for that city changed as well as total cost: it increases by difference between old minimal cost and new minimal cost. Keep in mind that if you are removing last flight from multiset, then there is no more appropriate return flight for this city and you should terminate the process.

Proceed these steps moving boundaries to the right until the process terminates. In this way you've reviewed the same prefixes and corresponding suffixes as in 50 point solution described above. Select minimal cost among these variants and get your deserved 100 points. Expected complexity: $O(m \cdot \log m + n)$.

Problem Tutorial: "Michael and Charging Stations"

Group #1. ($n \leqslant 20$) Before solving any subtask one observation is required: suppose at day i we have x_i bonuses. Then exists optimal solution, which spends 0 or $min(a_i, x_i)$ bonuses every day.

It's quite easy to proof: suppose we have some optimal solution and i is a first day, when neither 0 nor $min(a_i, x_i)$ bonuses were spent. If i is a last day on which non-zero amount of bonuses was spent, we can notice that solution spending $min(a_i, x_i)$ bonuses that day is more optimal, so first solution was optimal. So let's consider next day after i, when non-zero amount of bonuses was spent, say j, and amount of bonuses spent at day j is s_j (Also, amount of bonuses spent on day i is s_i). Let's look at solution that spends $s_i + min(s_i - min(a_i, x_i), s_j)$ bonuses at day i and $s_j - min(s_i - min(a_i, x_i), s_j)$. That solution is still correct and still optimal, but it spends $min(a_i, x_i)$ at day i or 0 at day j. Anyway this operation increases first day i when neither i nor $min(a_i, x_i)$ bonuses were spent or first day j after it, when non-zero amount of burles were spent. But we can't increase i or j infinitely, so, after some iterations of such transformation, solution, spending 0 or $min(a_i, x_i)$ bonuses in each day.

That immediately gives us $O(2^n \cdot n)$ solution: just try for each position spend 0 or $min(a_i, x_i)$ bonuses. This solution is expected to score 30 points.

Group #2. ($n \leqslant 1000$) To solve subtask 2 (and score 60 points) it's possible to consider dynamic programming approach: let $dp_{i,j}$ be minimum amount of money that is possible to spend at first i days to pay for all chargings and have $100 \cdot j$ bonuses on card. At first, $dp_{0,0} = 0$ and $dp_{i,j} = \infty$. Then we can easy calculate all states going through all states with something like this code:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= 2 * n; j++) {
        dp[i + 1][j + a[i] / 1000] = min(dp[i + 1][j + a[i] / 1000], dp[i][j] + a[i]);
        dp[i + 1][j - min(j, a[i] / 100)] = min(dp[i + 1][j - min(j, a[i] / 100)],
 dp[i][j] + a[i] - 100 * min(j, a[i] / 100));
    }
```

```
}
```

Of course, j can be up to $2 \cdot n$, because at each day it's possible to earn at most 2 bonuses.

Group #3. ($n \leqslant 3 \cdot 10^5$)

Consider the following observation: there exists an optimal solution, which never has more 3000 bonuses on bonus card. To proof it let's first proof following lemma:

Lemma 1: There exists an optimal solution which spends only 0 or a_i bonuses at day i if there are at least 3000 bonuses at card at the beginning of day i.

Lemma 1 proof: Let's introduce some designations. Let x_i be amount of bonuses at the beginning of day i and s_i be amount of bonuses spent at day i. Also let's call day i "fractional"if $s_i \neq 0$ and $s_i \neq a_i$, and call day i "interesting"if $s_i \neq 0$. Let's proof lemma2 and lemma3 at first:

Lemma 2: Assume $x_i \geq 3000$ and j — next after i interesting day and k — next after j interesting day. Then there exists an optimal solution in which k is not a fractional day or j is not a fractional day.

Lemma2 proof: Suppose is some optimal solution j and k are fractional days. Let's consider a solution spending $s_j + min(s_k, a_j - sj)$ bonuses at day j and $s_k - min(s_k, a_j - sj)$ at day k. This solution is still correct, because $x_i \geq 3000$, so for days j and k there is enough bonuses and still optimal. Lemma2 is proved.

Lemma 3: Assume $x_i \geq 3000$ and j — next after i interesting day. Then there exists an optimal solution is which j is not a fractional day.

Lemma 3 proof: Consider some optimal solution with fractional day j. At first let's proof that j is not last interesting day. Suppose, j is last interesting day in solution. But we can make a solution that spends a_i bonuses at day i (because $a_i \leq 3000$) and it will be more optimal. Contradiction. So there exists next after j interesting day. Let's call it k. Let's consider 2 cases:

Case 1 ($a_j = 1000$): Let's spend consider solution spending 1000 bonuses at day j and $a_k - (1000 - s_j)$) at day k. It's still correct and optimal but j is not a fractional day.

Case 2 ($a_j = 2000$): There are two subcases:

Case 2.1 ($a_k = 2000$): Let's spend consider solution spending 2000 bonuses at day j and $a_k - (2000 - s_j)$) at day k. It's still correct and optimal but j is not a fractional day.

Case 2.2 ($a_k = 1000$): Let's proof, k is not last interesting day. Assume k is last interesting day. Consider a solution spending 2000 bonuses at day j and 1000 bonuses at day k. It's correct but more optimal that initial solution. Conrtadiction. Now let p be next after k interesting day (k is not a fractional day by lemma2). If $2000 - a_j \leq 1000$ we can consider solution which spends 2000 bonuses at day j, $1000 - (2000 - a_k)$ bonuses at day j and s_p bonuses at day p. If $2000 - s_j > 1000$ let's consider a solution which spends $s_j + 1000$ bonuses at day j, 0 bonuses at day k and s_p at day p. But by lemma2 $s_p = a_p$, so we can consider solution that spends 2000 bonuses at day j 0 bonuses at day k and $a_p - (2000 - s_j - a_k)$ at day k. All of these solutions are correct and optimal.

Lemma 1 proof (end): At first, of course there is at least one interesting day after i (Otherwise, it's more optimal to charge at day i using bounses, but in initial solution $s_i = 0$ because $x_{i-1} \leq 3000$ and $x_i > 3000$). Let's call that day j and by lemma3 j is not fractional day. Let's consider 4 cases now:

Case 1: ($a_i = 1000, a_j = 1000$). Let's consider a solution with $s_i = 1000$ and $s_j = 0$. It's correct and still optimal, but $x_i \leq 3000$.

Case 2: ($a_i = 2000, a_j = 2000$). Same as case1.

Case 3: ($a_i = 2000, a_j = 1000$). Let's consider 2 subcases:

Case 3.1: j is not last interesting day. Let k be next interesting day. It $a_k = 1000$ consider a solution spending 2000 bonuses at day i, 0 bonuses at days j and k. It's still correct and optimal, but $x_i \leq 3000$. If $a_k = 2000$ consider a solution a spending 2000 bonuses at day i, 1000 bonuses at day j and 0 bonuses at day k. It's correct and optimal too, and $x_i \leq 3000$ too.

Case 3.2: j is last interesting day. Let's construct solution this way. At first let's set $s_i = 1000$ and $s_j = 0$. Then let's iterate over all ineteresting days after j, say k, in order in increasing time and set $s_i = s_i + min(2000 - s_i, s_k), s_k = s_k - min(2000 - s_i, s_k)$. If after this process we still have some bonus left just add it to s_i. At the end, s_i will be equal 2000 because we spent all bonuses, solution will still be correct and optimal, but $x_i \leq 3000$.

Case 4: $(a_i = 1000, a_j = 2000)$. Let p be last day before i with $s_p \neq 0$. If $a_p = 1000$ consider a solution with $s_p = 0, s_i = 0, s_j = 2000$. It's correct, optimal and $x_t \leq 3000$ for each $t \leq i$. If $a_p = 2000$, consider a solution with $s_p = 2000, s_i = 0, s_j = 0$. It's correct, optimal and $x_t \leq 3000$ for each $t \leq i$, too.

So for all cases we can make correct and optimal solution such there is no $x_i \leq 3000$ for all i, or number of first day with $x_i > 3000$ increases, but it can't increase forever, so after some amount of opereations solution with $x_i \leq 3000$ for all i will be constructed.

Because of this fact we can consider dynamic programming approach described before but notice, that we should consider only states with $j \leq 30$. It will have $O(n)$ complexity. Moreover, looking at states with $j = 30$ is required. It's possible to make a test on which solution, that looks at states with $j \leq 29$ will be incorrect.

Problem Tutorial: "Lada Malina"

The key part of a solution is to understand what are the locations that may be accessed from the origin in T seconds. First observation is that we should investigate it only in case when $T = 1$ because T is simply a scale factor. Let's denote this set for $T = 1$ as P.

Group #1. For the first group it's easy to see that P is a square with vertices in points $(\pm 2, 0), (0, \pm 2)$. So, the first group may be solved with a straightforward $O(qn)$ approach: we iterate through all the factories and check if it's possible to get for cars from i-th factories to the car exposition. We can rotate the plane by 45 degrees (this may be done by the transformation $x' = x + y, y' = x - y$), after this each query region looks like a square. Therefore, it's necessary to check if point lies inside a square:

$$\begin{cases} -2T \leq (px_i - fx_j) + (py_i - fy_j) \leq 2T \\ -2T \leq (px_i - fx_j) - (py_i - fy_j) \leq 2T \end{cases}$$

Group #2. In the second group propeller velocities are two arbitrary vectors. It can be shown that P will always be a parallelogram centered in the origin, built on vectors $2v_1$ and $2v_2$ as sides. Thus, this group is a matter of the same $O(qn)$ approach with a bit more complicated predicate: one should be able to check that an integer point belongs to an integer parallelogram. The key observation is that we may find an appropriate transformation of a plane that transforms this set into a rectangle. Indeed, there always exists an affine transformation performing what we want. As an additional requirement, we want to transform coordinates in such way that they are still integral and not much larger than the original coordinates. The transformation looks like following:

$$\begin{cases} x' = vx_1 \cdot y - vy_1 \cdot x \\ y' = vx_2 \cdot y - vy_2 \cdot x \end{cases}$$

The first expression is a signed distance to the line parallel to the vector v_1, and the second one is the signed distance from the line parallel to the vector v_2. Easy to see that belonging to some query parallelogram can be formulated in terms of x' and y' independendly belonging to some ranges.

Group #3. Second group should be a hint for the third group. One can find that the set $P = w_1v_1 + \ldots + w_kv_k || w_i | \leq 1$ is always a central-symmetric polygon with the center in the origin. Actually, this Polygon is a Minkowski sum of k

segments $[-v_i, v_i]$. Minkowski sum of sets $A_1, A_2, \ldots, A_k$ is by definition the following set: $A_1 + A_2 + \ldots + A_k = \{a_1 + a_2 + \ldots + a_k | a_1 \in A_1, a_2 \in A_2, \ldots, a_k \in A_k\}$. It can be built in $O(k \log k)$ time, although in this problem k is very small, so one may use any inefficient approach that comes into his head, like building a convex hull of all points $\pm v_1 \pm v_2 \ldots \pm v_k$.

After we found out a form of P, it's possible to solve the third group of tests in $O(qnk)$ by checking if each possible factory location belongs into a query polygon in $O(k)$ time.

Following groups are exactly the same, but the constraints are higher, they require using some geometric data structure to deal with range queries.

Groups #4 and #5. Fourth group and fifth group are very similar to first and second group correspondingly, but we need to process the requests faster. After the transformation of the plane, the request can be reformulated as "find sum of all factories inside a square so any 2d data structure may be applied, like a segment tree of segment trees. Another approach is to use a sweeping line algorithm with a segment tree or an appropriate binary search tree, achieving a time complexity $O((q+n)\log^2 n)$ or $O((q+n)\log n)$.

Group #6. To solve the sixth group we need to use a trapezoidal polygon area calculation algorithm applied to our problem. Calculate the sum of points in each of $2k$ trapezoid areas below each of the sides of a polygon, and then take them with appropriate signs to achieve a result. Such trapezoid area can be seen as a set of points satisfying the inequalities $l \le x \le r$ and $y \le kx + b$. Under transformation $x' = x, y' = y - kx$, this area becomes a rectangle, leading us to an $O((q+n)k \log n)$ time solution.