Problem Tutorial: "First Mission"

To print the string in C++, one can write:

```
cout << ''May the judge be with you\n'';
```

To print the string in Java, one can write:

```
System.out.println(''May the judge be with you'');
```

To print the string in Python, one can write:

```
print ''May the judge be with you'';
```

Problem Tutorial: "Basketball"

In this problem the best scenario is that the opposing team scores baskets of the lowest value and your team scores baskets of the higher value. So let $x = \min(p, q)$, and $y = \max(p, q)$. The droids will end up with $n + bx$ points. You want to find the minimum k such that:

$$m + ky > n + bx.$$

Solving, we get

$$k > \frac{n + bx - m}{y}$$

Since k must be an integer, we find that the answer is $k = \lfloor \frac{n+bx-m}{y} \rfloor + 1$ or 0 if $n + bx < m$ (Herman's team doesn't need to score any baskets).

Problem Tutorial: "Unfair War"

Observe that since Herman has the cards 1 through n and knows which cards Jar Jar will play, if Jar Jar plays a card with value x, Herman can just play his card with value $x + 1$ and win. However, if Jar Jar plays the highest card (value n), Herman cannot possibly win, so he should just play the card with value 1. Consequently, the answer is m if Jar Jar does not play the highest card and $m - 1$ if Jar Jar does play the highest card.

Problem Tutorial: "Smallest Substring"

In this problem note that we cannot test every single contiguous subarray and take the shortest one containing k ones. This is because there are $\binom{n}{2}$ subarrays which means such an algorithm will take at least $O(n^2)$ time, which is too slow.

Instead let's look at the where the original array A contains 1's and store the indices (in order) in a list - call the list L. More formally our list L will have the property $L[1] < L[2] < \cdots < L[m]$, where $A[L[i]] = 1$ for all $1 \leq i \leq m$, where m is the size of the list (the number of 1's in the original array).

Then we know that the subarray $A[L[i]...L[i + x]]$ will contain $x + 1$ 1's by construction. And the length of that subarray will be $L[i + x] - L[i] + 1$.

We are interested in finding the length of the shortest subarray that contains k 1's, or more formally,

$$\min_{i=1}^{m-k+1} L[i + k - 1] - L[i] + 1.$$

We can find this minimum using a single loop that takes $O(m) = O(n)$ time. Populating the list L also can be done with a single loop, so it can be done in $O(n)$ time. The overall runtime is $O(n)$.

Problem Tutorial: "Maximum Product"

First, we can sort the array in nondecreasing order of the numbers. Note that to maximize the product, it is best to choose the four largest numbers (possibly 4 positives), the four smallest numbers (possibly 4 negatives), or the two largest numbers with the two smallest numbers (possibly 2 positives and 2 negatives). Hence, after sorting, we can just check these three cases and find the maximum product. The overall runtime is $O(n \log n)$ due to sorting.

Problem Tutorial: "Human Pyramid"

Consider the greedy approach in which we place the lightest person at the top, the next two lightest people at the level right below, etc. It can be shown by induction that this construction satisfies the conditions.

Hence, if l is the number of levels in the pyramid, we just need to find the largest l such that $\binom{l+1}{2} \leq n$. This can be found by trying all possibilities of l starting from 1. Note that $\frac{l(l+1)}{2} \leq n \implies l^2 < l(l+1) \leq 2n$, so $l < \sqrt{2n}$. It follows that we check at most $\sqrt{2n}$ values, so the overall time complexity is $O(\sqrt{n})$.

(Using some additional math, we can also determine the maximum l in $O(1)$.)

Problem Tutorial: "Cookies"

Let's say we pick a cookies from Unkar Plutt's Junkyard. Given that cookies from the same shop cost the same and we are definitely going to buy a cookies from this shop, it is optimal to choose the a cookies with the highest happiness values.

After buying a cookies from Unkar Plutt's Junkyard, we have $c_1 = \max(0, c - ax)$ dollars left. Thus, we can buy $b = \min\left(n, \left\lfloor \frac{c_1}{y} \right\rfloor\right)$ cookies from Jabba's Cookie Hut. Again, we should choose the b cookies with the highest happiness values.

As a result, given the value for a, we can easily calculate b in $O(1)$. Now we just need to find the sum of the highest a and b happiness values for the cookies in the two shops. However, this can be precomputed by sorting the cookies in descending order of happiness values and then using an array of prefix sums. Hence, given a, computing the total happiness value can also be done in $O(1)$.

It follows that we can now just loop through all possible values of a from 0 to m inclusive and take the maximum of all happiness values that result. The overall time complexity is $O(n \log n)$ due to sorting.

(Alternatively, one can keep track of a running prefix sum and a pointer to the cookies from Jabba's Cookie Hut instead of precomputing the whole prefix sum array. This works because as we increase a, b can only decrease or stay the same, so the pointer only moves in one direction.)

Problem Tutorial: "GCD"

For this problem assume $a > b$. If in the input $a < b$, then we just swap them, and if $a = b$ then the answer is just a.

Note that for any $0 \leq i < b$, $\gcd(a - i, b - i)$ is a factor of both $a - i$ and $b - i$. This means we can write $\gcd(a - i, b - i) \cdot k_1 = a - i$ and $\gcd(a - i, b - i) \cdot k_2 = b - i$ for some k_1, k_2.

Subtracting, we get

$$\begin{aligned} & \gcd(a - i, b - i) \cdot k_1 - \gcd(a - i, b - i) \cdot k_2 = (a - i) - (b - i) \\ \implies & \gcd(a - i, b - i)(k_1 - k_2) = a - b \end{aligned}$$

This means that for all $0 \leq i < b$, $\gcd(a - i, b - i)$ is a factor of $a - b$.

So the answer must also be a factor of $a - b$, so let's loop over the factors of $a - b$.

If we have a factor d where $d > b$, then clearly there is no such i where $\gcd(a - i, b - i) = d$, since i must be non-negative so $\gcd(a - i, b - i) \leq b$. So d cannot be our answer in this case.

Otherwise we have a factor d where $d \leq b$. In this case we can always find an i such that $\gcd(a-i, b-i) = d$, namely $b - d$.

Also note that the divisors occur in pairs with the one less than or equal to $\sqrt{a-b}$, so the overall runtime is $O(\sqrt{a-b})$.

Problem Tutorial: "Racing"

First, we cannot find the time in which every pair of ships pass each other and take the last time, since that would take $O(n^2)$, which is too slow.

Note that when no more ships pass each other, it means that the ships are sorted in order of velocity (the ship that is the farthest ahead is the fastest, the ship right behind it is the second fastest, etc.).

This means the last time a ship passes another must occur between ships that are adjacent when ordered by velocity. Note that there are only $n - 1$ of these adjacent pairs.

So we sort the ships by velocity which takes $O(n \log n)$, and then we find the times that the $n-1$ adjacent pairs of ships pass each other, and we take the last time, which takes $O(n)$, so our overall algorithm takes $O(n \log n)$.

Note the answer is -1 when the ships sorted by position in the input is the same ordering as if the ships were sorted by velocity.

Problem Tutorial: "Painting"

We can try reversing the steps that Lingxiao took to find a possible way of painting the board.

Since we know that Lingxiao only paints a row or column at a time, then when the painting is finished there has to be an entire row or column that is the same color. We can then try "unpainting" that row or column.

When we unpaint, we change all the cells in that row or column to a special color. This special color effectively serves as a wildcard color - it can be any color we want it to be since it could be any color before Lingxiao painted over it.

So we continuously unpaint rows and columns that don't have are missing either red or blue (either they consist of only red and the wildcard color or blue and the wildcard color).

Eventually, the entire board will be wildcard color, at which point we are done.

We can keep track of the number of reds, blues, and wildcards in each row and column. So for each iteration, we find a row or column that we can unpaint (missing red or blue) and has not already been unpainted. This takes $O(n + m)$. Then we actually unpaint the row or columns, updating the number of reds, blues, and wildcards in the rows and columns that are affected by the change. This also takes $O(n + m)$. Note that once we unpaint the entire board we are done, so we paint at most $O(n + m)$ times, so the overall algorithm runs in $O((n + m)^2)$.