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: "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: "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: "Minimizing with SQRT"

Note that we only have one operation to apply, so it makes sense to try a greedy approach. We want to always square root the largest element to reduce the sum by the most amount.

Thus, we just need to maintain a data structure that supports efficient insertion and deletion with and lookup of the maximum element, for example a heap (implemented as priority queue in most languages).

So at each step we find the maximum element, remove it, apply a square root, and add it back into the heap. We do this m times, so the overall algorithm takes $O(m \log n)$.

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: "Minimizing with SQRT and CBRT"

Note that there are two different operations we can use, so there is no clear greedy solution like there was in the square root version of this problem.

Instead, we can proceed with dynamic programming. First let us do some precomputation to find $sc(i, j, k)$, which gives the minimum value of a_i after applying j square root operations and k cube root operations. We can derive the following recurrence:

$$sc(i,j,k) = \min(\lfloor\sqrt{sc(i,j-1,k)}\rfloor, \lfloor\sqrt[3]{sc(i,j,k-1)}\rfloor),$$

and $sc(i,0,0) = a_i$ for all $1 \le i \le n$.

Note that in the end we want to minimize the sum of the numbers, so let's define another function $f(i,j,k)$ as the minimum sum of the first i numbers after applying j square roots and k cube roots. We can derive the following recurrence:

$$f(i,j,k) = \min_{u,v} f(i-1, j-u, k-v) + sc(i,u,v),$$

where $f(i,0,0) = \sum_{j=1}^{i} a_j$, for all $1 \le i \le n$.

Note that since $a_i \le 10^9$, the maximum number of square roots or cube roots that can be applied to a number before it becomes 1 is around 7. So this means that we only have to search for all $1 \le u, v \le 7$.

Note, the pre-computation takes $O(nk^2)$, where k is around 7 in this case.

Computing f takes $O(nm_1m_2k^2)$, since there are nm_1m_2 states, and each state takes $O(k^2)$ time to compute.

Overall, the algorithm takes $O(nm_1m_2k^2)$ time.

Problem Tutorial: "Triangle"

Reflect B and P across AC to get B' and P' respectively. Reflect A and P across BC to get A' and P'' respectively. Consider a path from P to Q to R (Figure 1). Note that $PQ + QR + RP = P'Q + QR + RP'' \ge P'P''$ by the triangle inequality. Since $P'P''$ is fixed, the optimal choice for Q and R is the intersection of AC and $P'P''$ and intersection of BC and $P'P''$ respectively. Hence, in this case, the answer is $P'P''$.

However, such a path is only valid if $P'P''$ intersects segments AC and BC. If $P'P''$ does not intersect AC but intersects BC (Figure 2), we have that $PQ + QR + RP = P'Q + QR + RP'' \ge P'A + AR + RP'' \ge P'A + AP''$. The first inequality comes from the fact that Q must lie on AC, and the second follows from the triangle inequality. Thus, the optimal placing for Q is $Q = A$ and the optimal placing for R is the intersection of BC and AP''. In this case, the answer is $P'A + AP''$.

If $P'P''$ does not intersect BC but intersects AC (Figure 3), we have that $PQ + QR + RP = P'Q + QR + RP'' \ge P'Q + QB + BP'' \ge P'B + BP''$. The first inequality comes from the fact that R must lie on BC, and the second follows from the triangle inequality. Thus, the optimal placing for Q is the intersection of AC and $P'B$ and the optimal placing for R is $R = B$. In this case, the answer is $P'B + BP''$.

Finally, if $P'P''$ does not intersect either AC and BC (Figure 4), we have $PQ + QR + RP = P'Q + QR + RP'' \ge P'C + CP''$ because Q must lie on AC and R must lie on BC. The optimal placing here is $Q = R = C$, so the answer is $P'C + CP''$.

Computation for reflections of points and checking whether two segments intersect can be done in $O(1)$, so the overall algorithm is $O(1)$.

Problem Tutorial: "Bits in a Grid"

In this problem, we can try a greedy approach. We place the 1's in the grid row by row. We always want to place the 1's in the columns with the highest column numbers.

So for each row i, we find r_i columns with the largest column numbers and place a 1 in each one of them. Then for each of those columns, we decrement the column number by 1, because we are allowed to place one less 1 in that column in the future. Note that if we ever try to place a 1 in a column that has column number 0, then we know that it is impossible to fill the grid to satisfy the constraints.

The proof that the greedy approach works can be done using induction.

Now we have basically reduced the problem to: given an array (in this case the array of column numbers) and n updates (where in each update i you decrement the r_i largest elements of the array by 1), find if after all the updates there are any negative entries in the array.

There are many ways to solve this problem, and we present one way here. First we sort the array c. It would be nice if for every update we could just decrement the last r_i elements in the array, but note that the order isn't always preserved when doing an update that way.

However, the order is almost preserved. Consider the following example:

Say we have the array: $c[] = [1, 2, 2, 2, 3, 5]$

and we want to decrement the largest 4 values. If we simply decrement the last 4 values, the new array becomes $c[] = [1, 2, 1, 1, 2, 4]$,

which isn't in sorted order. However, if we look at the segment of 2's in the original array from index 1 to 3 (0-indexed), then we see that we decremented the 2's in positions 2 and 3. Instead, we should have decremented the 2's in positions 1 and 2, giving us $c[] = [1, 1, 1, 2, 2, 4]$, which is still ordered.

This motivates the following idea: For each update i, we find the r_i-th largest number in c, which is $c[m - r_i]$. Then we find the segment $[i, j)$ in c that contains only $c[m - r_i]$, such that $c[i] = c[i + 1] = \cdots = c[j - 1] = c[m - r_i]$. In the example above, for the original array, we have $c[m - r_i] = c[6 - 4] = 2$, and $i = 1, j = 4$. We can find i, j in $O(\log m)$ using binary search. Recall that instead of decrementing indices $(m - r_i) \cdots j - 1$ and $j \cdots m$, we want to decrement indices $i \cdots (i + j - (m - r_i))$ and $j \cdots m$ in order to preserve the ordering.

Now it's just a matter of how do we do these range updates efficiently. One way is to use a Binary Indexed Tree to compute the range decrements in $O(\log m)$. But then to access entries in c would take $O(\log m)$ via point queries, giving an overall $O(n \log^2(m))$ algorithm.

Another way is instead of maintaining array c, we imagine we have an array g where $g[i] = c[i] - c[i - 1]$ and $g[0] = c[0]$. Then we store all the nonzero entries in g in an ordered map called *gaps* of (index, value) pairs.

$$c[] = [1, 2, 2, 2, 3, 5]$$
$$g[] = [1, 1, 0, 0, 1, 2]$$
$$gaps = \{(0, 1), (1, 1), (4, 1), (5, 2)\}$$

Figure 1 shows a visualization of the gaps (colored lines) on the example above.

The dashed line corresponds to the range decrement on $i \cdots (i + j - (m - r_i))$. Note that to perform this range decrement, we simply decrease the gap size at i by 1 (removing it if it becomes 0) and increment the gap at $(i + j - (m - r_i))$ (making a new one if there wasn't already a gap there). Note that these operations take $O(\log m)$.

Then we want to perform a range decrement on $j \cdots m$. This is equivalent to decreasing the gap size at j by 1 (removing it if it becomes 0). This also takes $O(\log m)$.

Overall, there are n updates, so this algorithm takes $O(n \log m)$. If at any point we try to decrement values that are already 0, then there is no solution.

Problem Tutorial: "Paths on a Full Binary Tree"

We can use divide and conquer to solve this problem. Let $f(i)$ be the number of paths with cost at most d in the subtree rooted at i. Then, we know that

$$f(i) = f(2i) + f(2i + 1) + x,$$

where x is the number of paths with cost at most d that pass through node i.

To compute x, we store the distances to i of all of the nodes in the subtree rooted at $2i$ in an array $nodes1[]$, as well as the distances to i of all nodes in subtree $2i+1$ in $nodes2[]$.

Then to find x we are essentially asking how many pairs i, j exist such that $nodes1[i] + nodes2[j] \leq d$. This can be done by sorting $nodes1[]$ and $nodes2[]$, and maintaining two pointers in the arrays to count the pairs, which takes $O(k \log k)$, where k is the size of the subtree rooted at i.

The time taken to compute is $T(n) = 2T(n/2) + O(n \log n)$, which gives complexity $O(n \log^2(n))$. We can improve this to $O(n \log n)$ by storing the distance arrays and merging them through a bottom up approach, but $O(n \log^2(n))$ is sufficient.

H. Triangle Diagrams

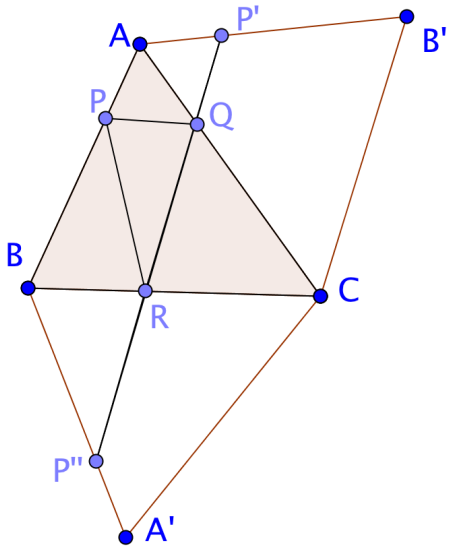

Figure 1

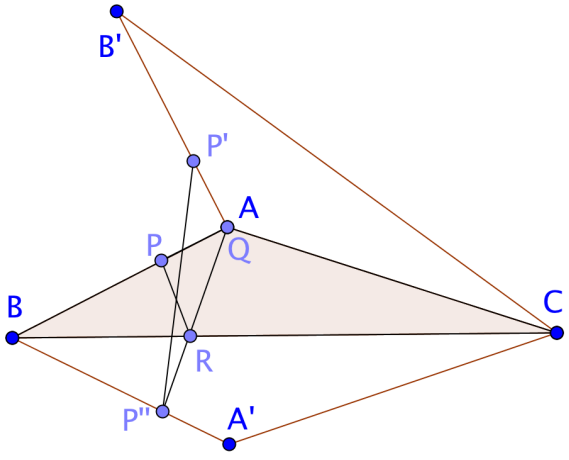

Figure 2

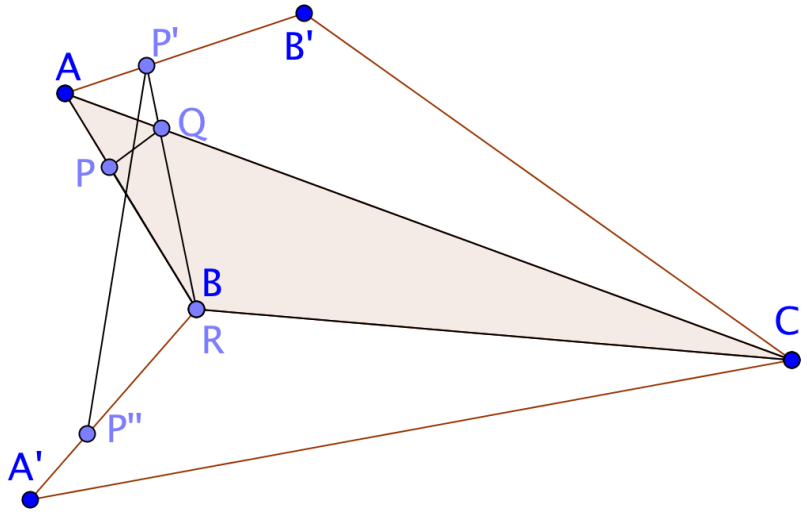

Figure 3

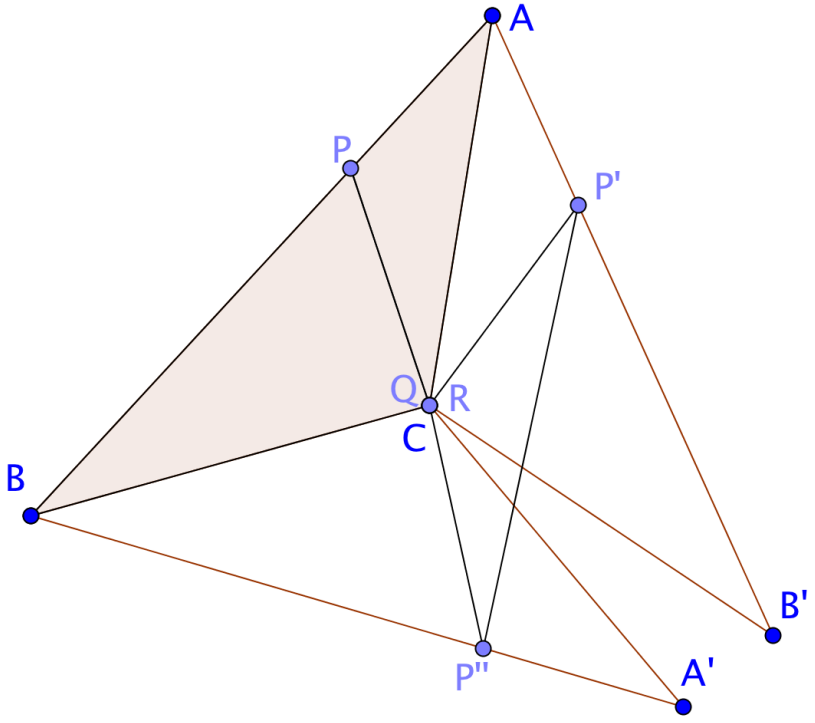

Figure 4

I. Bitgrid Diagram

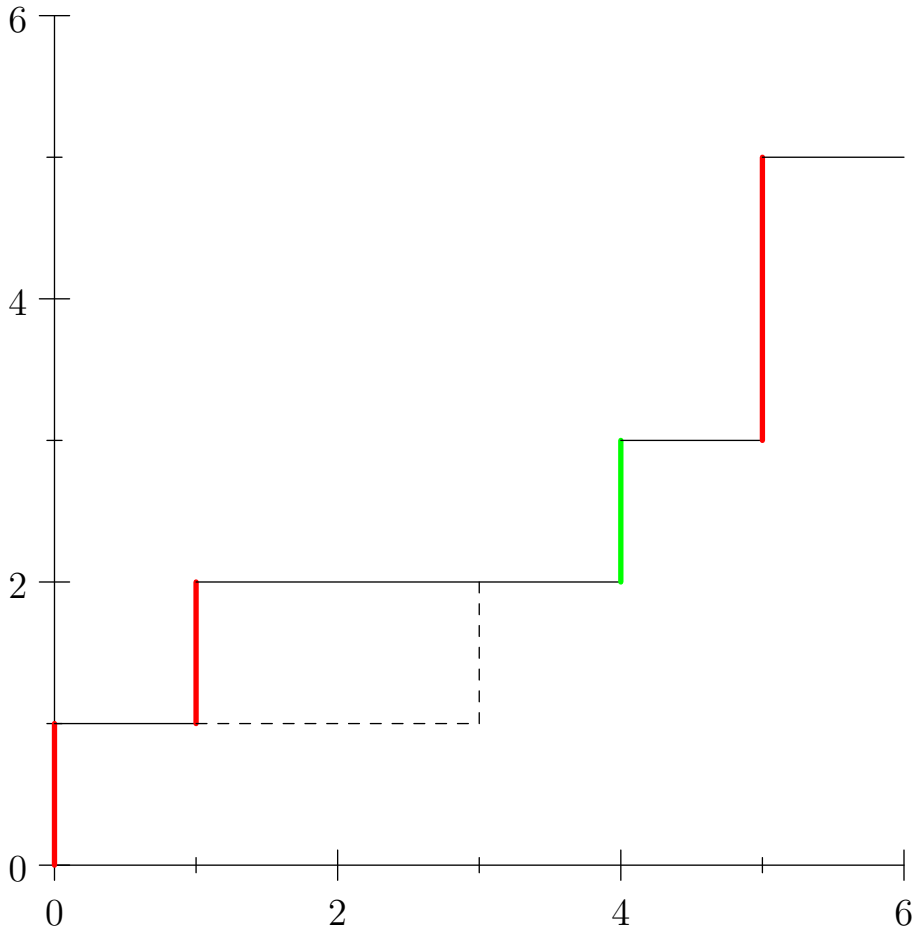

Figure 1