# Precise Gift Shop

Note: This problem had an unknown issue only with Test Case 3 during the contest and was later fixed after the contest.

The tricky part for this problem was to be able to properly handle round off errors with doubles. To avoid this, using `Math.round()` or something else could fix this. Aside from that, the rest of the problem is similar to the Novice problem, `Gift Shop`.

Store all souvenirs' names and total cost ($Q \cdot P$) in arrays and determine the min and max total cost. Loop through the total cost array, and on one line print out the names of the souvenirs whose total cost is equal to the max total cost. Do the same thing for the min total cost on the next line. Make sure to separate the names by a single space (trailing whitespace does not matter on HackerRank).

# Semi-Rare Candies

We can use the dynamic programming approach of Coin Change to solve this problem. Create two arrays: one to store the experience points that the semi-rare candies award and a *dp* array, where $dp[i]$ represents the fewest number of Semi-Rare Candies that are required to meet the experience point value of $i$. Note that the values in this array will be initialized to infinity. We should also keep a sum of the Semi-Rare candies that are collected so we can check if $X$, the experience point threshold, is attainable.

In order to solve this problem, we must build the solution through memoization; we'll use each Semi-Rare candy (represented by $c$) to determine the optimal number of Semi-Rare candies needed to generate a value $i$. We can thus use dynamic programming to find the number of ways to create $X$ (or the maximum number of experience points possible), as outlined below:

$$\text{dp[0]} = 0$$
$$\text{dp[i]} = \min(\text{dp[i]}, \text{dp[i - c]} + 1)$$

We're essentially updating the values in the *dp* array only if the value is attainable by adding the Semi-Rare candy experience point value to a previously-obtained value. Print out $X$ and $dp[X]$, or the maximum experience points possible if $X$ is unattainable.

## Pokémon Regions

There were many ways to solve this. The most common way involves sorting and placing Pokémon in their respective regions. Alternatively, this problem can actually be solved without sorting. Because $N$ is a small enough size for an array, and $P$ is unique, a String array, $arr$, where $arr[p]$ indicates the name of a Pokémon with Pokédex number $p$. When reading in the input, just do $arr[P] = S$. Afterwards, loop through the array starting at $i = 1$. In this loop, if $arr[i] \neq null$ print out the formatted Pokédex number and name, but also check for when to print out the Regions, which is simply four if statements.

## A Wild Pidgey Has Appeared!

The path from the bottom to the top of the field will be completely blocked off if the left and right sides of the field are connected by a series of bushes and buildings. For this, we can use Union-Find to determine whether or not elements are connected through a tree.

First, we need to develop a way to test for the intersection between different obstacles: buildings (rectangles) and bushes (circles). The intersection of two rectangles can be determined with a series of if-statements comparing the position of the rectangle edges. Two circles intersect if the distance between their center points is less than or equal to the sum of their radii. And to determine the intersection of a rectangle and a circle, we must first test which edge of the rectangle lies closest to the circle, and then test for a collision using the Pythagorean Theorem. Elaboration on collision detection can be found here.

With this in mind, we can now connect all obstacles that intersect with each other into a common tree using Union-Find. The left and right sides of the field can be represented as rectangles with height 1000 and width 0. If the left and right sides of the field belong to the same tree, output `No` Otherwise, output `Yes`.

## Guess That Pokémon

Given a name, $p_i$, from the list, if one person knows more names that start with $p_i$, they win, otherwise, neither of them win. The intended solution was to utilize a Trie and on each node (except the root) keep a count variable for both Ash and Brock to count how many times they traversed a certain prefix when inserting their Pokémon's names in the Trie. For every $p_i$ in the list, search the Trie for $p_i$, and if it exists, compare Ash's count and Brock's count to determine the answer, otherwise if it doesn't exist, then neither of them know $p_i$.

During the competition, a solution that involved sorting and binary searching the Pokémon names of Ash and Brock was attempted. It involved obtaining an index $x$ where $p_i$ is a prefix of a name in Ash's and Brock's list of names. It would count how many names started with $p_i$ from the left and from the right after obtaining the index $x$. It was almost successful. The team, unfortunately, did not take into

account that $a_i$ and $b_i$ are not necessarily unique as stated in the Constraints, meaning that Ash or Brock may catch a Pokémon they already have, but that still only counts as knowing one Pokémon's name. Therefore, another tricky part of this problem was to enter Ash's and Brock's Pokémon names in a Set to remove duplicates.

## Macaron Mania

For this problem, because the input has no specific ordering and queries depend on other data being processed first, we have to solve this problem offline

When reading in the queries, store the queries in a custom class with a number $x$ indicating the order in which they came into the input (this will help us store the answer into an array, $ans$, when we determine it). We have to keep track of two things: when people arrive and when people leave. Enter the queries in two arrays, *arriving* and *leaving*, and sort *arriving* based on arrival time (earliest first) then their National Pokédex Number (largest first but could do smaller first leading to an extra calculation later on) if arriving at the same time, $T_i$, and *leaving*, $T_i + W_i$ (earliest first).

Now we have to simulate the situation. First, we need to create an array, $arr$, where $arr[p]$ represents the number of people with a given National Pokédex Number $p$. Next we start looping through the *arriving* array and make sure to remove the people leaving (decrease $arr[P_i]$ by 1) when their time is less than or equal to a person's time arriving. Afterwards, obtain all the people that arrive at the same time, and for each person, update $arr[P_i]$ by 1 and find how many people they can skip which is the sum of the number of people behind them, $\sum_{k=P_i+1}^{10^6} arr[k]$ (the upper limit is $10^6$ because it is the largest time anyone can arrive in the problem). When that sum is obtained, store the answer in $ans$ using the person's $x$ value from when reading the input. Finally print the $ans$ array. This will solve some test cases but not all within the given time limit. To solve all test cases within the time limit, we can optimize this by using something that can perform range sums and updates with better time complexity, such as a Fenwick Tree.

## Pokémon Go To The Polls

This problem becomes easier by first handling guaranteed outcomes. We're safe to assume that if $Y \geq (N+1) \div 2$, then Candidate B has won; they have over half of the votes and have thus achieved a majority. We can also check to see if Candidate A wins if $X \geq (N+1) \div 2$, which evaluates to there being a majority of the total votes being for Candidate A. We check Candidate B first because if candidate B has at least a 50% chance, then A has NO chance of winning, a condition stated in the problem.

In most cases, the scenarios above don't yield an answer. We can use Binomial Coefficients to help us determine the probability that Candidate A wins. The idea here is that there are two outcomes: winning a vote or losing a vote, and binomial coefficients help us account for all $(2^M)$ of the different

voting scenarios the $M$ remaining voters can vote in. Considering that we need $K$ votes to win, let

$$S = \sum_{i=K}^{M} \frac{\binom{M}{i}}{2^M}$$

Candidate A will win if $S > W$, which is simply checking if there are enough "scenarios" (voting configurations where Candidate A gets $K$ votes) such that the ratio is greater than the threshold. If not, then we'll say that there's a tie due to a lack of information; we wouldn't want to be wrong and leave it up to chance!

Note: Binomial Coefficients can be easily calculated with dynamic programming.

## Elimination

To solve this, it requires sorting the 6 different stats separately. Creating a custom Comparable class or some sort of pairing to pair a Pokémon's name with its stat will be needed. Sort the stats in ascending order, and remove the first $K$ Pokémon in each category. A TreeSet of Strings can be used to store the unique names of the eliminated Pokémon in alphabetical order. Iterate over the TreeSet and print out the names. Remember to use `long` because of stats being as large as $10^{10}$.

## Marnie and Gloria

With this problem, we are given a tree. A way to think of the answer is to determine the amount of edges from $A$ to $B$, $E$. If $E$ is even, then both Marnie and Gloria travel $M = G = E \div 2$ routes, otherwise Marnie travels $M = \lceil E \div 2 \rceil$ routes and Gloria travels $G = E - M$ routes since Marnie always travels first.

A simple BFS may be able to solve some test cases, but it will not work in the given time limit for all test cases. To obtain a faster runtime, we can utilize an LCA algorithm that runs in $O(\log n)$. Using LCA, we can determine the amount of edges from $A$ to $B$ in the tree. The amount of edges, $E$, between $A$ and $B$ is simply `Math.abs(depth[L] - depth[A]) + Math.abs(depth[L] - depth[B])` where $L = LCA(A, B)$ and $depth[n]$ is the depth of a node $n$ from the root of the tree.

## Cerulean Waters

We can solve this by determining the rate at all times and storing it. Once we have this we can determine the volume at all times allowing for $O(1)$ queries if we store it.

Let *volume* be an array where $volume[t]$ indicates the amount of water in the pool at a time $t$. Let *rate* be an array where $rate[t]$ indicates the flow rate of the hose at time $t$. Initialize *rate* from index 1 to the end of the array with the value -1 (to indicate that a rate wasn't set, we'll use the number -1). When reading in the flow rate changes, set the rate at time $T_i$ to $R_i$ ($rate[T_i] = R_i$). If a rate at a given time $t$ wasn't set ($rate[t] = -1$) where $t > 0$, then the rate at time $t$ is equal to the rate at time $t - 1$ so $rate[t] = rate[t - 1]$. Based on this we can compute the rate for all times up to $10^6$ (max of $t_i/T_i$) if we start from computing from $t = 1$.

Calculating volume at time $t$ is simply the previous volume at time $t-1$ plus the previous rate at time $t-1$ so $volume[t] = volume[t-1] + rate[t-1]$. With this we can answer queries in constant time, $O(1)$, which will simply be $volume[t_i]$.

## Viridian Dash

Here we are a given a map and are required to find the shortest time starting from a location $(r_s, c_s)$ to the edge of the map. In addition, we have to print all the paths that result in the shortest amount of time. This problem can be seen as similar to Dijkstra's single-source shortest path algorithm where every location (except a location on the edge) in the map is connected to four adjacent locations with a certain weight (time).

For problems like these, we'll need a 2D array, $time$, where $time[r][c]$ is the shortest time to get from a starting location $(r_s,\ c_s)$ to $(r, c)$ in the map. Start with $time[r_s][c_s] = 0$ with every other $time[r][c] = \infty$ (an arbitrary large number) and add it to the Priority Queue. This Priority Queue will be made to prioritize smaller times first. Afterwards, we will traverse the whole map and obtain the shortest times at every $(r, c)$ if applicable.

In order to be able to draw the path, we need to be able to link locations $(r, c)$ to the previous locations $(r_p, c_p)$ it came from. While simultaneously determining the shortest time, we can create these links by referencing the originating location. Afterwards, we can perform a BFS on the locations that reached the edge of the map in the shortest time and draw the path. In order to pass all test cases within the given time limit, Fast IO (BufferedReader/PrintWriter/StringBuilder) must be used as well.

## Berry Buyers

To maximize Team Rocket's profit, they must sell consecutively. This can be done optimally by sorting the days of how long a berry is fresh for and selling one berry each day. The day before a berry is no longer fresh, sell all berries that will no longer be fresh the next day so that way they can still get the most money out of those berries. The resulting answer could be expressed as

$$NM + \sum_{i=1}^{D_{max}} Ai + B$$

where $D_{max}$ is the most consecutive days Team Rocket can sell. With this in mind, a solution can be made if we sort the fresh durations of the berries. From there we can process berries in order to obtain $D_{max}$. Another tricky part of this problem was that BigInteger was required as the profit grows beyond the range of a `long` in Java.