T2611 Gem Collection

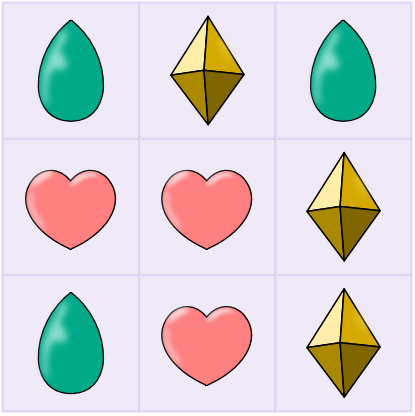
Daniel Hsieh {QwertyPi}

2026-05-23

Table of Contents

Background

Problem Idea and Preparation by `QwertyPi`
Special Thanks to `__declspec`, `happypotato`, `gasbug`, `firewater` for testing!



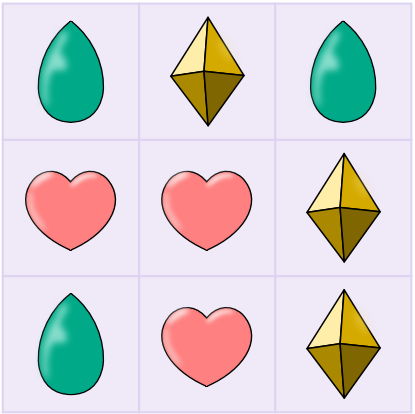
The Problem

Given N gem types and N gems per type.

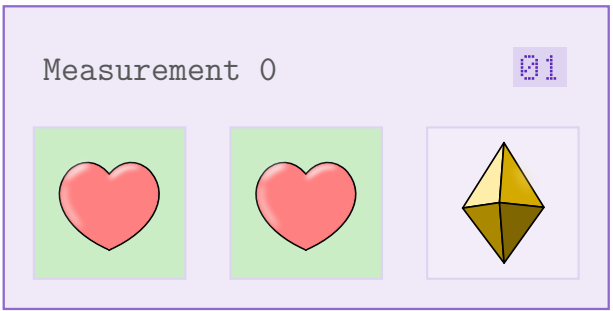
The types of the gems $A[0], A[1], \ldots, A[N^2 - 1]$ are unknown.

Query: Select exactly N distinct gems, return purity = number of gem pairs of the same type.

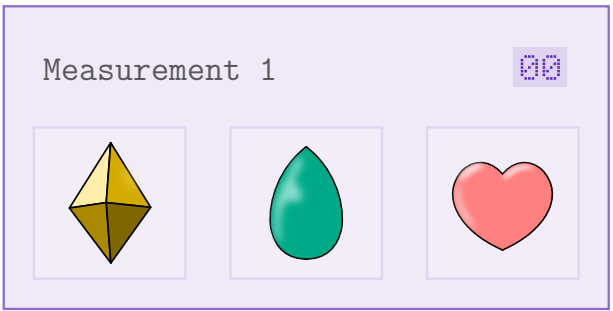
Recover the types of all the gems, using as few queries as possible.



Example

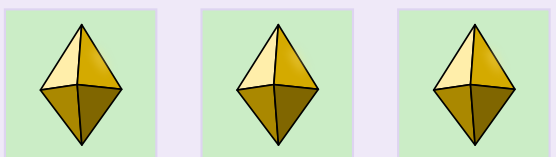


Example



Example

Measurement 2



Subtasks

#	Max	N	Constraints
1	7	≤ 40	$A[i] = i$ for $0 \leq i < N$
2	8		$A[i] = 0$ for $0 \leq i < N$
3	15		
4	46	≤ 100	$A[i] = 0$ for $0 \leq i < N$
5	24		

Query Limit: $Q \leq 128\,000$ ($2N^3$ for $N = 40$, $12.8N^2$ for $N = 100$)

Partial Scoring in Subtask 4 and 5. Full Score: $Q \leq 27\,000$ ($2.7N^2$ for $N = 100$).

Statistics

#	Score	Predict #	Actual #
1	7	60	61
2	8	60	63
3	15	20	25
4	18	10	14
	34	4	2
	46	0	0
5	$(0, 12]$	3	3
	$(12, 24]$	3	0

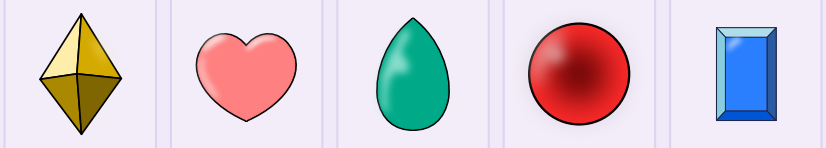
Table of Contents

Rainbow Set

For the ease of description later on, let's introduce some notations!

A *rainbow set* is a set of N gems of all *distinct* types.

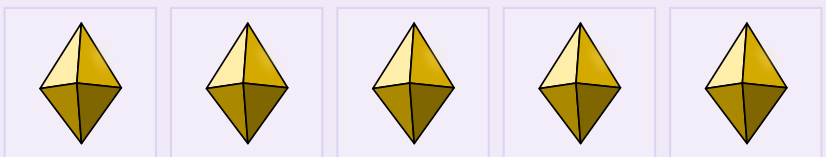
Example - Rainbow Set



Complete Set

A *complete set* is a set of N gems of all *identical* types.

Example - Complete Set



Subtask 1 (7%): $N \leq 40, A[i] = i$ for $0 \leq i < N$

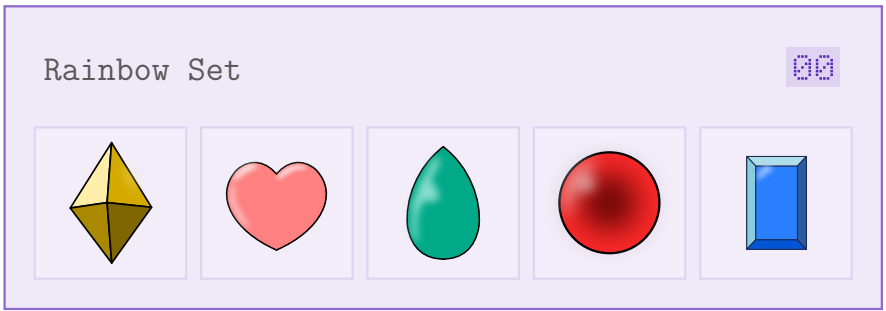
How to distinguish gems using the given *rainbow set*?

Simply query $\{0, 1, \ldots, i-1, i+1, \ldots, N-1, j\}$ to see if $A[j] = i$ for $j \geq N$!

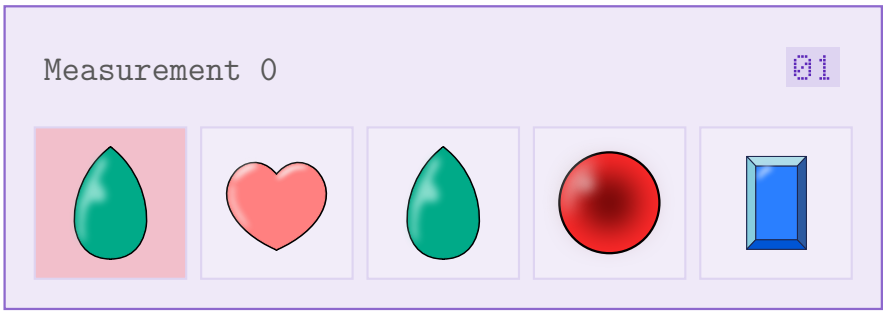
- If $A[j] = i$, then the query result would be 0.
- Otherwise, if $A[j] \neq i$, then the query result would be 1.

We can resolve each unknown gems one by one, each using at most N queries. The number of queries is bounded by N^3.

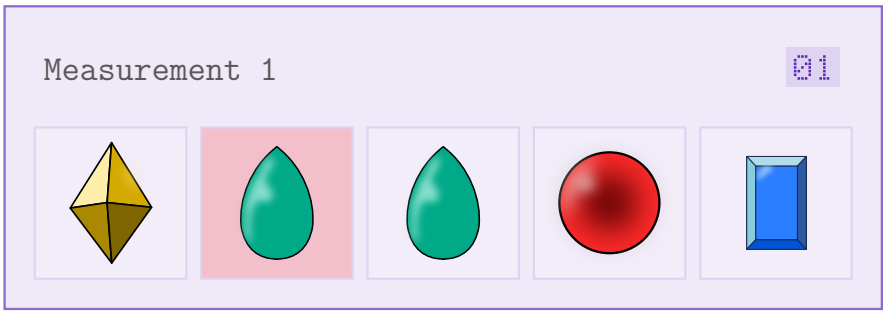
Compare with Rainbow Set



Compare with Rainbow Set

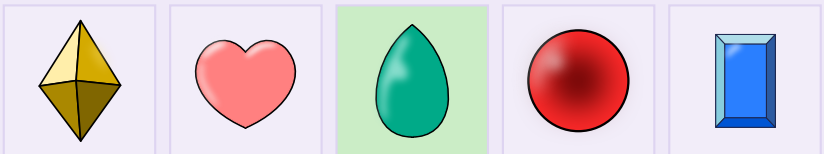


Compare with Rainbow Set



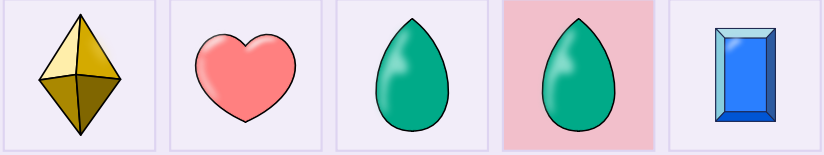
Compare with Rainbow Set

Measurement 2



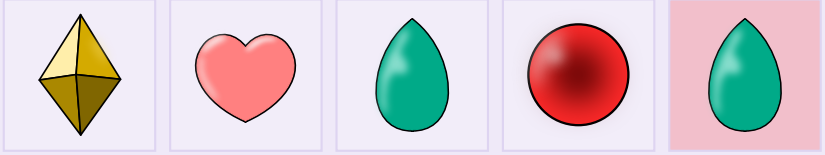
Compare with Rainbow Set

Measurement 3 01



Compare with Rainbow Set

Measurement 4 01



Subtask 2 (8%): $N \leq 40, A[i] = 0$ for $0 \leq i < N$

How to distinguish gems using the given *complete set*?

Suppose we query for $\{0, 1, \ldots, N-3, x, y\}$ where $x, y \geq N$.

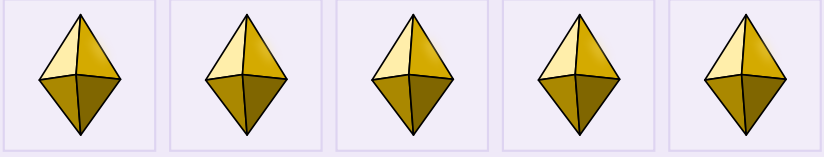
- If $A[x] = A[y]$, the query result would be $\frac{(N-2)(N-3)}{2} + 1$.
- Otherwise, if $A[x] \neq A[y]$, the query result would be $\frac{(N-2)(N-3)}{2}$.

Therefore, we can compare two gems using 1 query.

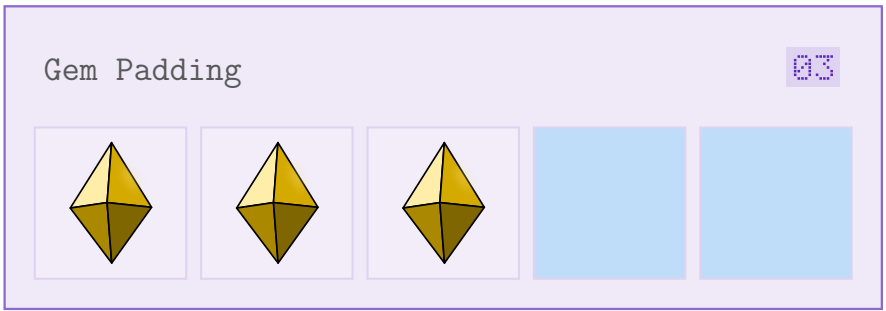
To find a complete set in N^2 queries, simply fix one unknown gem, then compare all others against it. The query count is bounded by N^3 overall.

Compare with Complete Set

Complete Set 10

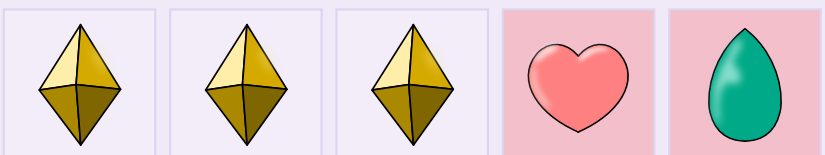


Compare with Complete Set



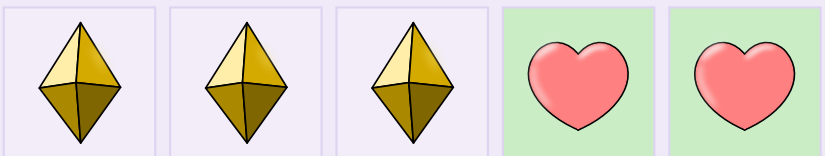
Compare with Complete Set

Case I: Different Gems



Compare with Complete Set

Case II: Equal Gems 04

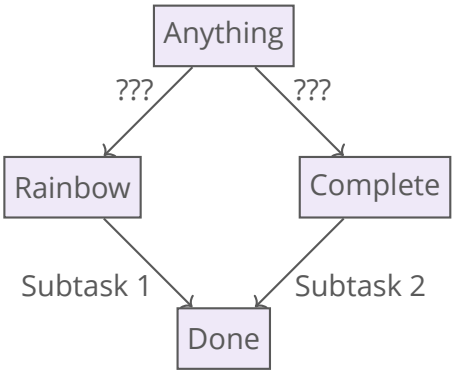


Subtask 3 (15%): $N \leq 40$

Now we don't have any extra information. How should we proceed?

Idea 1: If we find any rainbow set / complete set within N^3 queries, we can reuse subtask 1 / subtask 2 solution.

Idea 2: Rainbow set and complete set are the two extremes, with minimum and maximum query result respectively.

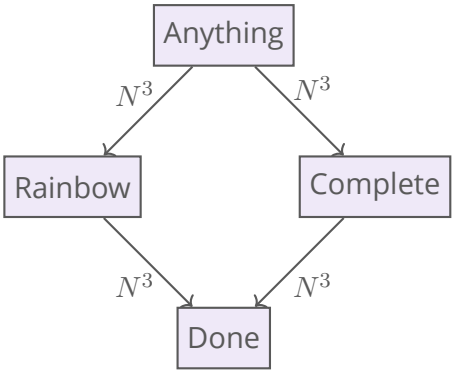


Subtask 3 (15%): $N \leq 40$

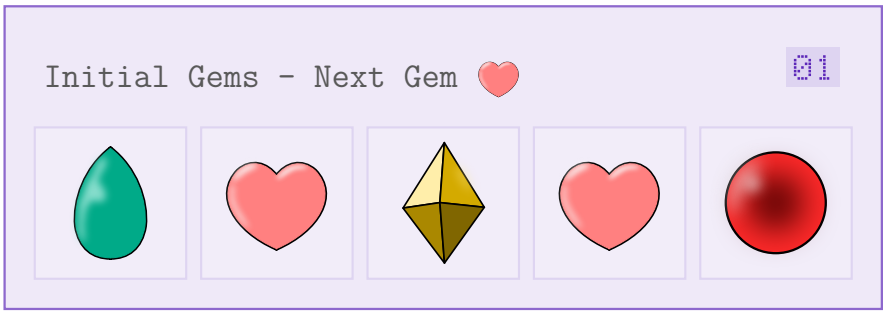
Idea 3: Loop through each gem and substitute each of the N chosen gems, then *greedily* pick the choice that leads to minimum / maximum query result!

Finding any rainbow / complete set costs N^3 queries, so in total the number of queries fits the $2N^3$ limit.

Side Note: Although sharing the same idea, there are *subtle differences* between finding rainbow / complete set. You should try justify your greedy algorithm's correctness.

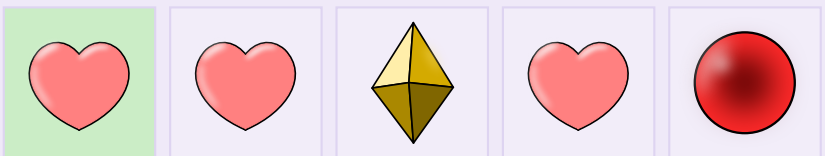


Finding Complete Set



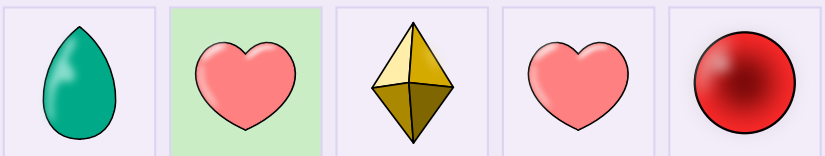
Finding Complete Set

Substitute Gem 0



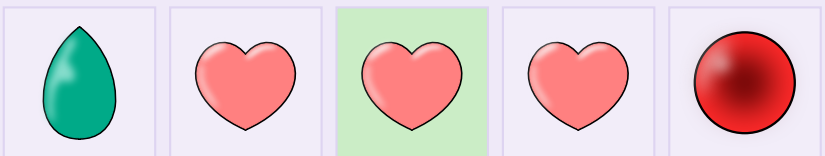
Finding Complete Set

Substitute Gem 1 01



Finding Complete Set

Substitute Gem 2

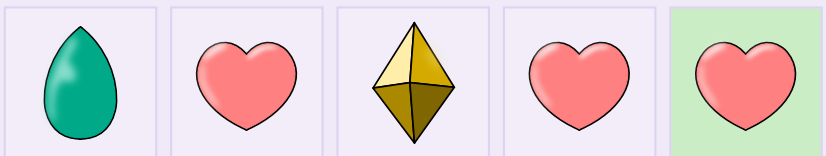


Finding Complete Set

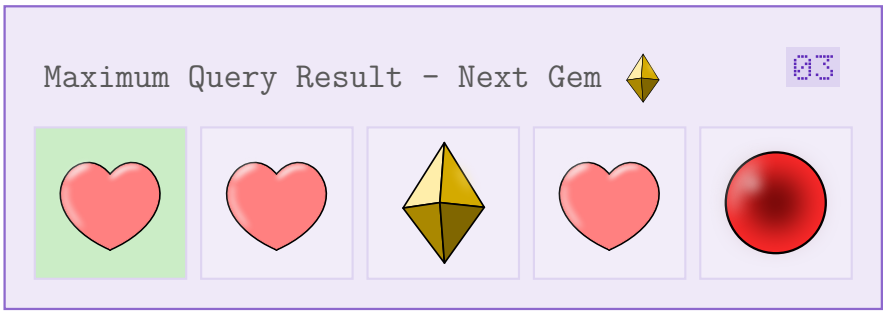
Substitute Gem 3 01

Finding Complete Set

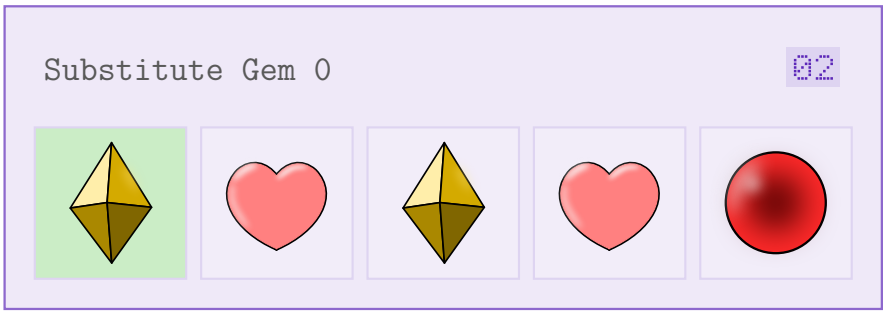
Substitute Gem 4



Finding Complete Set

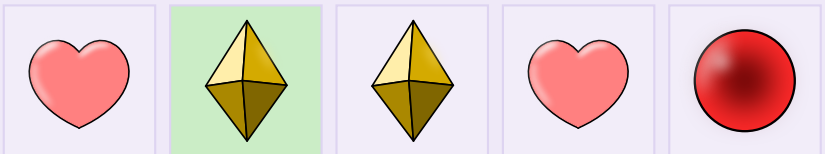


Finding Complete Set



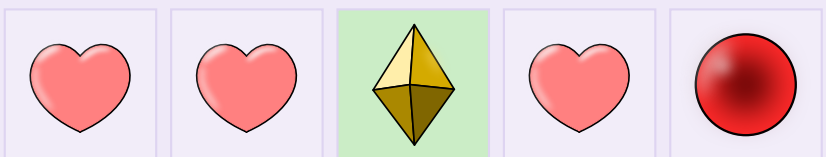
Finding Complete Set

Substitute Gem 1 ⊠2



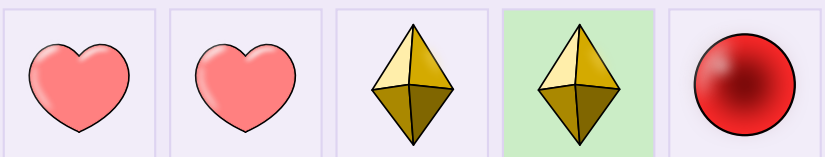
Finding Complete Set

Substitute Gem 2



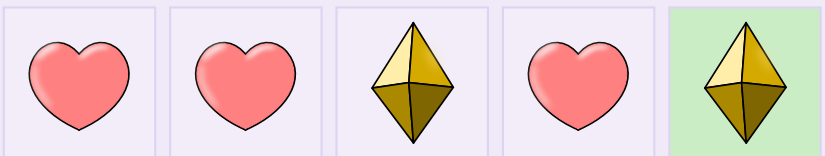
Finding Complete Set

Substitute Gem 3

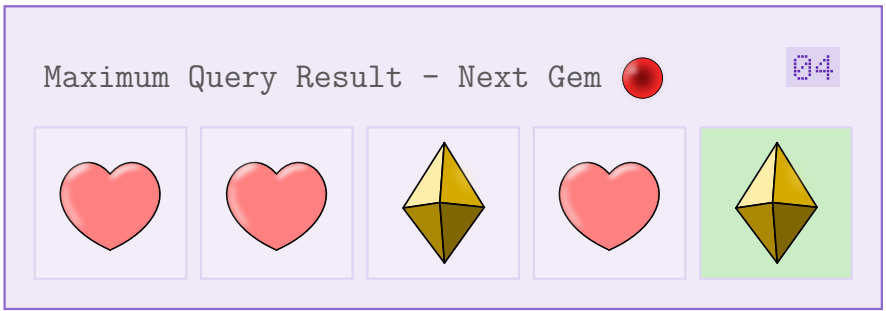


Finding Complete Set

Substitute Gem 4 04



Finding Complete Set



A thousand years later...

Finding Complete Set

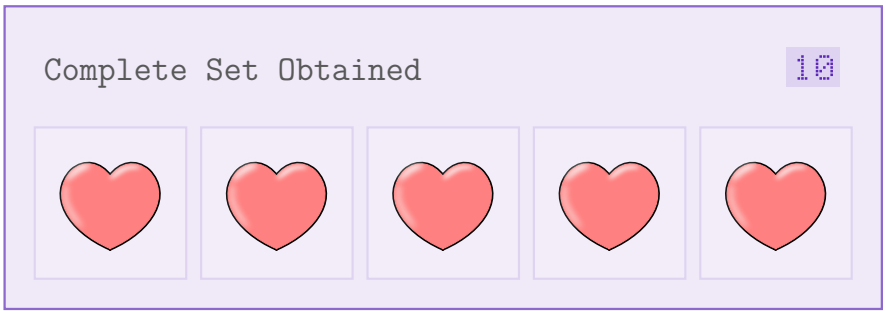


Table of Contents

Interpreting Partial Scoring

From now on, we restrict our focus on $N \leq 100$ subtasks.

For Subtask 4, a complete set is given. Your score is

$$\begin{cases} 46 & \text{if } Q \leq 27000 \\ 34 & \text{if } 27000 < Q \leq 32000 \\ 18 & \text{if } 32000 < Q \leq 128000 \end{cases}$$

For Subtask 5, no extra information is given. Your score is

$$\begin{cases} 24 & \text{if } Q \leq 27000 \\ 24 \times \dfrac{27000}{Q} & \text{if } 27000 < Q \leq 128000 \end{cases}$$

Interpreting Partial Scoring

Actually we only care about $N = 100$. Let's define $S = \frac{Q}{N^2}$.

For Subtask 4, a complete set is given. Your score is

$$\begin{cases} 46 & \text{if } S \leq 2.7 \\ 34 & \text{if } 2.7 < S \leq 3.2 \\ 18 & \text{if } 3.2 < S \leq 12.8 \end{cases}$$

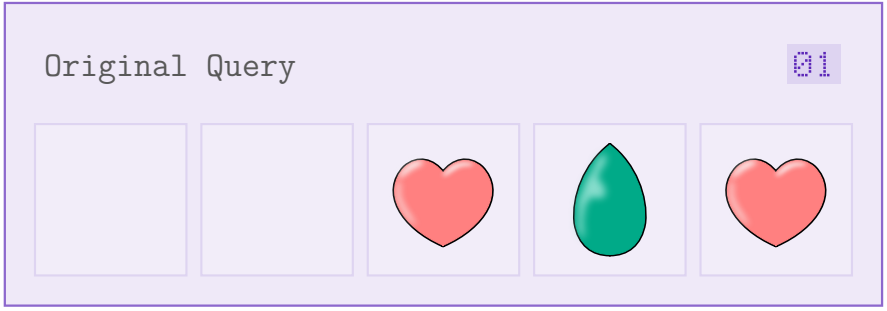
For Subtask 5, no extra information is given. Your score is

$$\begin{cases} 24 & \text{if } S \leq 2.7 \\ 24 \times \frac{2.7}{S} & \text{if } 2.7 < S \leq 12.8 \end{cases}$$

Intuitively, S is the rough number of queries you can spend on each gem.

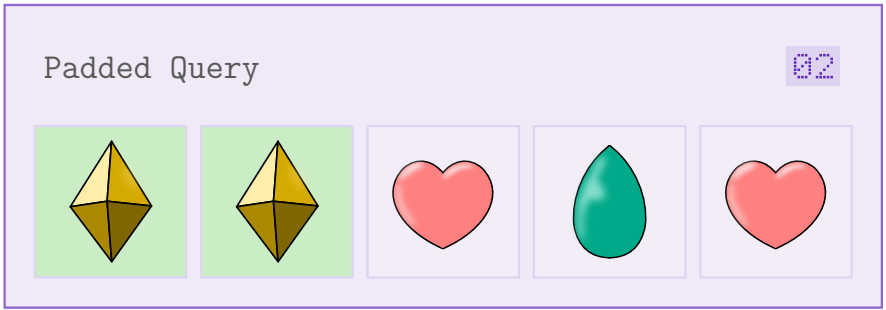
Using Complete Set

What is the use of *complete set*? Intuitively, we can use it as padding, so we can query with less than N gems as well.



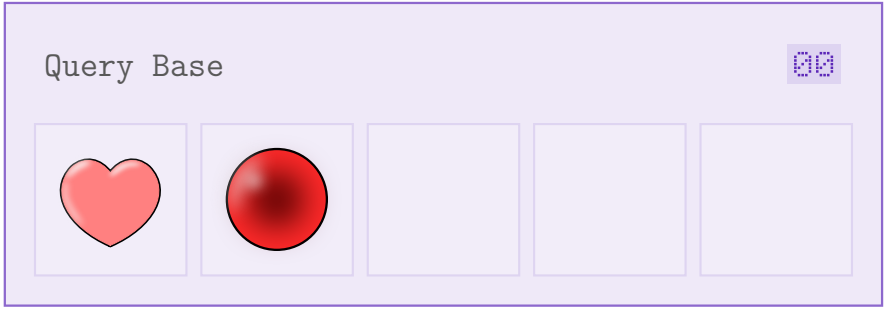
Using Complete Set

What is the use of *complete set*? Intuitively, we can use it as padding, so we can query with less than N gems as well.



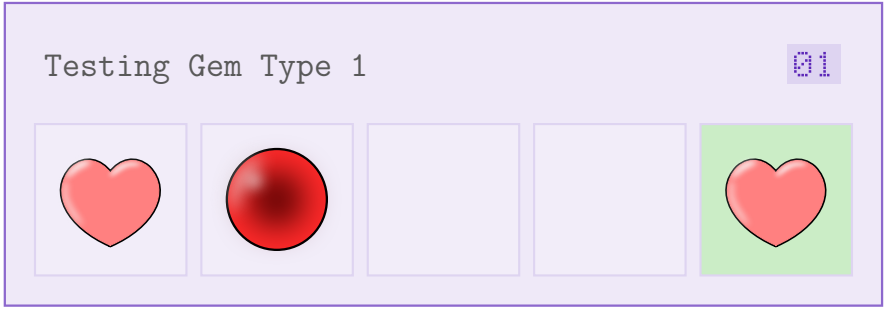
Faster than Linear

What is the simplest idea, *given a complete set*? *Binary Search*, by checking whether a gem is in a set of gems!



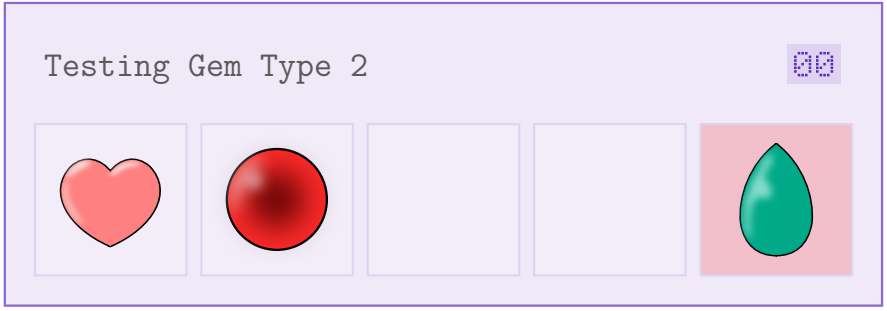
Faster than Linear

What is the simplest idea, *given a complete set*? *Binary Search*, by checking whether a gem is in a set of gems!



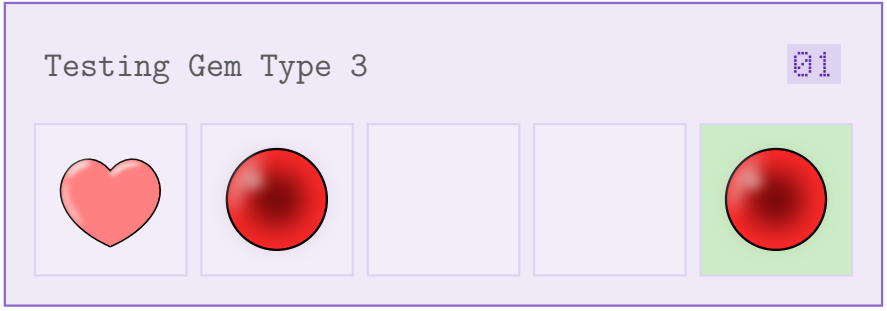
Faster than Linear

What is the simplest idea, *given a complete set*? *Binary Search*, by checking whether a gem is in a set of gems!



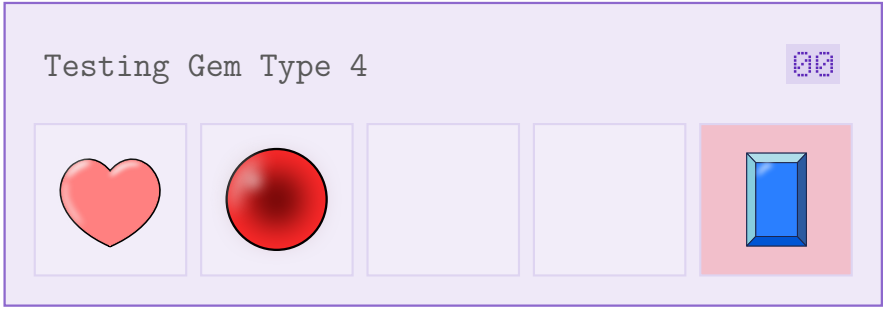
Faster than Linear

What is the simplest idea, *given a complete set*? *Binary Search*, by checking whether a gem is in a set of gems!



Faster than Linear

What is the simplest idea, *given a complete set*? *Binary Search*, by checking whether a gem is in a set of gems!



This yields a $S = \log_2 N \approx 7$ solution for Subtask 4.

Table of Contents

Better Query Method

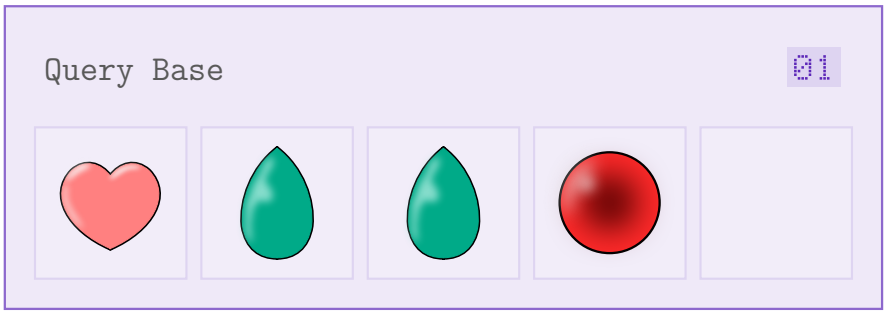
The returned value of queries is $\left[0, \frac{N(N-1)}{2}\right]$, but we are only using 0 or 1!

Given a set of $N-1$ gems A, we can query *number of appearance* of gem x in A.

Better Query Method

The returned value of queries is $\left[0, \frac{N(N-1)}{2}\right]$, but we are only using 0 or 1!

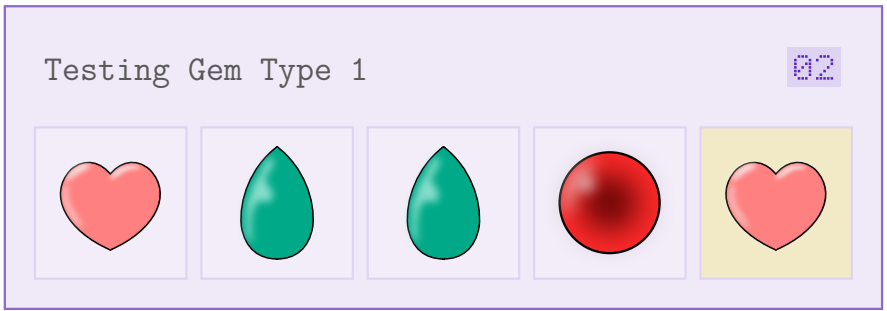
Given a set of $N-1$ gems A, we can query *number of appearance* of gem x in A.



Better Query Method

The returned value of queries is $\left[0, \frac{N(N-1)}{2}\right]$, but we are only using 0 or 1!

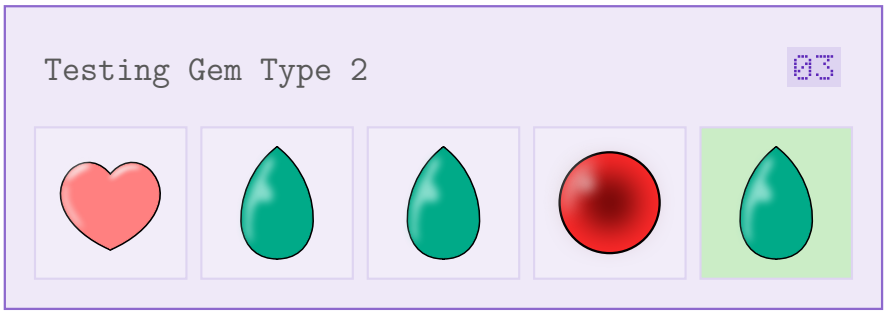
Given a set of $N-1$ gems A, we can query *number of appearance* of gem x in A.



Better Query Method

The returned value of queries is $\left[0, \frac{N(N-1)}{2}\right]$, but we are only using 0 or 1!

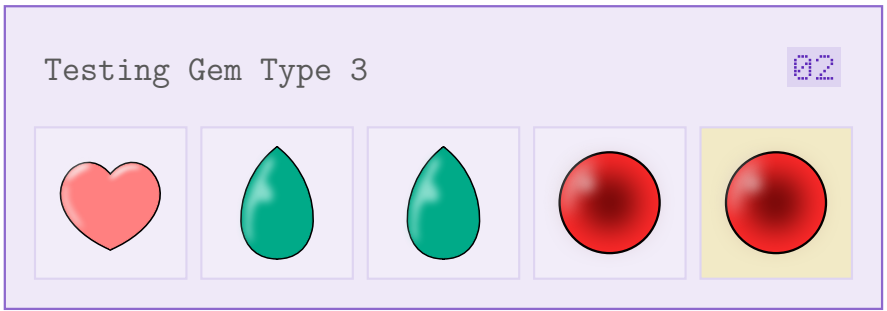
Given a set of $N-1$ gems A, we can query *number of appearance* of gem x in A.



Better Query Method

The returned value of queries is $\left[0, \frac{N(N-1)}{2}\right]$, but we are only using 0 or 1!

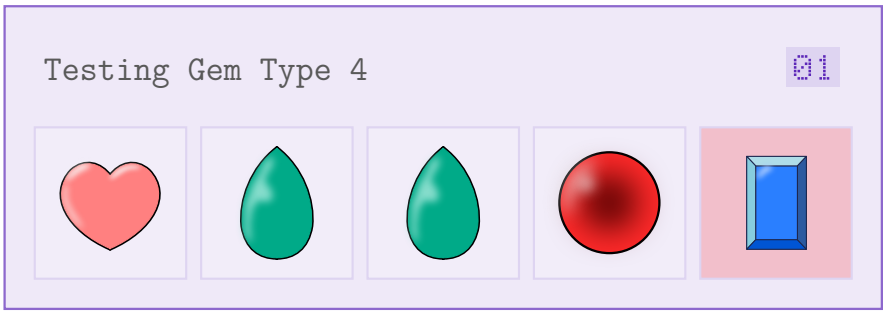
Given a set of $N-1$ gems A, we can query *number of appearance* of gem x in A.



Better Query Method

The returned value of queries is $\left[0, \frac{N(N-1)}{2}\right]$, but we are only using 0 or 1!

Given a set of $N-1$ gems A, we can query *number of appearance* of gem x in A.



Sqrt Search with Infinite Gems

Given *infinite* gems of each type. How to tell the type of a gem faster?

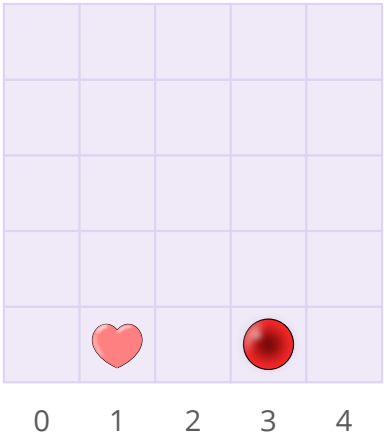
Suppose the type of a gem is known to be in $G = \{g_0, \ldots, g_{M-1}\}$.

Also let K be the number of sets we want to partition G into.

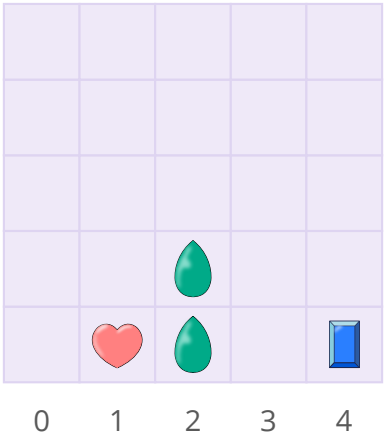
To split G among the K sets *evenly*, we can let the frequency array of A be

$$C[g_i] = i \bmod K \quad \text{for } 0 \leq i < M$$

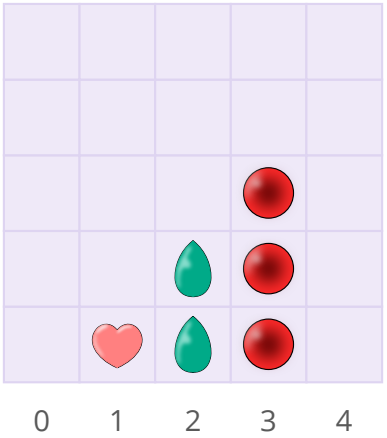
$M = 5, K = 2$



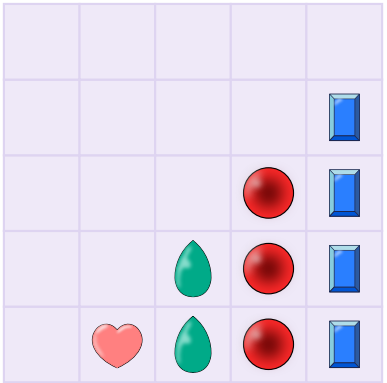
$M = 5, K = 3$



$M = 5, K = 4$



$M = 5, K = 5$



Sqrt Search with Infinite Gems

Note that the average of gems per type used $\leq \frac{K-1}{2}$. Therefore, as long as

$$\frac{M(K-1)}{2} \leq N-1$$

Then we can fit the gems into the query.

Thus, our choice for K is

$$K = \min\left(M, \left\lfloor \frac{2(N-1)}{M} \right\rfloor + 1\right)$$

For $N = 100$, M goes like $99 \to 33 \to 5 \to 1$, leads to $S \approx 3$.

Sqrt Search

Unfortunately, we do not have infinite gems of each type! Luckily, you can simply fallback to binary search or take min of $C[g_i]$ with the gems you have.

This works because $K = O(\sqrt{N})$, which means the extra cost is $O(N\sqrt{N})$. Together with $S \approx 3$, this fits into the limit $Q \leq 32000$ in Subtask 4 comfortably.

Table of Contents

Repeat Partition

All the above solutions assumes we have a complete set from the beginning. How to find the complete set to start with, of course in $O(N^2)$?

Partition

For gem sets X and Y where $|X| = N - 1$ and $X \cup Y$ forms K complete sets, we can partition $X \cup Y$ in $K \cdot N$ queries according to gem frequency in X.

How? Simply compare the query results for

1. Query $X \cup \{y\}$ for all $y \in Y$.
2. Query $X \setminus \{x\} \cup \{z_1, z_2\}$ for all $x \in X$ for some fixed $z_1 \neq z_2 \notin X$.

The implementation details is left as exercise for readers.

Repeat Partition

Partition

For gem sets X and Y where $|X| = N - 1$ and $X \cup Y$ forms K complete sets, we can partition $X \cup Y$ in $K \cdot N$ queries according to gem frequency in X.

The implementation is *somewhat* complicated (70 lines in author solution). Even if you don't obtain this during contest, you can probably survive with some weaker methods as long as they are feasible.

Randomly pick X to partition the gem sets repeatedly leads to $S \approx 3.7$.

You may also partition the smallest set repeatedly to obtain a complete set first, then apply sqrt search to obtain solution with $S \approx 3.2$.

Table of Contents

Further Optimisations

To get full score in this task, there are two pathway to take:

1. Parallel Sqrt Search. Still use the idea of sqrt search, but after partition of the gems, do sqrt search in parallel whenever possible. (Author best: $Q = 26069$)

2. Partition DP. Note that the sqrt search is far from optimal since we should be able to early break for some gems instead of using 3 queries every time. This can optimised using partition DP. (Author best: $Q = 26316$)

Either case you can easily pass the $Q \leq 27000$ limit. Unfortunately, both of them are *somewhat hard* to implement.

Takeaways

- Non-batch tasks are often easy for an entry score. Do leave some sufficient time for attempting it!
- Always write usable components for later use whenever you have some breakthroughs for some “stronger” queries, by making several queries with one function call. Do not feel constrained by the function signature provided!
- You should have tested locally instead of spamming the judge, even if is for the performance of randomised solutions! Make good use of the provided sample grader (or even modify it).