به نام خدا

در مورد سوال ۲ الگوریتم‌های زیادی وجود داره.

1. یکی از این الگوریتم‌ها که ساده‌ترینش هست اینه که از ابتدا تا انتهای اعداد اون کوئری رو پیمایش کنیم و اعدادی که وجود دارند رو تعدادشان را پیدا کنیم. این کار را بنده کردم و به جز ۵ تست کیس اول بقیه تست کیس‌ها را تایم لیمیت خوردم خب مشخص است که این ایده بهینه‌ای نیست
2. ایده دیگری که بنده داشتم که از نظر زمانی بهینه‌تر بود این بود که از ابتدا تا ایندکس iام آرایه را در دیکشنری نگه داریم در واقع آرایه‌ای از دیکشنری‌ها داریم که ایندکس هر آرایه بیانگر انتهای اون بازه است و ابتدای باز هم که همان عدد اول یا ایندکس آرایه است و دیکشنری که داریم از هر عدد به تعدادش مپ می‌کنه مثلاً آرایه ما عضو دومش ایندکس صفرم و اول آرایه که در سوال آمده‌اند را نگه داشته و تعداد هر کدام را نگه می‌داره مثلاً اگر ورودی ما ۲ ۲ ۳ ۴ باشه آرایه‌ای داریم که عضو اولش یه دیکشنری صرفاً از دو به یک ، عضو دومش از دو به دو چون که دو دو بار اومده و به همین شکل مثلاً عضو آخرشم که حالا در مثالی که عرض کردم ۲ ۲ ۳ ۴ دیکشنری هست که  کلیدهای آن به ترتیب ۲ ۳ ۴ و ولیوهای آن به ترتیب ۲ ۱ ۱ هستند به این ترتیب با وارد شدن هر کوئری با مراجعه به اون ایندکس از آرایه دو تا دیکشنری را از هم کم می‌کنیم ایندکس چهارم از آرایه‌ای که شامل دیکشنری هست داشته باشیم ۲ به چهار و درایندکس اول داشته باشیم دو به یک پس در اعداد بین دو سه چهار که اختلاف این‌هاست قطعاً سه تا دو داریم اما این از نظر زمانی بهینه است اما از نظر حافظه‌ای افتضاح است به همین دلیل بنده همون تست کیس‌هایی که در حالت اول تایم لیمیت می‌خوردم در حالت دوم مموری لیمیت خوردم.
3. ایده سوم استفاده از الگوریتم تصادفی یا همون رندوم است به این معنی که در ابتدا دیکشنری از اعداد به تعداد آن برای کل آرایه داریم پس در هنگام گرفتن کوئری‌ها ایندکس رندمی بین اول و آخر بازه مورد نظراست تولید می‌کنیم و سپس به کمک تابع bisect یا همون باینری سرچ در پایتون، ایندکس های بین اول و اخر کوئری را مشخص کرده و تعداد آن را پیدا میکنیم. اگر از نصف بیشتر شد خروجی کوئری مشخص است. پس از چند عدد رندوم اگر نشد، خروجی صفر است.

اما بنده پس از کلی این و اون ور کردن فهمیدم عدد اول در بسیاری از موارد جواب است و اگر نبود عدد صفر خروجی آن کوئری است. (گرفتن نمره 68) بالاخره پس از تست کردن های مختلف فهمیدم صرفا یا عدد اول بازه پاسخ است یا عدد وسط و در غیر این صورت پاسخ 0 است. این شد که این کد بنده نمره کامل رو گرفت. کد به زبان پایتون که کمتر کسی فکر میکرد که باهاش بشه نمره کامل گرفت. اما این الگوریتم درستی نیست چون به ازای اعداد خاص خروجی درست میدهد. پس نیاز به الگوریتم درست تری است. (و البته زبان دیگر)

البته این کد در زبان سی پلاس پلاس خیلی بعتر شد. استفاده از الگوریتم تصادفی بهتر. یعنی همین الگوریتم ولی به جای تست کردن فقط اولی و وسطی، بنده حدود 15 تا 20 عدد تصادفی را تست میکنم. حتی با 20 هم نمره کامل گرفتم ولی تایم لیمیت هم خوردم. عدد 17 عدد خوبی برای تولید این تعداد عدد تصادفی است تا جواب بهتر در زمان کمتری گرفت.

1. الگوریتم دیگر الگوریتم مو است که از اردر حدودا n رادیکال n است که هر جوره تایم لیمیت میخورد. ولی الگوریتم بر مرتب سازی کوئری ها بر اساس ایندکس شروع و همچنین تقسیم ارایه اصلی به دسته های رادیکال n ای است:

**Mo's Algorithm** is a square-root decomposition technique optimized for handling a large number of static range queries. It is particularly effective when all the queries are given at once and there are no updates to the array.

**How it works:**

* Divide the array into blocks of size approximately n\sqrt{n}n
* Sort the queries based on the **block** number of their starting points (and secondary sort by the ending point).
* As you process the sorted queries, you adjust the range by moving the left and right pointers incrementally, instead of recalculating the entire range from scratch.

**Process:**

1. **Sorting the queries**: You sort the queries first by their block and then by their right endpoint.
2. **Adjusting the range**: As you move from one query to the next, you only incrementally change the range by adding or removing elements (moving the left or right pointers).
3. **Efficient range calculation**: Since you only adjust a small portion of the range for each query, this saves time compared to recalculating the entire range.

**Complexity:**

* **Sorting queries**: O(q log q), where q is the number of queries.
* **Processing queries**: O(n+q)

1. الگوریتم دیگر الگوریتم درختی است که از اردر حدودا n لاگ n است. این روش هم به این شکل است:

**Segment Tree:**

A **segment tree** is a powerful data structure that allows you to efficiently solve range queries (like sum, minimum, maximum) and updates on an array in **logarithmic time**.

**How it works:**

* The segment tree divides the array into segments, each representing a range of elements.
* The tree is a complete binary tree, where each node represents some aggregate information about a segment (e.g., sum, max, count of elements).
* Each internal node covers a range of the array, and the leaf nodes store individual array elements.

**Operations:**

* **Query a range**: You can query any range of the array in O(log n)O(\log n)O(log n n) time by summing, or combining, the relevant segments.
* **Update an element**: You can update an element (or a range of elements) in O(log n), and the changes propagate through the tree.**Update an element**: You can update an element (or a range of elements) in O(log n) and the changes propagate through the tree.

**Example:**

For the problem you mentioned (finding majority element in a range):

1. Construct a segment tree where each node stores the **frequency** of numbers in its range.
2. For each query, you can merge the information from the left and right parts of the tree and check the frequency of each candidate.
3. Answer the query in **logarithmic** time.

**Complexity:**

* **Build**: O(n) for an array of size n.
* **Query**: O(log n)**Query**: O(log n)
* **Update**: O(log n)**Update**: O(log n)

**Pros:**

* Handles dynamic updates efficiently.
* Fast query times.

**Cons:**

* More complex to implement than a basic approach.
* Space complexity of O(2n) due to storing intermediate nodes.Space complexity of O(2n) due to storing intermediate nodes.