Last update: March 8, 2024 Original

MEX (minimal excluded) of a sequence

Given an array A of size N. You have to find the minimal non-negative element that is not present in the array. That number is commonly called the **MEX** (minimal excluded).

$$\begin{aligned}\operatorname{mex}(\{0,1,2,4,5\}) &= 3\\ \operatorname{mex}(\{0,1,2,3,4\}) &= 5\\ \operatorname{mex}(\{1,2,3,4,5\}) &= 0\end{aligned}$$

Notice, that the MEX of an array of size N can never be bigger than N itself.

The easiest approach is to create a set of all elements in the array A, so that we can quickly check if a number is part of the array or not. Then we can check all numbers from 0 to N, if the current number is not present in the set, return it.

Implementation

The following algorithm runs in $O(N \log N)$ time.

```
int mex(vector<int> const& A) {
    set<int> b(A.begin(), A.end());

    int result = 0;
    while (b.count(result))
        ++result;
    return result;
}
```

If an algorithm requires a $O(N)$ MEX computation, it is possible by using a boolean vector instead of a set. Notice, that the array needs to be as big as the biggest possible array size.

```
int mex(vector<int> const& A) {
    static bool used[MAX_N+1] = { 0 };

    // mark the given numbers
    for (int x : A) {
        if (x <= MAX_N)
            used[x] = true;
    }

    // find the mex
```

```
    int result = 0;
    while (used[result])
        ++result;

    // clear the array again
    for (int x : A) {
        if (x <= MAX_N)
            used[x] = false;
    }

    return result;
}
```

This approach is fast, but only works well if you have to compute the MEX once. If you need to compute the MEX over and over, e.g. because your array keeps changing, then it is not effective. For that, we need something better.

MEX with array updates

In the problem you need to change individual numbers in the array, and compute the new MEX of the array after each such update.

There is a need for a better data structure that handles such queries efficiently.

One approach would be take the frequency of each number from 0 to N, and build a tree-like data structure over it. E.g. a segment tree or a treap. Each node represents a range of numbers, and together to total frequency in the range, you additionally store the amount of distinct numbers in that range. It's possible to update this data structure in $O(\log N)$ time, and also find the MEX in $O(\log N)$ time, by doing a binary search for the MEX. If the node representing the range $[0, \lfloor N/2 \rfloor)$ doesn't contain $\lfloor N/2 \rfloor$ many distinct numbers, then one is missing and the MEX is smaller than $\lfloor N/2 \rfloor$, and you can recurse in the left branch of the tree. Otherwise it is at least $\lfloor N/2 \rfloor$, and you can recurse in the right branch of the tree.

It's also possible to use the standard library data structures `map` and `set` (based on an approach explained here). With a `map` we will remember the frequency of each number, and with the `set` we represent the numbers that are currently missing from the array. Since a `set` is ordered, `*set.begin()` will be the MEX. In total we need $O(N \log N)$ precomputation, and afterwards the MEX can be computed in $O(1)$ and an update can be performed in $O(\log N)$.

```
class Mex {
private:
    map<int, int> frequency;
    set<int> missing_numbers;
    vector<int> A;

public:
    Mex(vector<int> const& A) : A(A) {
        for (int i = 0; i <= A.size(); i++)
```

```
            missing_numbers.insert(i);

        for (int x : A) {
            ++frequency[x];
            missing_numbers.erase(x);
        }
    }

    int mex() {
        return *missing_numbers.begin();
    }

    void update(int idx, int new_value) {
        if (--frequency[A[idx]] == 0)
            missing_numbers.insert(A[idx]);
        A[idx] = new_value;
        ++frequency[new_value];
        missing_numbers.erase(new_value);
    }
};
```

Practice Problems

- AtCoder: Neq Min
- Codeforces: Informatics in MAC
- Codeforces: Replace by MEX
- Codeforces: Vitya and Strange Lesson
- Codeforces: MEX Queries

Contributors:

jakobkogler (80.16%) maleksware (15.08%) adamant-pwn (3.97%) harsh-1806 (0.79%)