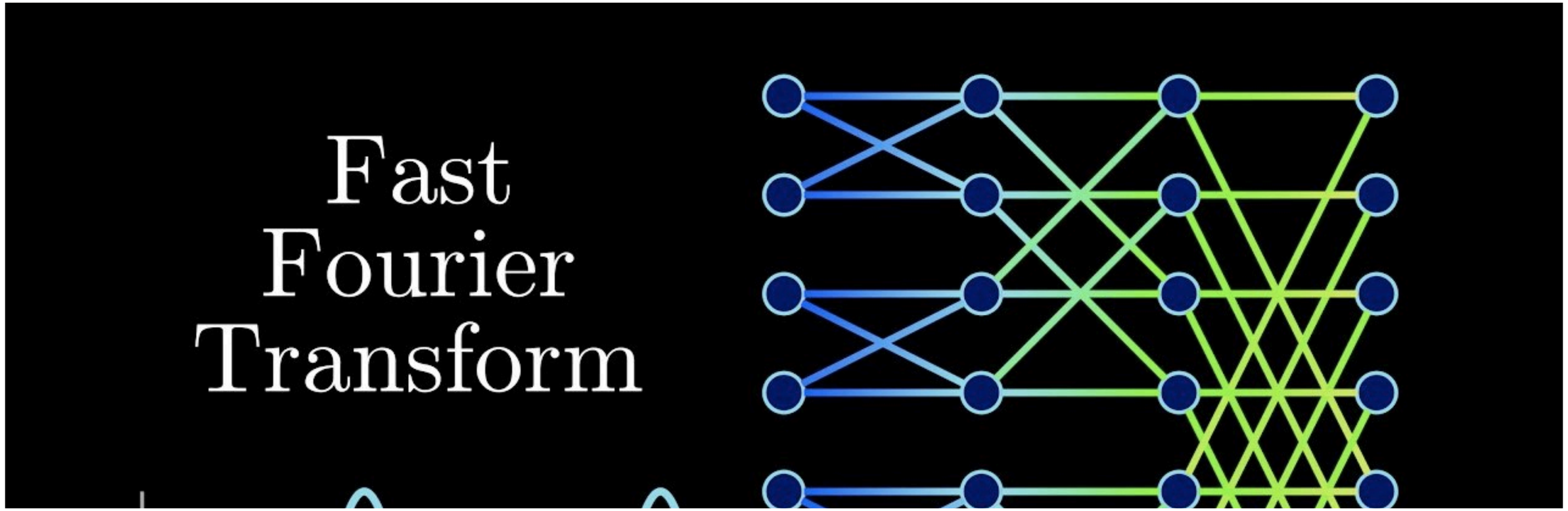

Codeforces

Fast Fourier transform

Ajay Kumar

PROBLEM:

Given an array A, find the smallest positive integer d such that d doesn't divide $|A_i - A_j|$ for any $1 \leq i < j \leq N$.

EXPLANATION:

Let $M = \max(A)$ be the maximum element of the array.
Note that any difference between array elements will also be $\leq M$.

So, if we were able to compute every pair of differences, we could solve the problem fairly easily, almost brute-force:

- Iterate d from 1 to $M + 1$
- Once d is fixed, check whether $d, 2d, 3d, \ldots$ appear as a difference of array elements.
- If none of them appear the answer is d; otherwise increment d by 1 and try again.

Assuming we can check whether x appears as a difference in $\mathcal{O}(1)$, the complexity of the above algorithm is $\mathcal{O}\left(\frac{M}{1} + \frac{M}{2} + \frac{M}{3} + \ldots\right) = \mathcal{O}(M \log M)$ (it's the harmonic series multiplied by M).

Finding all differences

The task of finding all pairwise differences among array elements (or between two different arrays) is in fact a rather standard application of FFT, the fast fourier transform.

The idea is to use polynomials, as follows:
Consider two polynomials $p(x) = p_0 + p_1 x + p_2 x^2 + \ldots$ and $q(x) = q_0 + q_1 x + q_2 x^2 + \ldots$, where:

- p_i is the number of times i occurs in A.
- q_i is the number of times $M - i$ occurs in A (recall that $M = \max(A)$)

p and q are degree-$(M + 1)$ polynomials.
Let $r(x) = p(x) \cdot q(x)$ be their product.

Note that for any x and y, such that $x \leq y \leq M$, exactly $p_y \cdot q_{M-x}$ is added to r_{M+y-x}.
In particular, for $d \geq 0$, r_{M+d} is exactly the number of pairs whose difference is d, which is exactly what we want!

r is the product of two polynomials, which can be computed in $\mathcal{O}(M \log M)$ using FFT.

TIME COMPLEXITY:

$\mathcal{O}(N + M \log M)$ per testcase, where $M = \max(A)$.

Fast Fourier transform

- This code is an implementation of a Fast Fourier Transform (FFT) algorithm.
- FFT is an efficient algorithm for computing the discrete Fourier transform (DFT) of a sequence.
- The discrete Fourier transform is a mathematical operation that decomposes a sequence of equally-spaced samples of a function into a weighted sum of sinusoids, each with a different frequency.
- The code first defines a number of constants and includes several standard libraries. It then defines the fft function, which takes a sequence of complex numbers and computes their DFT using the FFT algorithm. The function operates in-place on the input sequence and uses a provided array of complex roots of unity to help compute the transform efficiently.
- The code also defines an inverseFft function, which takes the DFT of a sequence and computes the inverse DFT.
- This function is simply a variant of the fft function that performs the inverse transform and scales the output by the length of the input sequence.

- The code also defines an inverseFft function, which takes the DFT of a sequence and computes the inverse DFT
- . This function is simply a variant of the fft function that performs the inverse transform and scales the output by the length of the input sequence.
- The code also defines a convolution function, which takes two sequences and computes their convolution.
- This function uses the fft and inverseFft functions to compute the convolution efficiently by transforming the sequences into the frequency domain, multiplying their DFTs, and then transforming the result back into the time domain.
- Finally, the code contains a main function that demonstrates how to use the fft, inverseFft, and convolution functions by applying them to some sample inputs.

```
void fft(vector<complex<ld>> &a, vector<int> &indexInEnd, vector<complex<ld>>
&roots)
{
    int n = (int)a.size();
    for (int i = 0; i < n; ++i) if (i < indexInEnd[i]) swap(a[i], a[indexInEnd[i]]);
    for (int length = 1, rootStep = n / 2; length < n; length <<= 1, rootStep >>= 1)
    {
        for (int i = 0; i < n; i += 2 * length)
        {
            for (int j = i; j < i + length; ++j)
            {
                complex<ld> value = roots[(j - i) * rootStep] * a[j + length];
                a[j + length] = a[j] - value;
                a[j] = a[j] + value;
            }
        }
    }
}
```

```
void inverseFft(vector<complex<ld>> &a, vector<int> &indexInEnd, vector<complex<ld>> &roots){
    fft(a, indexInEnd, roots);
    for (int i = 0; i < a.size(); ++i) a[i] /= a.size();
    reverse(a.begin() + 1, a.end());
}
```

- The code also defines an inverseFft function, which takes the DFT of a sequence and computes the inverse DFT
- This function uses the fft and inverseFft functions to compute the convolution efficiently by transforming the sequences into the frequency domain, multiplying their DFTs, and then transforming the result back into the time domain.

```
vector<ll> convolution(vector<ll> &a, vector<ll> &b){

    int resultSize = (int)a.size() + (int)b.size() - 1;
    int n = 1 << (32 - __builtin_clz(resultSize) - ((resultSize & (resultSize - 1)) == 0));
    vector<complex<ld>> tA(n), tB(n);
    for (int i = 0; i < a.size(); ++i) tA[i] = a[i];
    for (int i = 0; i < b.size(); ++i) tB[i] = b[i];
    vector<int> indexInEnd(n);
    for (int i = 1, log2 = -1, length = (31 - __builtin_clz(n)); i < n; ++i)
    {
        if (!(i & (i - 1))) ++log2;
        indexInEnd[i] = indexInEnd[i ^ (1 << log2)];
        indexInEnd[i] |= (1 << (length - log2 - 1));
    }
    vector<complex<ld>> roots(n);
    for (int i = 0; i < n; ++i){
        ld alpha = 2 * PI * i / n;
        roots[i] = {cos(alpha), sin(alpha)};
    }
    fft(tA, indexInEnd, roots);
    fft(tB, indexInEnd, roots);
    vector<complex<ld>> tC(n);
    for (int i = 0; i < n; ++i) tC[i] = tA[i] * tB[i];
    inverseFft(tC, indexInEnd, roots);
    vector<ll> c((int)a.size() + (int)b.size() - 1);
    for (int i = 0; i < (int)c.size(); ++i) c[i] = (long long)(tC[i].real() + 0.5);
    return c;
}
```

```
int32_t main()
{
   ios_base::sync_with_stdio(false);
   cin.tie(0);
   vector<vector<int>> divis(N+1);
   for(int i=1;i⩽N;i++)
        for(int j=i;j⩽N;j+=i)
            divis[j].pb(i);

   int t;
   cin >> t;
```

```
while(t--){
        int n;
        cin >> n;
        vector<int> ar(n);
        for(int i=0;i<n;i++)
            cin >> ar[i];
        int mx=*max_element(ar.begin(),ar.end());
        vector<long long> A(mx+1,0),B(mx+1,0);
        for(int i=0;i<n;i++){
            A[ar[i]]++;
            B[mx-ar[i]]++;
        }
        auto diff=convolution(A,B);
        vector<int> bad(mx+2,0);
        for(int i=1;i⩽mx;i++){
            if(diff[i+mx]){
                for(auto divisors : divis[i])
                    bad[divisors]++;
            }
        }
        int ans=2;
        for(;bad[ans];ans++)
        {}
        cout << ans << "\n";
   }
```