2601. Prime Subtraction Operation

You are given a **0-indexed** integer array `nums` of length `n`.

You can perform the following operation as many times as you want:

- Pick an index `i` that you haven't picked before, and pick a prime `p` **strictly less than** `nums[i]`, then subtract `p` from `nums[i]`.

Return *true if you can make `nums` a strictly increasing array using the above operation and false otherwise.*

A **strictly increasing array** is an array whose each element is strictly greater than its preceding element.

Example 1:

```
Input: nums = [4,9,6,10]
Output: true
Explanation: In the first operation: Pick i = 0 and p = 3, and then subtract 3 from nums[0], so that nums becomes [1,9,6,10].
In the second operation: i = 1, p = 7, subtract 7 from nums[1], so nums becomes equal to [1,2,6,10].
After the second operation, nums is sorted in strictly increasing order, so the answer is true.
```

Example 2:

```
Input: nums = [6,8,11,12]
Output: true
Explanation: Initially nums is sorted in strictly increasing order, so we don't need to make any operations.
```

Example 3:

```
Input: nums = [5,8,3]
Output: false
Explanation: It can be proven that there is no way to perform operations to make nums sorted in strictly increasing order, so the answer is false.
```

Constraints:

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 1000`
- `nums.length == n`

2601. Prime Subtraction Operation

/*

Brute force

Time complexity: $O\left(n \times limit \times \sqrt{(limit)}\right)$

Space complexity: O(1)

*/

```
class Solution {
  public:
    bool primeSubOperation(std::vector<int>& nums){
      auto is_prime=[&](int p)-> bool{
        int sr=sqrt(p);
        for(int i=2;i<=sr;++i) {
          if(p%i==0) return false;
        }
        return true;
      };

      int n=nums.size();

      for(int i=0;i<n;++i){
        // Largest prime should be less than nums[i]-nums[i-1]
        int limit=i==0?nums[0]:nums[i]-nums[i-1];

        // if limit<=0, means nums[i]-nums[i-1]<=0
        // ==> nums[i]<=nums[i-1]
        // ==> we cannot make a strictly incresing array
        if(limit<=0) return false;

        // Find largest prime, such that 2<= Largest prime < limit
        // Largest prime=0, otherwise
        int j=limit-1;
        while(j>=2 && !is_prime(j)) j--;
        int largest_prime=j>=2?j:0;

        // Subtract the largest prime number that
        // we found from nums[i]
        nums[i]-=largest_prime;
      }
      return true;
    }
};
```

2601. Prime Subtraction Operation

```
/*
    Brute force+Preprecessing primes numbers
    Time complexity: O(mx * sqrt(limit)+n)
    Space complexity: O(mx)
*/
class Solution {
    public:

        bool primeSubOperation(std::vector<int>& nums){

            auto is_prime=[&](int p)-> bool{
                int sr=sqrt(p);
                for(int i=2;i<=sr;++i) {
                    if(p%i==0) return false;
                }
                return true;
            };

            int mx=*std::max_element(nums.begin(),nums.end());

            std::vector<int> all_primes(mx+1);
            for(int i=2;i<=mx;++i) all_primes[i]=is_prime(i)?i:all_primes[i-1];

            int n=nums.size();
            for(int i=0;i<n;++i){
                int limit=i==0?nums[0]:nums[i]-nums[i-1];

                if(limit<=0) return false;

                int largest_prime=all_primes[limit-1];

                nums[i]-=largest_prime;
            }
            return true;
        }
};
```

2601. Prime Subtraction Operation

```
/*
    Sieve of Eratosthenes
    Time complexity: O(n+mx log log mx)
    Space complexity: O(mx)
*/
class Solution {
    public:
        bool primeSubOperation(std::vector<int>& nums){
            int mx=*std::max_element(nums.begin(),nums.end());
```

```
            // Build the sieve array.
            std::vector<bool> sieve(mx+1,true); // Mark all as primes
            sieve[0]=sieve[1]=0; // 0 and 1 are not primes
            int sr=sqrt(mx+1);
            for(int i=2;i<=sr;++i){
                if(sieve[i]){ // If is prime
                    // Mark its all multiples as not prime
                    for(int j=i*i;j<=mx;j+=i){
                        sieve[j]=false;
                    }
                }
            }
```

```
      int n=nums.size();

      // Start by storing the current value as 1, and the initial index as 0.
      int cur=1;
      int i=0;
      while(i<n){
        // Compute the difference needed to make nums[i] equal to currValue.
        int diff=nums[i]-cur;

        // if diff<0, means that nums[i]<cur
        // Can't make strictly increasing array
        if(diff<0) return false;

        // If the difference is prime or zero, then nums[i] can be made
        // equal to currValue.
        if(sieve[diff] || diff==0){
          i++;
          cur++;
        }
        // Otherwise, try for the next currValue.
        else cur++;
      }

      return true;
    }
};
```