Classical problems and tricks

11 March 2016

Find all the primes smaller than N

You are given an integer N, return an array with all the primes that are smaller than or equal to N.

e.g. N = 20

return: [2,3,5,7,11,13,17,19]

Solution: Eratosthenes's Sieve

Idea: Eliminate the numbers that are not primes

Decent approach: For each i from 2 to N, mark 2*i, 3*i, …. (N/i) * i as not prime.
Complexity: N(1/2 + 1/3 + … + 1/N) ~ O(NlogN) because math.

Better approach: If the current number is marked as not prime, all its multiples were also marked as not prime, so we can skip it. Note: if the current number is not marked, it is certainly a prime, so we do the multiple marking only for primes.
Complexity: N * (sum (1/p), p prime <= N) ~ O(N*log(log(N)) also because math.

Speedup but same complexity: For each prime p, mark p*p, (p+1)*p,...(N/p)*p

Interval adding updates

You have an array of N elements equal to 0, and M queries of the type:

- (x,y,z): add value z to all the elements between a[x] and a[y]

Print the array after all the queries are performed.

Example: N = 5; queries (1,3,2) (2,4,1)

result: [2,3,3,1,0]

Constraint: N and M are around 1,000,000

Brute force solution

For every query iterate through all of the indexes in the interval and add the value.

This will be too slow: O(N * M)

It calculates the state of the array after each query, which we don’t care about.

Solution

For a given array a[1], a[2],....,a[N], let's look at the difference array:

b[1] = a[1], b[2] = a[2]-a[1],...., b[N] = a[N]-a[N-1]

What happens to b when we have a query?

The new a: a[1], ….a[x-1], a[x]+z, a[x+1]+z, ...a[y] + z, a[y+1], ...a[N]

The new b: All values stay the same except b[x] → b[x] + z and b[y+1] → b[y+1] - z

Let's perform all queries on b, then reconstruct a as the prefix sums of b.

Complexity: O(N+M) → O(1) for each query and O(N) for the reconstruction

2D Generalization

Query: add value z in a submatrix with top left (x1,y1) and bottom right (x2,y2).

The brute force solution would involve adding z to all cells (i, j) with x1 <= i <= x2 and y1 <= j <= y2, which would be N^2 per query.

Hint: In 1D, we did the queries on the array whose prefix sum is our array.

2D Generalization solution

Let’s call the “prefix matrix” of A, the matrix P so that P[x][y] is the sum of all elements (i,j) in A with i<=x and j<=y.

Let’s do the queries on a matrix B whose prefix is our input matrix!

Add z to B[x1][y1] => add z to all A[i][j] with i>=x1, j>=y1

Remove z from B[x1][y2+1] and B[x2+1][y1] => z is added to all A[i][j] with x1>=i>=x1, y2>=j>=y1, but also subtracted from all A[i][j] with i>=x2+1 and j>=y2+1

Fix this by adding z to B[x2+1][y2+1] !

Complexity: O(number of queries + size of matrix)

Majority element

You have an array of length N, return the majority element or -1 if there isn’t any.

Majority element is defined as an element that appears at least N/2 + 1 times.

Solution 1: Hashes

Just iterate through all the elements and count how many times they appear. If there is one that reaches N/2 +1 return it, else return -1.

Solution 2: Sorting

What if you can’t use extra memory ?

Well if there is a value that appears N/2 +1 times, it will clearly appear in the middle of the sorted array.

We’ll sort the array and count the occurrences of v[N/2]. This will be O(NlogN).

But can be done with O(N) if we use quicksort’s Nth element algorithm.

Solution 3: Fighting

Let’s consider the following process: the numbers repeatedly pair up and “fight” each other (different → both die, equal → they stay) and stop when they are all equal, or all dead. If there is a majority element, there will be numbers equal to its value left at the end of the process (but we still need to test).

To do this in linear time, we traverse the array and maintain the value and counter of the current candidate. If the counter is 0, we add the current number with counter 1. Otherwise, if the current value is the same as the candidate we increase the counter, and if it’s different we decrease the counter. At the end we check the candidate is indeed the majority element by counting the occurrences.

Majority element + update

You have the same problem as before, only now you have 3 extra operations that may come in any order:

- Add(x) which inserts x in the array
- Del(x) which deletes an occurrence of x from the array
- Majority() returns the majority element (or -1 if there is none).

Hint: Array + hashes + randomness.

Majority element + update solution

We use an array to store all the numbers and a map to count value occurrences.

When we insert or delete numbers, we do so in the array, and we also increase / decrease the counters for the corresponding values.

Let’s assume there is a majority element. If we select a random element from the array, what’s the probability that we select the majority? → at least 1/2

So let’s select a random element from the array K times, and check if its counter is > N/2. If after k times we haven’t found one, return -1.

Probability of failing will be 1/(2^K), which for K = 100 shouldn’t happen in this universe.

Calculate x ^ y

Given x and y, calculate the remainder of dividing x ^ y by number MOD.

Constraint: x <= 1,000,000,000; y <= 10^18

Solution: Logarithmic exponentiation

If y was a power of 2 (y = 2^k), we could simply start from x and repeatedly square the result k times. We can do something similar in the general case:

- If y is 0, return 1
- For y > 0, compute z = x ^ (y/2) recursively
- If y is even, return z*z % MOD
- Else return x*z*z % MOD

Complexity: O(log(y))

Calculate the Nth fibonacci element

The Fibonacci sequence is given by x0 = 0, x1 = 1, x[N] = x[N-1] + x[N-2] for N > 1.

Calculate the Nth term modulo MOD, where N is 10^18.

Solution

If we look at the vector y[N] = (x[N], x[N+1]), we have

y[N+1] = (x[N+1], x[N+2]) = (x[N+1], x[N] + x[N+1]) = (0, 1; 1, 1) * y[N].

Repeating this, we have y[N] = A^N * y[0] (and x[N] is the first term of y[N]).

We can now apply the same method we did for x^y, but with matrix multiplications instead.

Complexity: O(log(N))