Code Jam 2015 - Round 1A

Analysis: Haircut

We describe a few algorithms that solve the problem, of increasing efficiency.

Direct simulation

Iterate over each minute, from the shop opening at time T = 0 until the last customer is served. At time T assign new customers to all available barbers (a barber is available if T is multiple of his M). Finally, report the barber who serves you.

```
public int naiveGetBarberNumber(int N) {
  int customer = 1;
  for (int T = 0; ; T++) {
    for (int barber = 0; barber < B; barber++) {
      if (T % M[barber] == 0) {
        if (customer == N) return barber;
        customer++;
      }
    }
  }
}
```

This algorithm has time complexity O(N * max(M) * B), so it will be very slow, even for a small input, since N can be as large as 1,000,000,000.

Exploit periodicity

Consider the case where there are two barbers B1 and B2, who take 2 and 3 minutes respectively to cut a customer's hair.

Time	Events
T = 0	Both barbers are ready to serve customers. B1 serves customer #1 and B2 serves customer #2
T = 2	B1 serves customer #3
T = 3	B2 serves customer #4
T = 4	B1 serves customer #5
T = 6	Both barbers are ready to serve customers. B1 serves customer #6 and B2 serves customer #7

At T = 6, both barbers have become available, just as they were at T = 0. So we will see the same pattern of availability for the next 6 minutes as we did for the first 6 minutes — at T = 2 + 6, B1 will serve customer #(3+5); at T = 3 + 6, B2 will serve customer #(4+5), and so on until T = 6 + 6, at which point the process starts repeating itself again.

What's so special about 6? It's the least common multiple (LCM) of M1 = 2 and M2 = 3. At time T = LCM(M1, M2) = 6 each barber is available, because T % M = 0 for every barber. We can compute the LCM of all Ms as follows: LCM(M1, M2, M3, ...) = LCM(M1, LCM(M2, M3, M4, â€¦)) and the least common multiple of two numbers is LCM(A, B) = A * B / GCD(A, B).

We can exploit the fact that the whole process is periodic and only simulate for a small number of customers. For example, say M1 = 2, M2 = 3, and you are N = 14th in line. We already know that we have a period of LCM(2, 3) = 6. During one phase B1 serves LCM(2, 3) / M1 = 3 customers and B2 serves LCM(2, 3) / M2 = 2 customers, i.e. in total 5 customers per phase are served in the shop. Since N = 14, you will be served in the 3rd phase. When the third phase starts you are going to be 4th in line, because a total of 10 customers have been served in the previous two phases. Finally, to figure out your barber's number, we can naively simulate your phase, similar to what we did in the first solution.

Since we are only simulating a single phase, we only really need to simulate at most LCM(M1, M2, M3, â€¦) minutes. So our improved algorithm has time complexity O(B * LCM(M1, M2, M3, â€¦)). Note that the LCM of all Ms is not going to exceed max(M)^B, i.e. LCM of all Ms is less than 25^5 for the small input.

```
public int slowGetBarberNumber(int N) {
  int period = M[0];
  for (int i = 1; i < B; i++)
    period = period / gcd(period, M[i]) * M[i];
  int customers_per_phase = 0;
  for (int i = 0; i < B; i++)
    customers_per_phase += period / M[i];
  int N_in_my_phase = N % customers_per_phase;
  return naiveGetBarberNumber(N_in_my_phase != 0
    ? N_in_my_phase : customers_per_phase);
}
```

For the large input, B and Ms can be as high as 10,000, so the LCM of them can get very large, and this approach will not work.

Binary Search

For a given time T, it is easy to compute the number of customers who have been assigned to a barber up to and including at time T. The number of customers who have been assigned to barber i is T/M_i + 1 (rounded down), so we can just sum these values for all the barbers.

```
public int countServedCustomers(long T) {
  if (T < 0) return 0;
  int served_customers = 0;
  for (int barber = 0; barber < B; barber++)
    served_customers += T / M[barber] + 1;
  return served_customers;
}
```

This means we can use a binary search to figure out the time T when you are going to be served. After that, all you are left to do is figure out which of the available barbers at time T is going to serve you.

Keep in mind that at time T multiple barbers may become available, so you have to account for the customers that are ahead of you in line and are going to be served at the same time. Since you know that you will be served at time T, the number of potential customers ahead of you that are going to be served at time T is less than the number of available barbers. We can then simulate that, given that we know the number of customers that are going to be served up to and including time T-1. More precisely, the number of customers to be seated in the barber chair at time T is equal to countServedCustomer(T) - countServedCustomers(T-1).

What should the bounds for the binary search be?

For the upper bound, imagine a worst-case scenario: every customer ahead of you is served by the slowest and the only available barber. Meaning you are guaranteed to be served after max(M)*N minutes. For the lower bound, in the best case you are going to be served at the time the shop opens.

The implementation below assumes that you always will have been served at T = high or earlier, while at T = low you have not been served yet. So initially we want low to be -1, not 0.

The final complexity of this solution is O(B * log(N * max(M))).

```
public int fastGetBarberNumber(int N) {
  long low = -1, high = 10000L*N;
  while (low + 1 < high) {
    long mid = (low + high) / 2;
    if (countServedCustomers(mid) < N)
      low = mid;
    else
      high = mid;
  }
  long T = high;
  int customers_served_before =
    countServedCustomers(T - 1);
  int customers_to_be_served =
    N - customers_served_before;
  for (int barber = 0; barber < B; barber++)
    // Is the barber available at time T?
    if (T % M[barber] == 0) {
      customers_to_be_served--;
      if (customers_to_be_served == 0)
        return barber;
    }
}
```