Code Jam 2016 - Qualification Round

Analysis: Coin Jam

One approach here is to enumerate strings of length **N** and see if they are jamcoins. To iterate over potential jamcoins, we can iterate over the base-2 interpretations (odd numbers between $2^{N-1}+1$ and $2^{N}-1$, inclusive) and use a recursive method to convert these to different bases.

For the Small dataset, it is sufficient to use trial division to find factors of potential jamcoins. Concretely, we can look for a nontrivial divisor of an integer *k* by testing divisibility of every integer in increasing order from 2 to the square root of *k*, inclusive and stopping if we find one. Searching past the square root of the number is not needed as if *k* has some nontrivial divisor *d*, then *k / d* is also a non-trivial divisor and the smaller of *d* and *k / d* is at most the square root of *k*. A sample implementation of this in C++ looks like this:

```
long long convertBinaryToBase(int x, int base) {
  // Some languages have built-ins which make this easy.
  // For example, in Python, we can avoid recursion and
  // just return int(bin(x)[2:], base)
  if (x == 0)
    return 0;
  return base * convertBinaryToBase(x / 2, base) + (x % 2);
}

long long findFactor(long long k) {
  for (long long d = 2; d * d <= k; d++)
    if (k % d == 0)
      return d;
  return 0;
}

void printCoins(int N, int X) {
  for (long long i = (1 << N-1) + 1; X > 0; i += 2) {
    vector<long long> factors;
    for (int base = 2; base <= 10; base++) {
      long long x = convertBinaryToBase(i, base);
      long long factor = findFactor(x);
      if (!factor)
        break;
      factors.push_back(factor);
    }
    if (factors.size() < 9)
      continue;

    cout << convertBinaryToBase(i, 10);
    for (long long factor : factors)
      cout << " " << factor;
    cout << endl;
    X -= 1;
  }
}
```

Solving the large test case may present unique challenges, depending on the programming language used. We're given that **N** = 32, which means that in base 10 we'll have 32-digit

numbers, which we can't store in a 64-bit integer. These numbers are also large enough that running the trial division algorithm on a single prime would take a huge amount of time, potentially longer than the duration of this contest! While we can solve these issues by stopping the trial division early (say, after checking up to 1000) and using arbitrary precision integers, there's actually a much nicer approach!

We can make some observations about jamcoins if we look at the output to the Small dataset from our program above. Considering the first 50 jamcoins of length 16, we find that 18 of them have divisors "3 2 3 2 7 2 3 2 3" and 11 of them have divisors "3 2 5 2 7 2 3 2 11". What's the pattern in these numbers? The numbers 5, 7, 11 from the second list provide a useful hint: they're one more than their respective bases. Giving this some thought, we can notice that the second list of divisors is formed by taking the smallest prime factor of *b*+1 for each base *b*. This suggests that *b*+1 is always a divisor for each of these 11 jamcoins, and we can easily verify that this is true. Understanding the other common divisor list is left as an exercise for the reader.

Note that *b*+1 in base *b* is 11_b. A simple test exists to determine divisibility of 11 in base-10: a number is divisible by 11 if the sum of its digits in odd positions and the sum of its digits in even positions differ by a multiple of 11. This divisibility rule for 11 can be extended to a simple rule: a string of `0`s and `1`s which starts and ends with a `1` and has the same number of `1`s at odd and even indices is divisible by *b*+1 when interpreted as a base-*b* number. Such a string is therefore a jamcoin, but not every jamcoin necessarily matches this condition. A stricter condition that may be easier to notice is that a string of `0`s and `1`s which starts and ends with a `1` is a jamcoin if all `1`s are paired, i.e. it matches the regular expression `11(0|11)*11`. For example, in any base *b*, $11011_b = 1001_b \times 11_b$.

The last example here also suggests a more general rule. Consider any string `p` of `0`s and `1`s with at least two characters, which starts and ends with a `1`. Any string which is `p` repeated multiple times with any number of `0`s between repetitions is a jamcoin. In the previous example we have `p` = `11` and this general rule can be expressed as the regular expression `(1[01]*1)(0*\1)+`. For example, in any base *b*, $\underline{11101}000\underline{11101}0\underline{11101}_b = 11101_b \times 100000001000001_b$.

We can use any of these rules to mine jamcoins easily. The Python 2 code below mines jamcoins with exactly 5 pairs of `11`s, which finds enough for both the Small and Large datasets.

```
def printCoins(N, X):
  # N digits, 10 1s, N-10 0s
  for i in range(N-10):
    for j in range(N-10-i):
      for k in range(N-10-i-j):
        l = N-10-i-j-k
        assert l >= 0
        template = "11{}11{}11{}11{}11"
        output = template.format("0"*i, "0"*j, "0"*k, "0"*l)
        factors = "3 2 5 2 7 2 3 2 11"
        print output, factors
        X -= 1
        if X == 0:
          return
  # If we get here, we didn't mine enough jamcoins!
  assert False
```