Homework 5, CSE 232

Due October 1st Note: On most of the problem sets through the semester, I'll put a horizontal line with "Optional" under it. Any problems below this section are encouraged - I think they're interesting and will help you learn the subject - but not necessary to complete in order to get credit for the homework.

Problem 1

This week's problem is GCJ Qualification 2011 - Candy Splitting.

Here's the problem statement:

> Sean and Patrick are brothers who just got a nice bag of candy from their parents. Each piece of candy has some positive integer value, and the children want to divide the candy between them. First, Sean will split the candy into two piles, and choose one to give to Patrick. Then Patrick will try to calculate the value of each pile, where the value of a pile is the sum of the values of all pieces of candy in that pile; if he decides the piles don't have equal value, he will start crying.
>
> Unfortunately, Patrick is very young and doesn't know how to add properly. He almost knows how to add numbers in binary; but when he adds two 1s together, he always forgets to carry the remainder to the next bit. For example, if he wants to sum 12 (1100 in binary) and 5 (101 in binary), he will add the two rightmost bits correctly, but in the third bit he will forget to carry the remainder to the next bit:

1100 + 0101 —— 1001

> So after adding the last bit without the carry from the third bit, the final result is 9 (1001 in binary). Here are some other examples of Patrick's math skills:
>
> $5 + 4 = 1$
>
> $7 + 9 = 14$
>
> $50 + 10 = 56$
>
> Sean is very good at adding, and he wants to take as much value as he can without causing his little brother to cry. If it's possible, he will split the bag of candy into two non-empty piles such that Patrick thinks that both have the same value. Given the values of all pieces of candy in the bag, we would like to know if this is possible; and, if it's possible, determine the maximum possible value of Sean's pile.

Input

> The first line of the input gives the number of test cases, T. T test cases follow. Each test case is described in two lines. The first line contains a single integer N, denoting the number of candies in the bag. The next line contains the N integers C_i separated by single spaces, which denote the value of each piece of candy in the bag.

Output

> For each test case, output one line containing "Case #x: y", where x is the case number (starting from 1). If it is impossible for Sean to keep Patrick from crying, y should be the word "NO". Otherwise, y should be the value of the pile of candies that Sean will keep.

Limits

> $1 \leq T \leq 100$.
>
> $1 \leq C_i \leq 106$.

Small dataset

$2 \leq N \leq 15$.

Large dataset

$2 \leq N \leq 1000$.

a) Using the way in which Patrick adds numbers, write a function that takes a list of candy values and returns how much Patrick thinks all the candies are worth together.

Patrick is adding binary numbers with the XOR operation rather than binary addition. As such, we can write this function as:

```
In [1]: def patrick_addition(candies):
            total = 0
            for candy in candies:
                total ^= candy #XOR in Python
            return total
```

b) What are the maximum number of possible ways Sean can divide the candies into two piles for the Small and the Large cases? Write a complete search solution that tests each possible division for whether Patrick is satisfied and finds the maximum value for Sean's pile.

For each division, either Sean's or Patrick's pile can contain a particular candy, so the total number of possible subsets is 2^N. For the small case, this is only 32,768, while it is approximately 10^{300} for the large case.

We can use a single integer as a bitmask to represent Sean's pile for each division, from 1 to $2^N - 1$ (*not* 0 to 2^N, because each pile must have at least one candy), constructing Patrick's complementary pile from Sean's.

```
In [50]: def bitmask_to_list(mask, candies):
             return [candy for i, candy in enumerate(candies) if (mask >> i) & 1]

         def complement_set(mask, N):
             return ((1 << N) - 1) ^ mask

         def patrick_addition(candies):
             total = 0
             for candy in candies:
                 total ^= candy #XOR in Python
             return total

         def complete_search(candies):
             N = len(candies)
             maxval = -1
             for sean in xrange(1, pow(2, N) - 1):
                 patrick = complement_set(sean, N)

                 #Turn the bitmasks into lists of candy values
                 sean = bitmask_to_list(sean, candies)
                 patrick = bitmask_to_list(patrick, candies)

                 if patrick_addition(sean) == patrick_addition(patrick):
                     #Patrick's happy!
                     maxval = max(sum(sean), maxval)
```

```
        if maxval == -1:
            return "NO"
        else:
            return maxval
```

c) We can calculate whether two numbers are equal using no direct comparison operators (e.g., "==", "!=", etc.) but instead with only a single binary operator. Write a function that, given, Patrick's estimate of the values of two piles of candy, returns a Boolean describing whether he thinks they're equal.

The XOR operation should work for this. If we XOR a number with itself, it will always give us 0 as an output, while XORing a number with any different number will give us something other than zero. Thus, we can write this as:

```
In [8]: def patrick_equal(value1, value2):
            return not bool(value1 ^ value2)
```

d) Given your answers to the above questions, how does Patrick's satisfaction depend on how Sean divides the candies into two piles? Use this result to write a linear-time solution to the problem that correctly solves the Large input.

Because Patrick adds with XOR operations and we can use another XOR operation to check if the values of the two piles are equal, Patrick takes the following approach to deciding whether to cry:

Cry if and only if $(s_1 \wedge s_2 \wedge ... \wedge s_{n1}) \wedge (p_1 \wedge p_2 \wedge ... \wedge p_{n1})$ is greater than zero.

Because XOR operations are commutative, this is the equivalent of XORing the values of all candies in the bag together, regardless of how Sean decides to divide them. This means that Patrick's satisfaction will *never* depend on how Sean divides the candies, but rather only on the values of the individual candies. As such, we can check to see if Patrick is happy in linear time, and if he is, Sean can simply choose the smallest candy to put in Patrick's pile and keep the rest for himself.

```
In []: def patrick_addition(candies):
           total = 0
           for candy in candies:
               total ^= candy #XOR in Python
           return total

       def divide_candies(candies):
           if not patrick_addition(candies):
               return "NO"
           else:
               return sum(candies) - min(candies)
```

Optional No optional work this week.