code jam hello, world!

Practice Mode

Contest scoreboard | Sign in

Round 1B 2012

A. Safety in Numbers

B. Tide Goes In, Tide Goes Out

C. Equal Sums

Contest Analysis

Questions asked

Submissions

Safety in Numbers

10pt	Not attempted **2687/5608 users** correct (48%)
11pt	Not attempted **2008/2680 users** correct (75%)

Tide Goes In, Tide Goes Out

18pt	Not attempted **682/892 users** correct (76%)
18pt	Not attempted **619/670 users** correct (92%)

Equal Sums

6pt	Not attempted **2257/2531 users** correct (89%)
37pt	Not attempted **149/853 users** correct (17%)

Top Scores

Gennady.Korotkevich	100
bmerry	100
hansonw	100
marcina	100
ZhukovDmitry	100
random.johnnyh	100
yeputons	100
rng..58	100
pashka	100
mikhailOK	100

Contest Analysis

Overview | Problem A | Problem B | Problem C

The Small Input

If you want to forget about the large input and go straight for the small, then Equal Sums might look like a classic dynamic programming problem. Here is a sketch of one possible solution in Python:

```
def GetSetWithSum(x, target):
  if target == 0: return []
  return GetSetWithSum(x, target - x[target])

def FindEqualSumSubsets(S):
  x = [0] + [None] * (50000 * 20)
  for s in S:
    for base_sum in xrange(50000 * 20, -1, -1
      if x[base_sum] is not None:
        if x[base_sum + s] is None:
          x[base_sum + s] = s
        else:
          subset1 = GetSetWithSum(x, base_sum
          subset2 = GetSetWithSum(x, base_sum
          return subset1, subset2
  return None
```

The idea is that, in x, we store a way of obtaining every possible subset sum. If we reach a sum in two different ways, then we can construct two subsets with the same sum.

For the small input, this approach should work fine. However, x would have to have size 500 * 10^{12} for the large input. That is too big to work with, and you will need a different approach there.

The Large Input

The first step towards solving the large input is realizing that the statement is very misleading! Let's suppose you have just 50 integers less than 10^{12}. There are 2^{50} ways of choosing a subset of them, and the sum of the integers in any such subset is at most 50 * 10^{12} < 2^{50}. By the Pigeonhole Principle, this means that there are two different subsets with the same sum.

So what does that mean for this problem? Well, even if you had just 50 integers to choose from instead of 500, the answer would never be "Impossible". You need to find some pair of subsets that have the same sum, and there is a lot of slack to work with. The trick is figuring out how to take advantage of that slack.

The Birthday "Paradox" Method

The simplest solution is based on a classic mathematical puzzle: the birthday paradox.

The birthday paradox says that if you have just 23 people in a room, then some two of them probably have the same birthday. This can be surprising because there are 365 possible days, and 23 is much smaller than 365. But it is true! One good way to look at is this: there are 23 choose 2 = 253 pairs of people, and each pair of people has a 1/365 chance of having the same birthday. In particular, the *expected* (aka average) number of pairs of people with the same birthday is 253 / 365 = 0.693... Once you write it that way, it is not too surprising that the probability of having at least *one* pair matching is about 0.5.

It turns out this exact reasoning applies to the Equals Sums problem just as well. Here is a simple algorithm:

- Choose 6 random integers from **S**, add them up, and store the result in a hash set. (Why 6? We'll come back to that later...)
- Repeat until two sets of 6 integers have the same sum, then stop.

After choosing t sets, there will be t choose 2 pairs, and each set will have sum at most $6 * 10^{12}$. Therefore, the expected number of collisions would be approximately $t^2 / (12 * 10^{12})$. When t is around 10^6, this expectation will be near 1, and the reasoning from the birthday paradox says that we'll likely have our collision.

That's it! Since we can quickly generate 10^6 small subsets and put their sums into a hash-set, this simple algorithm will absolutely solve the problem. You may be worried about the randomness, but for this problem at least, you should not be. This algorithm is extremely reliable. By the way, this method will work on the small input as well, so you don't need to do the dynamic programming solution if you do not want to.

Note: There are many similar approaches here that can work. For example, in our internal tests, one person solved the problem by only focusing on the first 50 integers, and trying random subsets from there. The proof below works for this algorithm, as well as many others.

A Rigorous Proof

If you have some mathematical background, we can actually prove rigorously that the randomness is nothing to worry about. It all comes down to the following version of the birthday theorem:

Lemma: Let X be an arbitrary collection of N integers with values in the range [1, R]. Suppose you choose t + 1 of these integers independently at random. If $N \geq 2R$, then the probability that these randomly chosen integers are all different is less than $e^{-t^2 / 4R}$.

Proof: Let x_i denote the number of integers in X with value equal to i. The number of ways of choosing t + 1 distinct integers from X is precisely

$$\text{sum}_{(1 \leq i_1 < i_2 < \ldots < i_{t+1} \leq R)} [x_{i_1} * x_{i_2} * \ldots * x_{i_{t+1}}].$$

For example, if t=1 and R=3, the sum would be $x_1 * x_2 + x_1 * x_3 + x_2 * x_3$. Each term here represents the number of ways of choosing integers with a specific set of values. Unfortunately, this sum is pretty hard to work with directly, but Maclaurin's inequality states that it is at most the following:

$$(R \text{ choose } t+1) * [(x_1 + x_2 + \ldots + x_R) / R]^{t+1}$$
$$= (R \text{ choose } t+1) * (N/R)^{t+1}.$$

On the other hand, the number of ways of choosing *any* t + 1 integers out of X is equal to $N \text{ choose } t+1$. Therefore, the probability p that we are looking for is at most:

$$[(R \text{ choose } t+1) / (N \text{ choose } t+1)] * (N/R)^{t+1}$$
$$= R/N * (R-1)/(N-1) * (R-2)/(N-2) * \ldots * (R-t)/(N-t) * (N/R)^{t+1}.$$

Now, since $N \geq 2R$, we know the following is true for all $a \leq t$:

$$(R-a) / (N-a)$$
$$< (R - a/2)^2 / [R * (N-a)] \quad \text{(this is because } (R - a/2)^2 \geq R(R-a))$$
$$\leq (R - a/2)^2 / [N * (R-a/2)]$$
$$= (R - a/2) / N.$$

Therefore, p is less than:

$$R * (R - 1/2) * (R - 2/2) * \ldots * (R - t/2) / R^{t+1}.$$

It is now easy to check that $(R-a/2) * (R-t/2+a/2) \leq (R-t/4)^2$, from which it follows that p is also less than:

$$(R - t/4)^{t+1} / R^{t+1}$$
$$= (1 - t/4R)^{t+1}$$
$$< (1 - t/4R)^t.$$

And finally, we use the very useful fact that $1 - x \leq e^{-x}$ for all x. This gives us $p < e^{-t^2 / 4R}$ as required.

In our case, X represents the sums of all 6-integer subsets.

We have N = 500 choose 6 and R = 6 * 10^{12}. You can check that N ≥ 2R, so we can apply the lemma to estimate the probability that the algorithm will still be going after t+1 steps:

- If t = 10^6, the still-going probability is at most 0.972604477.
- If t = 5*10^6, the still-going probability is at most 0.499351789.
- If t = 10^7, the still-going probability is at most 0.062176524.
- If t = 2*10^7, the still-going probability is at most 0.000014945.
- If t = 3*10^7, the still-going probability is at most 0.00000000001.

In other words, we have a good chance of being done after 5,000,000 steps, and we will *certainly* be done after 30,000,000 steps. Note this is a mathematical fact, regardless of what the original 500 integers were.

Comment: What happens if you look at 5-element subsets instead of 6-element subsets? The mathematical proof fails completely because N < R. In practice though, it worked fine on all test data we could come up with.

A Deterministic Approach

The randomized method discussed above is certainly the simplest way of solving this problem. However, it is not the only way. Here is another way, this time with no randomness:

- Take 7,000,000 3-integer subsets of **S**, sort them by their sum, and let the difference between the two closest sums be d_1. Let X_1 and Y_1 be the corresponding subsets.
- Remove X_1 and Y_1 from **S**, and repeat 25 times to get X_i, Y_i and d_i for i from 1 to 25.
- Let Z = {d_1, d_2, ..., d_{25}}. Calculate all 2^{25} subset sums of Z.
- Two of these subset sums are guaranteed to be equal. Find them and trace back through X_i and Y_i to find two corresponding subsets of **S** with equal sum.

There are two things to show here, in order to justify that this algorithm works:

- ***S** will always have at least 7,000,000 3-integer subsets.* This is because, even after removing X_i and Y_i, **S** will have at least 350 integers, and 350 choose 3 > 7,000,000.
- *Z will always have two subsets with the same sum.* First notice that the subsets in the first step have sum at most 3 * 10^{12}, and so two of the sums must differ by at most 3 * 10^{12} / 7,000,000 < 500,000. Therefore, each d_i is at most 500,000. Now, Z has 2^{25} subsets, and each has sum at most 25 * 500,000 < 2^{25}, and so

it follows from the Pigeonhole Principle that two subsets have the same sum.

Other similar methods are possible too. This is a pretty open-ended problem!