Integers

1 Sum and product of a list

We often write a list in square brackets. For example, [John, Mary, Tim, Mary] is a list of people. In a list, the order matters.

For a list of numbers L, its sum is written $\sum L$, and its product $\prod L$. Here are some examples.

- $\sum[3, 4, 8] = 15$. Explanation: suppose I've bought a bag of 3 apples, a bag of 4 apples and a bag of 8 apples. Then I've bought 15 apples in total.
- $\sum[\,] = 0$. Explanation: suppose I've bought no bags. Then I've bought 0 apples in total.
- $\prod[3, 4, 8] = 96$. Explanation: suppose I'm ordering a 3-course meal at a restaurant that offers 3 starters, 4 mains and 8 desserts. Then I have 96 meal options. On the other hand, if the restaurant offers 3 starters, 0 mains and 8 desserts, then I have 0 meal options.
- $\prod[\,] = 1$. Explanation: suppose I'm ordering a 0-course meal. Then I have 1 meal option.

Generally, we start with the neutral element. To multiply a list of numbers, we start with 1, which is neutral for multiplication. Then we multiply by each number in the list.

This is why $a^0 = 1$, and it is also why $0! = 1$.

2 Introducing Integers

We've seen that $\mathbb{N}$ with addition and multiplication forms a commutative semiring. Let's now move to the set of integers, written $\mathbb{Z}$.

$$\mathbb{Z} \quad = \quad \{\ldots, -3, -2, -1, 0, 1, 2, 3\}$$

Any sum or product of integers is an integer. In fact, $\mathbb{Z}$ forms a *commutative ring*, i.e., not only do all the commutative semiring laws hold, but every number has an additive inverse:

$$a + (-a) \quad = \quad 0$$

From the commutative ring laws, we can derive lots of other properties, such as the *additive cancellation* law: if $a + x = a + y$ then $x = y$. (This also holds in $\mathbb{N}$, which isn't a commutative ring.) Another derived property is $-(-a) = a$.

3 Integer intervals

An *integer interval* is a set of consecutive integers. When mentioning such a set, we must specify whether the endpoints are included or not. The convention is that square brackets [,] indicate inclusion, and round brackets (,) indicate exclusion. For example:

- $[-3 \mathinner{..} 5)$ is the set of integers n such that $-3 \leqslant n < 5$. This is $\{-3, -2, -1, 0, 1, 2, 3, 4\}$.
- $[-3 \mathinner{..} 5]$ is the set of integers n such that $-3 \leqslant n \leqslant 5$. This is $\{-3, -2, -1, 0, 1, 2, 3, 4, 5\}$.
- $[23 \mathinner{..} 23)$ is the set of integers n such that $23 \leqslant n < 23$. This is the empty set.
- $[23 \mathinner{..} \infty)$ is the set of integers n such that $23 \leqslant n < \infty$. This is the infinite set $\{23, 24, 25, \ldots\}$.
- $\mathbb{Z}$ is $(-\infty \mathinner{..} \infty)$, and $\mathbb{N}$ is $[0 \mathinner{..} \infty)$.

4 Mod and div for integers

What are $-432 \text{ div } 100$ and $-432 \bmod 100$? To answer this question, we must solve

$$-432 \quad = \quad m \times 100 + r$$

where m is an integer and r is in the range $[0\,..\,100)$. The solution is $m = -5$ and $r = 68$. Thus

$$\begin{aligned} -432 \text{ div } 100 &= -5 \\ -432 \bmod 100 &= 68 \end{aligned}$$

Warning: in Java, `floorDiv` and `floorMod` behave like this, but other operations behave differently. Always check the documentation.

5 Representing integers on paper

There are two ways to represent an integer on paper.

One is the *sign-and-magnitude* notation, e.g. $+623$ or -127. For a nonzero integer, we first we give the sign $+$ or $-$, and then the magnitude, which is a positive natural number. Note that zero is a special case, since $+0 = -0$.

The other is *complement* notation, in a given base. Let's consider base ten. $+623$ is written as $\dot{0}623$, where the $\dot{0}$ is short for $\ldots 000$. Likewise 0 is represented as $\dot{0}$. What about -127? Let's do a subtraction:

$$\begin{array}{cccccc} \ldots & 0 & 0 & 0 & 0 & 0 \\ - & & & 1 & 2 & 7 \\ \hline \ldots & 9 & 9 & 8 & 7 & 3 \\ \hline -1 & -1 & -1 & -1 & -1 & \end{array}$$

So we get $\dot{9}873$. Here's another way of obtaining this:

$$\begin{aligned} -127 \bmod 10^0 &= 0 \\ -127 \bmod 10^1 &= 3 \\ -127 \bmod 10^2 &= 73 \\ -127 \bmod 10^3 &= 873 \\ -127 \bmod 10^4 &= 9873 \\ -127 \bmod 10^5 &= 99873 \end{aligned}$$

and so forth.

A negative number, when written in decimal complement notation, always begins $\dot{9}$. In binary complement notation, it always begins $\dot{1}$. In both of these notations, a nonnegative number begins $\dot{0}$.

6 Representing integers on a computer

Java has a type called `int`. It's not a *genuine* integer type, because an item of this type is allocated 32 bits. We interpret it as binary complement notation, with the first bit having a dot. This representation is called *32-bit two's complement*.

For example the number

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$$

is interpreted as binary complement notation:

$$\dot{0}111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$$

representing $2^{31} - 1$. Add 1 to this and we get

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$$

which is interpreted as binary complement notation

$$\dot{1}000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$$

representing -2^{31}. This format is called *32 bit two's complement*.

To summarize:

- The C type `unsigned int` uses 32 bits, and its range is $[0 \mathinner{..} 2^{32})$.
- The Java type `int` uses 32 bits, and its range is $[-2^{31} \mathinner{..} 2^{31})$.
- The Java type `long` uses 64 bits, and its range is $[-2^{63} \mathinner{..} 2^{63})$.

Each of these types behaves in a circular way. You will not be warned when your program crosses a "dangerous gap", so it's your responsibility to ensure this doesn't happen.

For example, in 2012, when the pop song Gangnam Style had been viewed 2^{31} times on YouTube, the displayed view count was -2^{31}, because the software used `int`. Subsequently the software was changed to use `long`. A spokesperson said: "We never thought a video would be watched in numbers greater than a 32-bit integer, but that was before we met [the singer] Psy."

What if your Java program needs bigger or smaller integers than `long` can accommodate? Consider using the `BigInteger` type. It's guaranteed to work for the range $[-2^m \mathinner{..} 2^m]$, where m is the largest possible `int` value, i.e., $2^{31} - 1$, which in Java is called `Integer.MAX_VALUE`. But outside this range, nothing is guaranteed, and your program's behaviour may depend on the implementation.