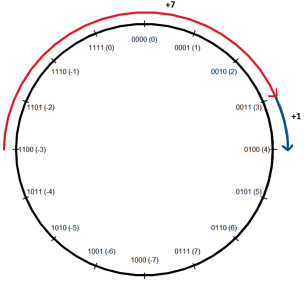
Algorithms and Data Structures for DS

Encoding
Machine numbers for $\mathbb{Z}$



Learning goals

- Signed magnitude representation
- One's complement
- Two's complement
- Integer overflow

MACHINE NUMBERS FOR $\mathbb{Z}$

Different options to represent $\mathbb{Z}$ on a computer:

- Signed magnitude representation
- Excess encoding
- One's complement
- Two's complement

Each representation has advantages and disadvantages regarding:

- Symmetry of the representable value range
- Uniqueness of representation
- Execution of arithmetic operations

SIGNED MAGNITUDE REPRESENTATION

- 31 bits for number + 1 sign bit

$$x = (-1)^{u_{32}} \sum_{i=1}^{31} u_i 2^{i-1}$$

	Bit u_i										
	sign	31	...	8	7	6	5	4	3	2	1
-1	1	0	...	0	0	0	0	0	0	0	1
0	1/0	0	...	0	0	0	0	0	0	0	0
1	0	0	...	0	0	0	0	0	0	0	1
51	0	0	...	0	0	1	1	0	0	1	1
		2^{30}	...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- Range in 32-bit: $-2^{31} + 1$ to $2^{31} - 1$
- Representation of zero not unique
- Addition/subtraction is cumbersome

MACHINE NUMBERS FOR $\mathbb{Z}$: EXCESS CODE

- Values are shifted by bias (so that they are not negative)

$$x = \sum_{i=1}^{32} u_i 2^{i-1} - 2^{31}$$

	Bit u_i										
	32	31	...	8	7	6	5	4	3	2	1
-2^{31}	0	0	...	0	0	0	0	0	0	0	0
-1	0	1	...	1	1	1	1	1	1	1	1
0	1	0	...	0	0	0	0	0	0	0	0
1	1	0	...	0	0	0	0	0	0	0	1
$2^{31} - 1$	1	1	...	1	1	1	1	1	1	1	1
	2^{31}	2^{30}	...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- Range in 32-bit: -2^{31} to $2^{31} - 1$
- Unique 0
- Still no simple addition/subtraction of binary numbers

ONE'S COMPLEMENT

- $-x$ represented by bitwise complement of x
- First bit is sign bit again

$$x = \sum_{i=1}^{31} u_i 2^{i-1} - u_{32}(2^{31} - 1)$$

	Bit u_i										
	32	31	...	8	7	6	5	4	3	2	1
-51	1	1	...	1	1	0	0	1	1	0	0
-1	1	1	...	1	1	1	1	1	1	1	0
-0	1	1	...	1	1	1	1	1	1	1	1
0	0	0	...	0	0	0	0	0	0	0	0
1	0	0	...	0	0	0	0	0	0	0	1
51	0	0	...	0	0	1	1	0	0	1	1
	$-(2^{31}-1)$	2^{30}	...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

ONE'S COMPLEMENT

- Let $\tilde{x}$ be the bitwise complement of x
- We check the correctness of the formula:

$$
\begin{aligned}
\tilde{x} &= \sum_{i=1}^{31} \tilde{u}_i 2^{i-1} - \tilde{u}_{32}(2^{31}-1) \\
&= \sum_{i=1}^{31} \underbrace{(1-u_i)}_{\text{complement}} 2^{i-1} - \underbrace{(1-u_{32})}_{\text{complement}} (2^{31}-1) \\
&= \sum_{i=1}^{31} 2^{i-1} - (2^{31}-1) - \sum_{i=1}^{31} u_i 2^{i-1} + u_{32}(2^{31}-1) \\
&= -1 + 2^{31} - (2^{31}-1) - (\sum_{i=1}^{31} u_i 2^{i-1} - u_{32}(2^{31}-1)) = -x
\end{aligned}
$$

ONE'S COMPLEMENT

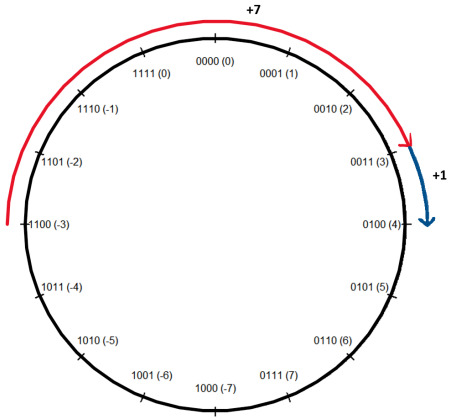
- Covered number range in 32-bit: $-2^{31} + 1$ to $2^{31} - 1$.
- Very easy conversion from positive to negative
- No unique 0
- Addition/subtraction works better here, but the sum must be corrected by adding the carry bit afterward

  Example: $7 - 3$ in 4-bit system:

	0111	|(7)
+	1100	|(-3)
	(1)0011	| Carry-Bit
+	0001	| add 1
	0100	|(4)

ONE'S COMPLEMENT

- Why adding a 1?



- Because of the 0 being represented twice, a 1 must be added when overflowing the 0

TWO'S COMPLEMENT

- In two's complement, negative numbers are formed by flipping the bits and then adding 1

Example: Conversion of -51_{10}:

	Bit u_i										
	32	31	...	8	7	6	5	4	3	2	1
$\lvert -51_{10} \rvert$	0	0	...	0	0	1	1	0	0	1	1
invert	1	1	...	1	1	0	0	1	1	0	0
add 1	1	1	...	1	1	0	0	1	1	0	1
	-2^{31}	2^{30}	...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

TWO'S COMPLEMENT

- Example: Two's complement

	Bit u_i										
	32	31	...	8	7	6	5	4	3	2	1
-2^{31}	1	0	...	0	0	0	0	0	0	0	0
-51	1	1	...	1	1	0	0	1	1	0	1
-1	1	1	...	1	1	1	1	1	1	1	1
0	0	0	...	0	0	0	0	0	0	0	0
1	0	0	...	0	0	0	0	0	0	0	1
51	0	0	...	0	0	1	1	0	0	1	1
$2^{31}-1$	0	1	...	1	1	1	1	1	1	1	1
	-2^{31}	2^{30}	...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

The coded number is then: $x = \sum_{i=1}^{31} u_i 2^{i-1} - u_{32} 2^{31}$

TWO'S COMPLEMENT

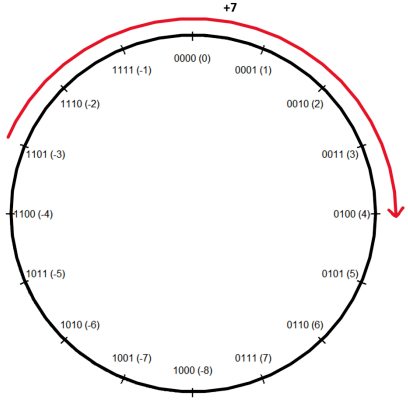
- Covered number range in 32-bit: -2^{31} to $2^{31} - 1$
- Harder to read but provides major advantages for the computer
- Unique representation of 0
- Addition and subtraction work as intended. Within the number range, two numbers can directly be added, ignoring the carry bit

  Example: $7 - 3$ in a 4-bit system:

```
      0111   |(7)
+     1101   |(-3)
-------------------
  (1)0100    |(4)
```

TWO'S COMPLEMENT

- Carry bit can be ignored here, as the representation of 0 is unique:



- Caution when leaving the number range:
- Example: 0011 (3) + 0101 (5) = 1000 (-8)

INTEGER OVERFLOW

- **Caution:** Arithmetic operations can result in **overflow**, a common programming error in languages like `C`, leading to undefined behavior (e.g., wraparound).

- **Example**: $(2^{31} - 1) + 1$.
  In a 32-bit two's complement representation, $(2^{31} - 1) + 1$ exceeds the representable range, as $(2^{31} - 1)$ is the largest possible value. Adding 1 causes an integer overflow, resulting in -2^{31}.

$(2^{31} - 1)$	01111111	11111111	11111111	11111111
$+1$	00000000	00000000	00000000	00000001
(-2^{31})	10000000	00000000	00000000	00000000

INTEGER OVERFLOWS - FROM WIKIPEDIA ENTRY

- *When Donkey Kong breaks on level 22 it is because of an integer overflow in its time/bonus. Donkey Kong takes the level number you're on, multiplies it by 10 and adds 40. When you reach level 22 the time/bonus number is 260 which is too large for its 8-bit 256 value register so it resets itself to 0 and gives the remaining 4 as the time/bonus - not long enough to complete the level.*

- *On 30 April 2015, the Federal Aviation Authority announced it will order Boeing 787 operators to reset its electrical system periodically, to avoid an integer overflow which could lead to loss of electrical power and ram air turbine deployment, and Boeing is going to deploy a software update in the fourth quarter.*

BINARY MULTIPLICATION

- Same in binary as decimal long multiplication

			1	0	1	
$\times$			1	1	0	
			0	0	0	(101×0)
$+$		1	0	1	0	$(101 \times 1$, shifted left by 1)
$+$	1	0	1	0	0	$(101 \times 1$, shifted left by 2)
	1	1	1	1	0	

- For neg. numbers in two's complement,
  convert them by inverting bits, then add 1
- Perform binary long multiplication on positive values
- If the result would be negative, convert it to two's complement

INTEGERS IN R

- Always 32-bit (also in 64-bit R), using two's complement

```
.Machine$integer.max
## [1] 2147483647

2^31 - 1
## [1] 2147483647

intToBits(-51L)
## [1] 01 00 01 01 00 00 01 01 01 01 01 01 01 01 01 01 01 01
## [19] 01 01 01 01 01 01 01 01 01 01 01 01 01 01

intToBits(51L)
## [1] 01 01 00 00 01 01 00 00 00 00 00 00 00 00 00 00 00 00
## [19] 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

INTEGERS IN R

- Int overflows are caught and set to NA

```
.Machine$integer.max + 1
## [1] 2147483648

str(.Machine$integer.max + 1)
## num 2.15e+09

str(.Machine$integer.max + 1L)
## Warning in .Machine$integer.max + 1L:
## NAs produced by integer overflow
## int NA
```

R ON 64-BIT SYSTEMS

- In 2010, 64-bit version released
- But: ints still encoded as 32-bit
- Largest integer s therefore about 2 billion
- When indexing vectors longer than this, R uses a trick
- Via doubles, ints can be represented reliably within $[-2^{53}, 2^{53}]$
- You can index with doubles now

```
c(1,2,3)[1.7]
## [1] 1
```

C DATA TYPES

Type	Explanation	Size (bits)	Range
short, short int, signed short, signed short int	Short signed integer type.	$\geq$16	[-32767, +32767]
unsigned short, unsigned short int	Short unsigned integer type.	$\geq$16	[0, 65535]
int, signed, signed int	Basic signed integer type.	$\geq$16	[-32767, +32767]
unsigned, unsigned int	Basic unsigned integer type.	$\geq$16	[0, 65535]
long, long int, signed long, signed long int	Long signed integer type.	$\geq$32	[-2147483647, +2147483647]
unsigned long, unsigned long int	Long unsigned integer type.	$\geq$32	[0, 4294967295]
long long, long long int, signed long long, signed long long int	Long long signed integer type.	$\geq$64	[-9223372036854775807, +9223372036854775807]

Table: `https://en.wikipedia.org/wiki/C_data_types`