Numerical Computing is Difficult

- integer types
  - byte (signed 8-bit)
  - short (signed 16-bit)
  - int (signed 32-bit)
  - long (signed 64-bit)
  - BigInteger (Immutable arbitrary-precision integers)
- floating point types
  - float (IEEE 32-bit floating point, with 23-bit mantissa and 8-bit exponent)
  - double (IEEE 32-bit floating point, with 52-bit mantissa and 11-bit exponent)

For the Inexperienced…

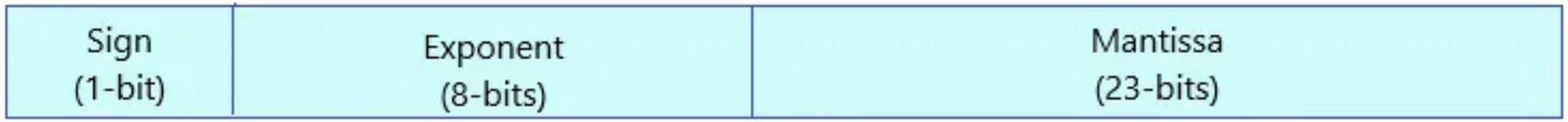
- (x+1) is not always greater than x
- (x+1) may be negative, even though x is positive
- (x+1) may equal x
- (x-1) may equal x
- MIN_VALUE and MAX_VALUE have different meanings for the integer types and the floating point types
- floating point is an approximation
- floating point cannot represent all fractions accurately

Numerical Computing is Difficult

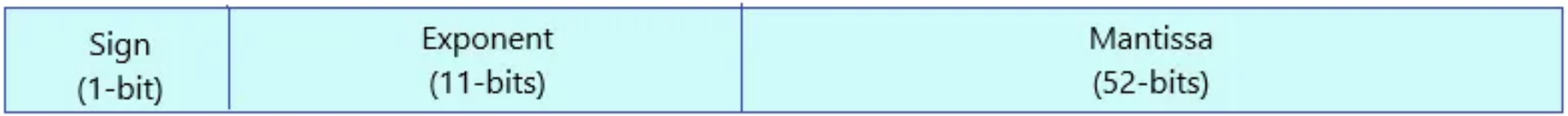
- integer types (byte, short, int, long)
- 8-bit wrapping examples shown below:
  - two's complement (signed)arithmetic:
    - 00000000 (binary) = 0 (decimal)
    - 00000001 (binary) = 1 (decimal)
    - 11111111 (binary) = -1 (decimal)
  - 01111111 (binary) = 127 (decimal)
    - 127 (decimal) + 1 (decimal) = 128 (decimal)
    - 01111111 (binary) + 1 (binary) = 10000000 (binary) = -128 (decimal)
  - 11111111 (binary) = -128 (decimal)
    - -11111111 (binary) = 11111111 (binary) = -128 (decimal)

Numerical Computing is Difficult

- floating point types (float and double)
- float has 32-bits of precision (the mantissa)



- double has 52-bits of precision (the mantissa)



Behaviour to be Aware Of

- Recurring (binary) values can't be represented:
  - 0.1 for example
  - Compare to 1/3≈0.33333 in decimal
- No wrap
  - float x+1.0F will give you x for large numbers
- Values run from
  - -Float.MAX_VALUE..Float.MAX_VALUE
  - -Double.MAX_VALUE..Double.MAX_VALUE
- The value closest to zero
  - Float.MIN_VALUE and Double.MIN_VALUE
- Infinity and –Infinity
- Not a number

Loss of Precision

- nextUp() and nextDown()
  - 16777215 is largest float until nextUp() is > 1.0
- System.out.println(16777216.0F-1.0F) -> 1.6777215E7
- System.out.println(16777216.0F+1.0F) -> 1.6777216E7
- float x=0.01F; float sum=0.0F;
  for (int i=0; i<100; i++) sum+=x; System.out.println(sum);
  - 0.99999934
- float x=0.01F; float y=100.0F; System.out.println(x*y);
  - 1.0

Testing Numerical Computing is Difficult

- For unbounded inputs, what are the BV's for the output?
  - e.g. int twice(int x); // returns 2*x
- For floating point, what are the EP's for the output?
  - e.g. float twice(float x); // returns 2*x
  - [Float.MIN_VALUE..Float.MAX_VALUE]
  - [Float.POSITIVE_INFINITY]
  - [Float.NEGATIVE_INFINITY]
  - [Float.NAN]
  - And what is the required precision?
    - All FP is an approximation, and precision loss increases as the value increases
- Never compare FP for equality – always for a range (required accuracy)

Testing Example1

- int avg(int x, int y) returns the average of x and y (rounded down)
- Input partitions:
  - x: Integer.MIN_VALUE..Integer.MAX_VALUE
  - y: Integer.MIN_VALUE..Integer.MAX_VALUE
- Output partitions:
  - Return value:
    - Can it be Integer.MIN_VALUE?
    - Can it be Integer.MAX_VALUE?

Partitions

- int avg(int x, int y) returns the average of x and y (rounded down)
- Input partitions:
  - x: [Integer.MIN_VALUE..Integer.MAX_VALUE]
  - y: [Integer.MIN_VALUE..Integer.MAX_VALUE]
- Output partitions:
  - Return value:
    - Can it be Integer.MIN_VALUE: yes (if x==y==Integer.MIN_VALUE)
    - Can it be Integer.MAX_VALUE : yes (if x==y==Integer.MAX_VALUE)
    - [Integer.MIN_VALUE..Integer.MAX_VALUE]

TCIs

- EP
  - 0 is special for any mathematical operation
  - Best to treat as [Integer.MIN_VALUE..-1][0][1..Integer.MAX_VALUE]
  - For x, y, and return value
    - Integer.MIN_VALUE/0
    - 0
    - Integer.MAX_VALUE/2
- BVA
  - For x, y, and return value
    - Integer.MIN_VALUE
    - -1
    - 0
    - 1
    - Integer.MAX_VALUE

Data/Test Cases

- Break our normal rules to avoid always having x==y
- And optimise to reduce number of test cases
- EP
  - x=Integer.MIN_VALUE/2, y=0, rv=Integer.MIN_VALUE/4
  - x=0, y=Integer.MAX_VALUE/2, rv=Integer.MAX_VALUE/4
  - x=Integer.MAX_VALUE/2, y=Integer.MIN_VALUE/2, rv=0

Data/Test Cases

- BVA
  - x=Integer.MIN_VALUE, y=-1, rv=(Integer.MIN_VALUE-1)/2
  - x=-1, y=0, rv=0
  - x=0, y=1, rv=0
  - x=1, y=Integer.MAX_VALUE, rv=(Integer.MAX_VALUE+1)/2
  - x=Integer.MAX_VALUE, y=Integer.MIN_VALUE, rv=0
  - rv=Integer.MIN_VALUE => x=Integer.MIN_VALUE, y=Integer.MIN_VALUE
  - rv=-1 => x=0, y=-2
  - rv=1 => x=2, y=0
  - rv=Integer.MAX_VALUE => x=Integer.MAX_VALUE, y=Integer.MAX_VALUE

Data/Test Cases - Problems

- x=1, y=Integer.MAX_VALUE, rv=(Integer.MAX_VALUE+1)/2
  - Can’t calculate (Integer.MAX_VALUE+1)/2 using int maths!!
  - **jshell> System.out.println((Integer.MAX_VALUE+1)/2);**
    -1073741824
- For int, can use long maths
  - **jshell> int v=(int)(((long)Integer.MAX_VALUE+1L)/2L);**
    v ==> 1073741824
- For long, have to be careful

Handling Long

- For long, have to be careful
  - Long.MAX_VALUE+x will wrap for x>0
  - So need to halve the values before adding
  - And then add 1 if x is odd to handle the ‘lost’ bit
  - ```
    jshell> long x=3L,y=5L; System.out.println((x/2L)+(y/2L));
      x ==> 3
      y ==> 5
      3
    ```
  - ```
    jshell> long x=3L,y=5L; System.out.println((x/2L)+(y/2L)+1);
      x ==> 3
      y ==> 5
      4
    ```

Testing Example

- float avg(float x, float y) returns the average of x and y
- Not stated so we assume standard java precision rules apply
  - Approximate values
  - Loss of precision
- EP
  - x=Float/MAX_VALUE/2.0F, y=0.0F
  - nextDown(Float.MAX_VALUE/4.0F)<=rv<=nextUp(Float.MAX_VALUE/4.0F)
- Or perhaps if the accuracy is specified as accurate to 1%
  - 0.99F*Float.MAX_VALUE/4.0F <= rv <= 1.01F*Float.MAX_VALUE/4.0F

Testing Example

- ```
  jshell> System.out.println(
                  Math.nextDown(Integer.MAX_VALUE/4F)+","+
                 (Integer.MAX_VALUE/4F)+","+
                  Math.nextUp(Integer.MAX_VALUE/4F));
  ```
- ```
  5.3687088E8,5.368709E8,5.36871E8
  ```