① LCV?
② scope of LCV
③ LCV = ?

## Loop Design Issues, contd.

⑥ What happens at edge cases ??

C, C++, Java: int uses 32-bits, 2-s complement representation to store integer values.
In 32 bits ∴ we can store values in range −2,147,483,648 .. 2,147,483,647

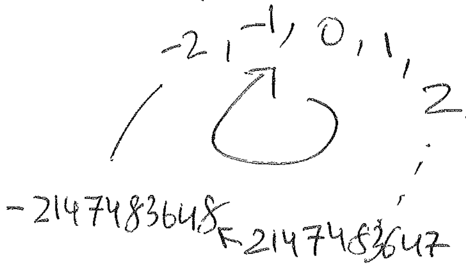
Consider the loop

```
for (int i = ___; i <= 2147483647; i++){
    | ≡
}
```

Edge Case: Suppose i = 2147483647.
Then the loop condition will be true.
The loop body will be executed.
Then in update, i will be incremented.
Q. What is the incremented value of i ??

→ 2147483647 + 1 = 2147483648 ← This # cannot be represented!!

So what happens ???




That is when i = 2147483647 is incremented, it results in i = −2147483648
∴ i will again be <= 2147483647 and so on. It will be an infinite loop!!

limits: C/C++: INT_MIN, INT_MAX
Java: Integer: MAX_VALUE, MIN_VALUE.

Q. What happens in Python ???

## Loop Design Issues, cont'd

(i) Does the PL allow iteration over user-defined types??

Java `int x[] = new int[N];`

we can write:

```
for (int a : x) {
  |  a takes x[0], x[1], .. x[N-1]
}
```

BUT YOU cannot do a = ____
only use the value

Now, what about iterating over LinkedLists, Binary Trees, etc.

In Java, one can define an Iterator object.

e.g.

```
public class BinaryTree<T> implements Iterator {
  |
}
```

Once defined, we can do:

```
BinaryTree<Person> Ptree;

for (Person p : Ptree) {
  —— p ——
}
```

Also, Arraylist has Iterator built-in

```
Arraylist<Place> places = ____

for (Place p : places) {
  |  ==
}
```

Python: all sequence objects ~~can~~ have iterators (iter object)

```
for <variable> in <sequence>:
    ≡
```

e.g. colleges = ["Bryn Mawr", "Haverford", "Swarthmore"]

```
for college in colleges:
    print(college)
    #Also...                    # strings are also sequences
    for l in college:
        print(l)
```

Extra: Defining Iterators for user-defined objects.
Two ways:

1. Implement Iterable interface
   `public Iterator iterator();` //required.

2. Iterator interface
   ```
   public boolean hasNext();
   public E next();
   public void remove();
   ```

e.g. we can also do this w/ ArrayList

```
ArrayList<Place> places = ...
Iterator<Place> iter = places.iterator();
while (iter.hasNext()) {
    Place p = iter.next();
    ≡
}
```

or, just do

```
for (Place p : places) {
    ≡
}
```

## Control flow: Recursion ≡ Iteration.

e.g. ① $sum = \sum_{i=1\ (low)}^{N\ (high)} i = 1 + 2 + \dots + N$

### Recursive

```
int sum(int low, int high) {
        if (low == high)
            return low;
        else
            return low + sum(sum (low+1, hight));
} //sum()
```

call: s = sum(1, N)

trace

sum(1, 3) = 1 + sum(2, 3)

6 = 1 + 5; sum(2, 3) = 5 = 2 + sum(3, 3); sum(3, 3) = 3

### Iterative

```
int sum (int low, int hight) {
      int result = 0;
      for (int i = low; i <= hight; i++)
              result += i;
      return result;

} //sum()
```

## Recursion

**Example 2:** gcd(a,b)

Recursive:

```
int gcd(int a, int b) {
    if (a==b)
        return a;
    else if (a > b)
        return gcd(a-b, b);
    else
        return gcd(a, b-a);
} //gcd()
```

Iterative:

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
} //gcd()
```

③ **Factorial:**

$0! = 1$

$n! = n * (n-1)!$

```
int factorial(int n) {
    if (n==0)
        return 1;
    else
        return n * factorial(n-1);
} //factorial()
```

```
int factorial(int n) {
    int result = 1;
    for (int i=1; i<=n; i++)
        result *= i;
    return result;
} //factorial()
```

**Tail Recursion:** when the last instruction (a return) is the recursive call.

All of the above are tail recursive functions.

Compilers can do Tail Call Optimization (TCO) so there is no overhead compared to iterative version!

# Data Types & Type Systems

1. What values can we compute with
2. Facilities for defining new types
3. Rules for type checking

Data Types are typically
- pre-defined/builtin in the design of PL
- user-defined.

## What is a data type?

Every data type has

- a name
- a set of values
- a set of operations

## Pre-defined Types

- Numbers
  - integers
  - floating point numbers
  - Complex Numbers?
- pointers →
- characters
- booleans
- enumerated types
- subrange types

scalar types (variable contains ONE value)

↕

composite or aggregate types

- records/structs
- arrays
- strings
- sets
- hashtables/dictionaries
- lists
- files

→ • image • table • etc..

# Basic Numeric Types

1. What is the internal representation used?
2. what memory size (bytes) is used to store a value?
3. What operations are available?
4. What happens when there is an over/underflow during runtime?

There is a tension between
- PL definition
- its implementation (compiler/interpreter)

Every aspect should be clearly spelled out in the reference manual.

## Integers

names

int x; C/C++/Java

var x int Go — also: int16 int32 int64 etc.

var x : integer Pascal

python is int

operations: +, -, *, /, % (mod) Python also has: **

Internal Representation: Binary, 2-s complement

8 bits

e.g $(0000\ 0101)_2 = 5$

$(1111\ 1011)_2 = -5$

n-bits:

range is $-2^{n-1} \,..\, 2^{n-1}-1$

n=16 bits $-2^{15} \,..\, 2^{15}-1 \Longrightarrow -32768 \,..\, 32767$

$\therefore 32767+1 = -32768$

## Integer types in PLs

| C/C++ | Java | Python | Go |
|---|---|---|---|
| byte (8) | byte (8) | int | int8, uint8 |
| short (16) | short (16) | ↓ | int16, uint16 |
| int (32) | int (32) | ↓ | int32, uint32 |
| long (64) | long (64) | unlimited | int64, uint64 |

also unsigned (byte, short, int, long) → $0..2^n-1$ range

Go: int, uint – typically 32-bits but it is machine dependent

## Floating Point Numbers

Names:

| C/C++/Java | Python | Go |
|---|---|---|
| float (32) | float | float32 |
| double (64) | | float64 |

Python float → implemented using C float — if you need higher precision???

Representation: All use the IEEE 754 Standard

Most computers use Floating Point Processing Units (FPUs) ⟷ (GPUs)

precision issues: 1/3 cannot be exactly represented!

Python: 1.0/3 = 0.3333333333333333 (16)

try a = 1.0 + 1.0 + 1.0
a = 3.000000000000000004 (15)

Also, try 1.1 + 2.2 = 3.3?? No
Python yields 3.3000000000000003

try 0.1 + 0.1 + 0.1 − 0.3
5.55 - - - $10^{-17}$