(B) Good Hash Functions

approximately satisfies SIMPLE UNIFORM HASHING:
→ each key is equally likely to hash to any of the m slots
This is tough because while we know what sort of keys we'll be adding (SIDs, SSNs, etc), but we don't know the specific subset of keys we'll be using

ASSUMPTION: keys are natural numbers.
Given a string (say) we'll translate it into a #:
"pt" p = 112, t = 116 (in ASCII), so we could do $(112 \cdot 128) + 116 = 14,452$
(better to 'shift' letters, so that key("pt") != key("tp");

SPEED: we also need the hash function to be (relatively) fast

- <u>Division Method</u>: hash(int key) {
  return key % tableSize; }

SPEED: this requires 1 cpu 'division' instruction, plus overhead for function call (which can be inlined in some languages
FYI: We'll need to be careful about which values we choose for tableSize.
→ NOT a power of 2: if → $= 2^p$, the we'll get the p low-order bits of key
(Similarly, if the keys are decimal numbers, then avoid using a power of 10. With table size = 1,000, the key 172,791,<u>124</u> → <u>124</u>)
Good values for tableSize: prime #, not to close to a power of 2

- **Multiplication Method:** double constantMultiplier = // >0, and <1 ("A")

```
public int hash(int key)
{   double dKey = (double) key
    double percent = constantMultiplier * dKey;      //①
    percent = percent - (int)percent;//get % part    ②
    return Math.floor(percent * tableSize);
}
```

(④ Math.floor; ③ percent * tableSize)

NUTSHELL: Take a constant value that is a % (i.e., it's between 0 and 1, not including 0 or 1), and multiply the key by that value ①. Then, remove the integer part, and take the decimal part ②. Treat that number as a percentage, and multiply the tableSize by it ③. In order to make sure you've got a whole # that's inside the table, round down ④

SPEED: Not too bad. Several operations, but each one is simple

FYI: Value of tableSize isn't critical

What to choose for "A"? "Knuth suggests that 0.6180339887* should work reasonably well"

- **Universal Hashing**
  - With the other methods, given a bad set of input data, lots of keys may hash to the same slot
    Example: symbol table of a compiler
  - What about creating a COLLECTION OF HASH FUNCTIONS, and then randomly choosing 1 to use @ table creation time (i.e., in the constructor)

NOTE: we won't look at any implementation details around this