

Data Representation and Compression



Exam 1?

Announcements

- The first lab exam is Monday during the lab session.
 - Sample exam on web site
 - Practice problems (with soln) are on the web site.
 - Python tutors will also help
- PA6 and OLI Data representation over the weekend

Lingering questions...

- Data Structures
- Arrays
- Linked Lists
- Hash Tables
- Associative Arrays

Key Point:

- Data needs to be stored in physical memory
- How we organize data in memory has consequences
- In this class, you are not implementing data structures – you are taking advantage of python's implementations...
- ...but you still need to understand (and make decisions) about these data structures.

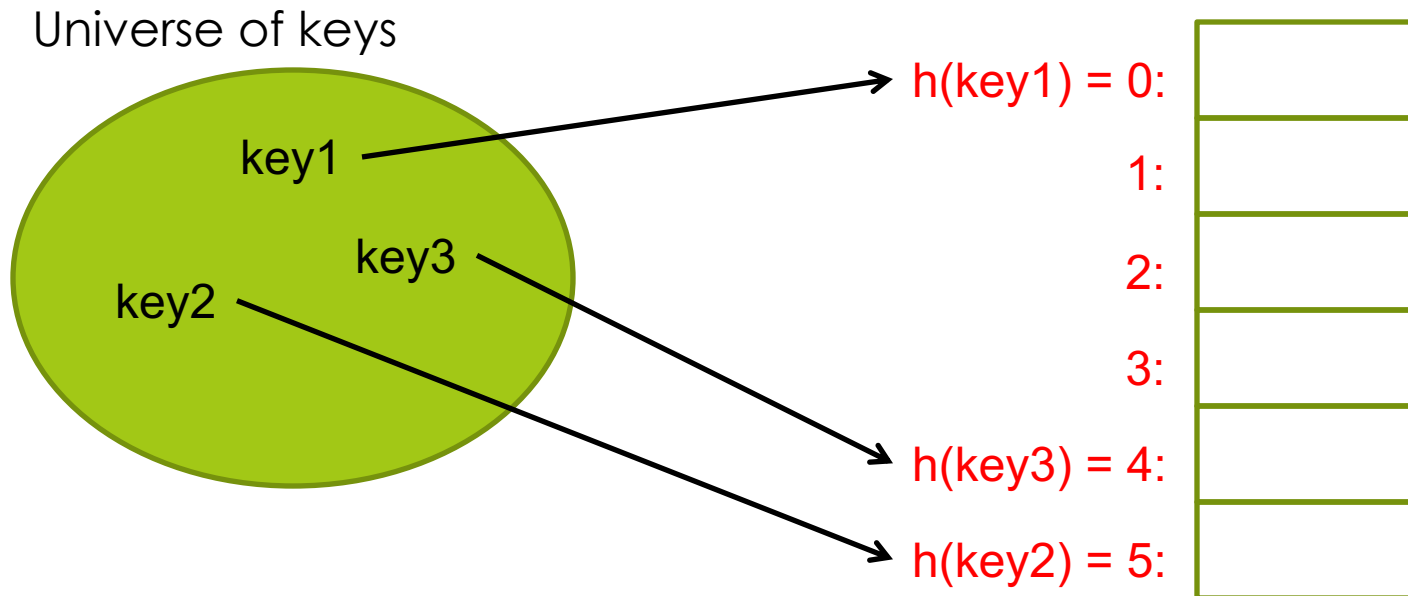
Recall Arrays and Linked Lists

	Advantages	Disadvantages
Arrays	Constant-time lookup (search) if you know the index	Requires a contiguous block of memory
Linked Lists	Flexible memory usage	Linear-time lookup (search)

Hashing tables are one approach to exploit the advantages of arrays and linked lists (to improve search time in dynamic data sets)?

Hashing

- A “hash function” $h(\text{key})$ that maps a key to an array index in $0..k-1$.
- To search the array `table` for that key, look in `table[h(key)]`



A hash function h is used to map keys to hash-table (array) slots. Table is an array bounded in size. The size of the universe for keys may be larger than the array size. We call the table slots buckets.

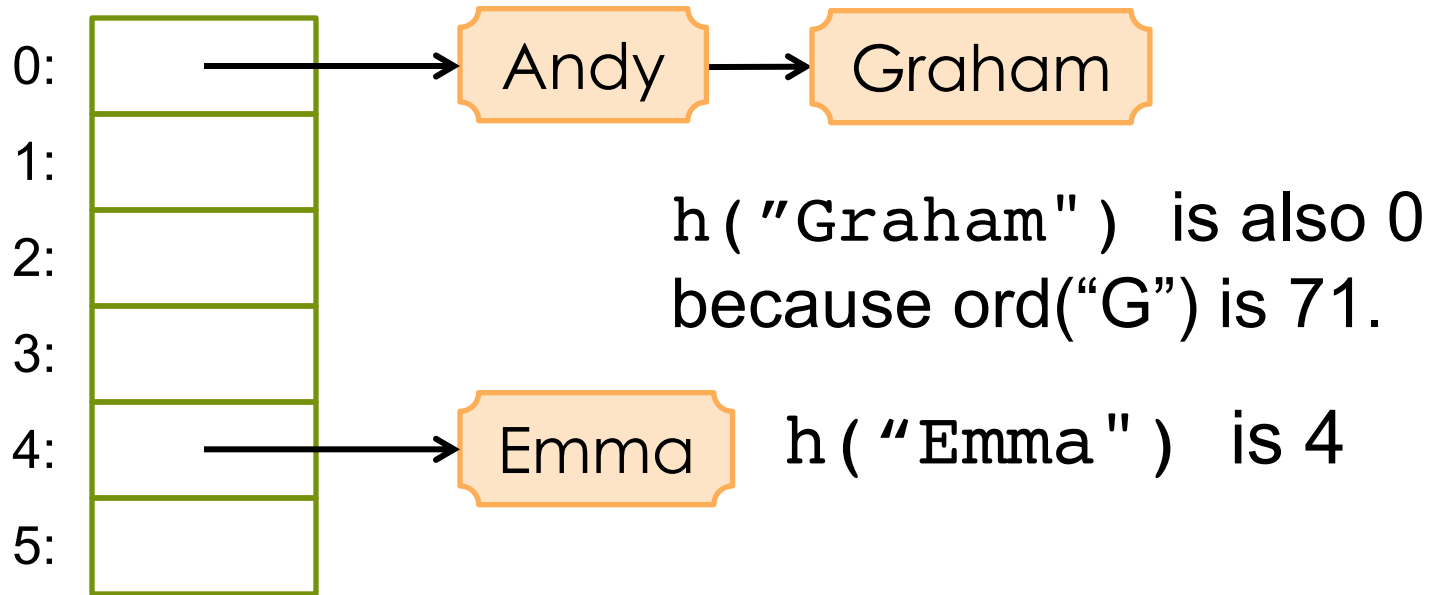
Example: Hash function

- Suppose we have (key,value) pairs where the key is a string such as (name, phone number) pairs and we want to store these key value pairs in an array.
- We could pick the array position where each string is stored based on the first letter of the string using this hash function:

```
def h(str):  
    return (ord(str[0]) - 65) % 6
```

Note $\text{ord}('A') = 65$

Add Element "Graham"



In order to add Graham's information to the table we had to form a link list for bucket 0.

Requirements for the Hash Function $h(x)$

- Must be fast: $O(1)$
- Must distribute items roughly uniformly throughout the array, so everything doesn't end up in the same bucket.

What's A Good Hash Function?

- For strings:
 - Treat the characters in the string like digits in a base-256 number.
 - Divide this quantity by the number of buckets, k .
 - Take the remainder, which will be an integer in the range $0..k-1$.

Fancier Hash Functions

- How would you hash an integer i ?
 - Perhaps $i \% k$ would work well.
- How would you hash a list?
 - Sum the hashes of the list elements.
- How would you hash a floating point number?
 - Maybe look at its binary representation and treat that as an integer?

Summary of Search Techniques

Technique	Setup Cost	Search Cost
Linear search	0, since we're given the list	$O(n)$
Binary search	$O(n \log n)$ to sort the list	$O(\log n)$
Hash table	$O(n)$ to fill the buckets	$O(1)$

Associative Arrays

- Hashing is a method for implementing associative arrays. Some languages such as Python have associate arrays (**mapping** between keys and values) as a built-in data type.
- Examples:
 - Name in contacts list => Phone number
 - User name => Password
 - Product => Price

Dictionary Type in Python

This example maps car brands (*keys*) to prices (*values*).

```
>>> cars = {"Mercedes": 55000,  
            "Bentley": 120000,  
            "BMW": 90000}
```

```
>>> cars["Mercedes"]
```

```
55000
```

Keys can be of any **immutable** data type.

Dictionaries are implemented using hashing.

Iteration over a Dictionary

```
>>> for i in cars:  
    print(i)
```

```
BMW  
Mercedes  
Bentley
```

Think what the loop variables are bound to in each case.

```
>>> for i in cars.items():  
    print(i)
```

```
("BMW", 90000)  
("Mercedes", 55000)  
  
("Bentley", 120000)
```

Note also that there is no notion of ordering in dictionaries. There is no such thing as the first element, second element of a dictionary.

```
>>> for k,v in cars.items():  
    print(k, ":", v )
```

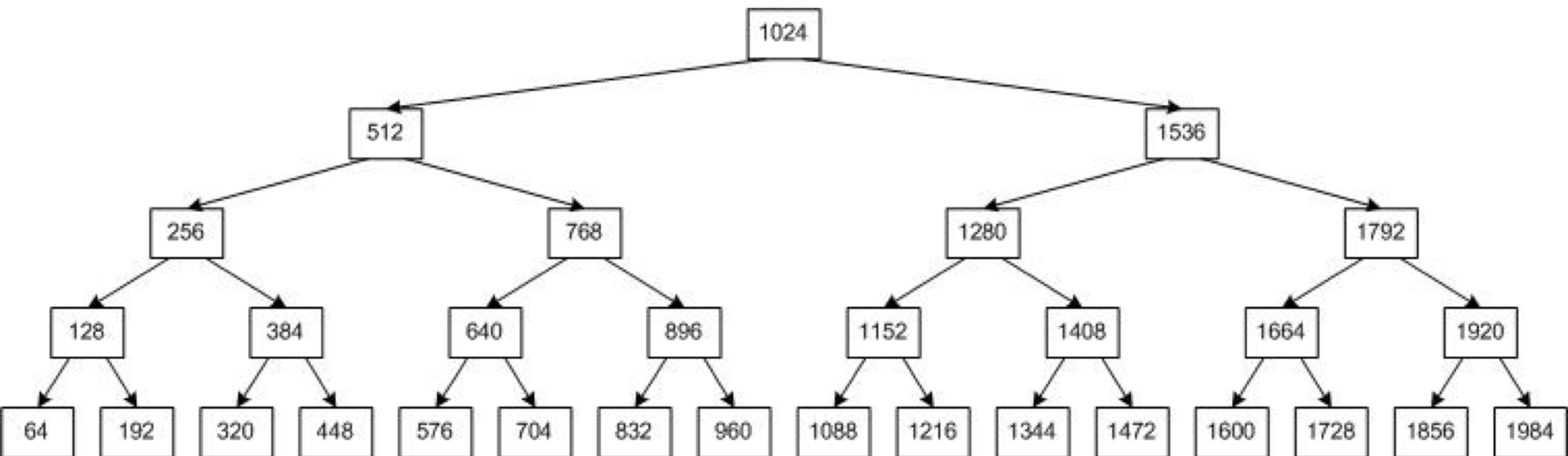
```
BMW : 90000  
Mercedes 55000  
Bentley : 120000
```


Some Dictionary Operations

- ▣ `d[key] = value` -- Set `d[key]` to `value`.
- ▣ `del d[key]` -- Remove `d[key]` from `d`. Raises a an error if `key` is not in the map.
- ▣ `key in d` -- Return `True` if `d` has a key `key`, else `False`.
- ▣ `items()` -- Return a new view of the dictionary's items ((`key`, `value`) pairs).
- ▣ `keys()` -- Return a new view of the dictionary's keys.
- ▣ `pop(key[, default])` If `key` is in the dictionary, remove it and return its value, else return `default`. If `default` is not given and `key` is not in the dictionary, an error is raised.

Source: <https://docs.python.org/>

Left – Node - Right

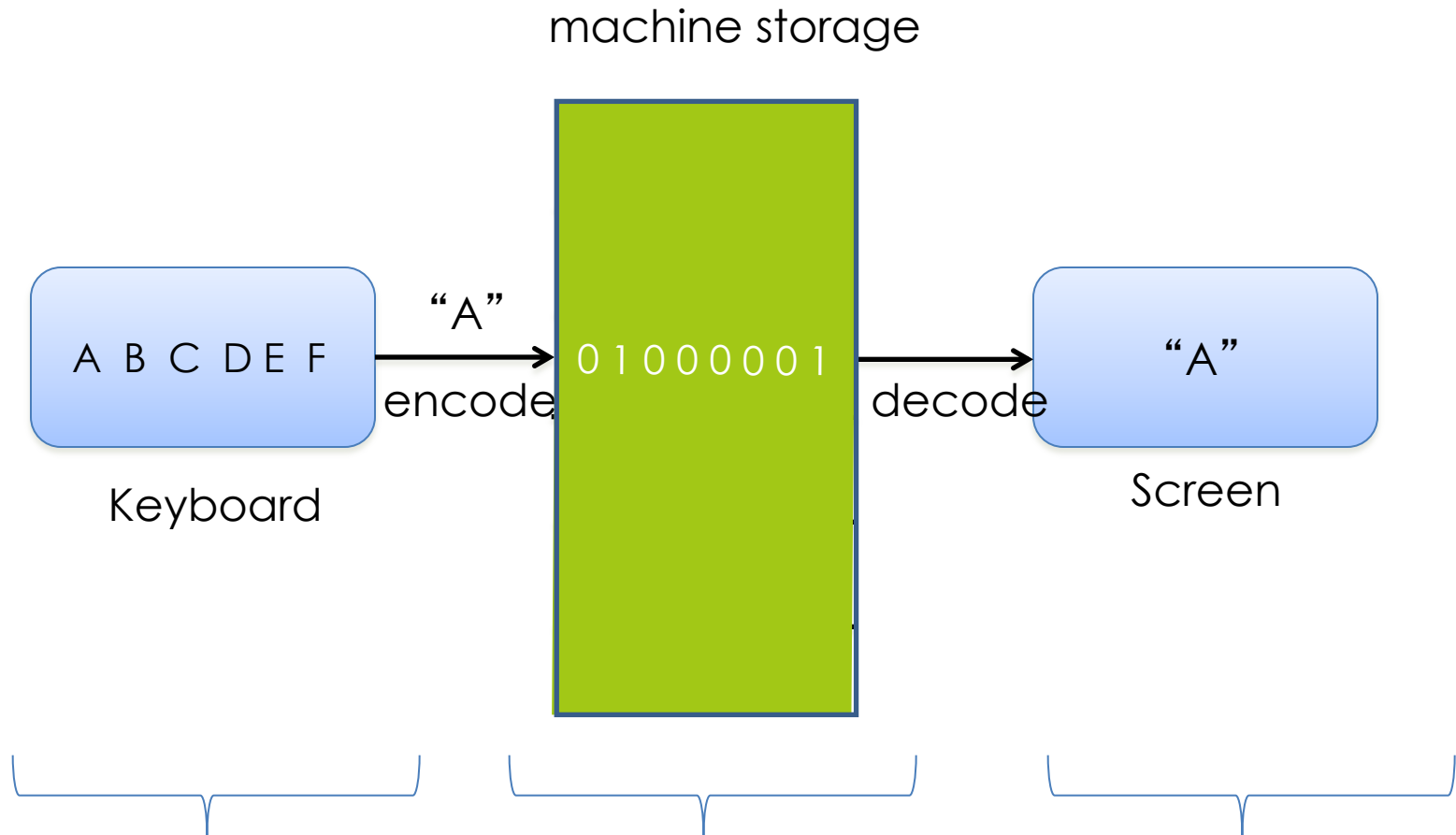


Representation

- We use computers to model i.e. *represent*, things in the real world:
 - Numbers, pictures, music, climate, markets...
- Three topics:
 - Representing numbers
 - Exploiting redundancy in representation (compression)
 - Representing images and sound

First, what do we mean by
Representation?

Representing Data



External representation Internal representation External representation

Digital Data


- ❑ Inside the digital machine it's all just
 - ❑ **binary** physical states (high or low voltages, etc.)
 - ❑ which we **interpret** as bits (1s and 0s)

- ❑ In turn we interpret these bits as representing data such as integers, real numbers, text, ...

- ❑ Machine storage is finite and divided into fixed-size chunks of bits
 - ❑ bytes, usually 8 bits
 - ❑ words, usually 64 or 32 bits
 - ❑ machine storage capacity usually expressed as number of bytes or words
 - ❑ loosely speaking: “memory size”

Types interpret bits

- ▣ a 32-bit "word" might be
1100 1100 1011 0111 0000 0000 0000 0000
- ▣ what this means depends on the machinery to interpret it, could be (**explore with 0xED**)

Type	Interpretation
"Raw" bits	1100 1100 1011 0111 0000 0000 0000 0000
Floating point number	6.59339 X 10 ⁻⁴¹
String (Unicode UTF-16)	책
RGB pixel color	
Little-endian integer	47052

Fundamental Issue: Information Capacity

# bits	Possible values								# possible values
1	0	1							2
2	00	01	10	11					4
3	000	001	010	011	100	101	110	111	8
4	0000	0001	0010	0011	0100	0101	0110	0111	16
	1000	1001	1010	1011	1100	1101	1110	1111	

$$2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16$$

Hmmm...could it be?

Yes, **k bits can represent 2^k different values.**

Today

- Numerals are not numbers!
 - place-value representations

- Positive and negative integers

- Real numbers and floating-point representations

You should be able to



- Count in unsigned binary
0, 1, 10, 11, 100, ...
- Add in binary and know what overflow is
- Determine the sign and magnitude of an integer represented in two's complement binary
- Determine the two's complement binary representation of a positive or negative integer

numerals are not numbers!

don't be drawn like moths to the flame of meaning*:

* Geoffrey Pullum

Numbers: semantics (quantities) versus syntax (numerals)

	Semantics	Syntax
What is it?	Our idea of quantity	How we write our idea of quantity
What is it good for?	Insight	Calculation, communication, computation
Example	 	II (Roman numeral) 2 (decimal Arabic numeral) 10 (binary numeral) – all with the same semantics!

machines don't have ideas!

only syntax!

Numerals aren't numbers, but

- ...to communicate a number (quantity), I have to write *something*
- I will write numbers (quantities) as ordinary base-10 numerals (or sometimes as words)

place-value syntax of numerals

representing non-negative integers (0, 1, 2, 3, ...)

Place-value numerals (base 10)

□ The *numeral* we write: 15627

□ What it means:

$$7 \times 10^0 + 2 \times 10^1 + 6 \times 10^2 + 5 \times 10^3 + 1 \times 10^4$$

□ **Problem:** electronic circuitry for base-10 arithmetic is slow.

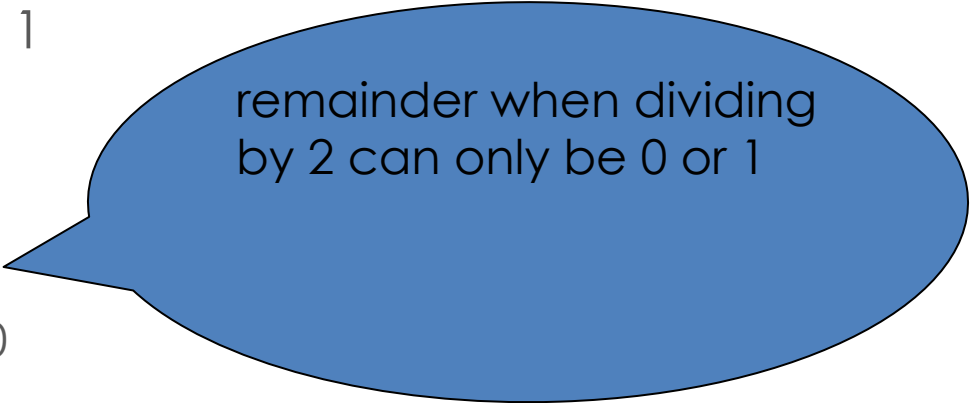
□ **Solution:** use place-value numerals, but in base 2–*binary notation*

Place-value numerals in general

- Choose a number b for the **base** or **radix**
- Choose list of **digits**, there must be b of them
 - **base 10 example: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
 - **base 2 example: 0, 1**
 - **base 16 example: 0, 1, ..., 9, A, B, C, D, E, F**
- To represent a quantity n in base b
 - integer divide n by b with remainder r (a **digit**)
 - repeat until the quotient is zero
 - the remainders are the digits in reverse order

Binary place-value example

- Base two, digits 0 and 1
- To represent “six”:
 - $6 // 2 = 3$ remainder 0



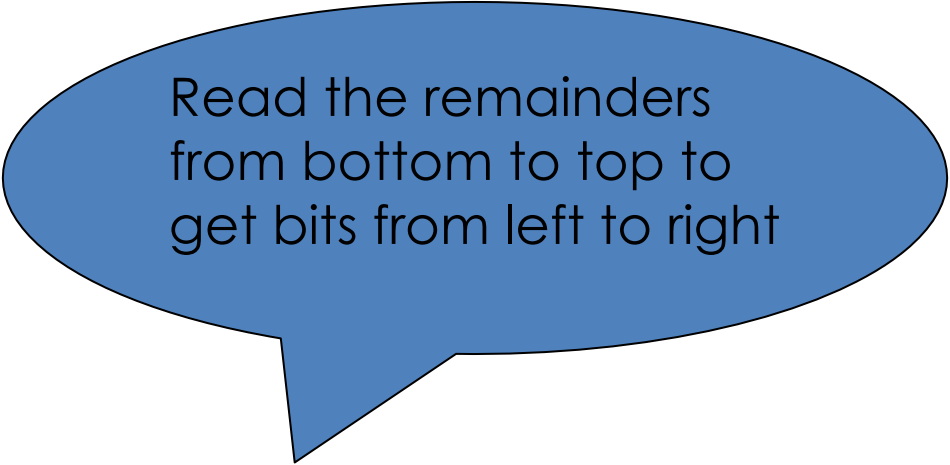
remainder when dividing
by 2 can only be 0 or 1

Binary place-value example

- Base two, digits 0 and 1
- To represent “six”:
 - $6 // 2 = 3$ remainder 0
 - $3 // 2 = 1$ remainder 1

Binary place-value example

- Base two, digits 0 and 1
- To represent “six”:
 - $6 // 2 = 3$ remainder 0
 - $3 // 2 = 1$ remainder 1
 - $1 // 2 = 0$ remainder 1



Read the remainders
from bottom to top to
get bits from left to right

Binary numeral: 110

- What it means:
 $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = \text{“six”}$

Information Capacity and Range

- Remember: k bits can represent 2^k different things
- So k -bit binary numerals represent $0 \dots 2^k - 1$
 - For $k = 3$,

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

Ranges for typical computer “word” sizes

<u>bits</u>	<u>minimum</u>	<u>maximum</u>
8	0	$2^8 - 1$ (255)
16	0	$2^{16} - 1$ (65,535)
32	0	$2^{32} - 1$ (4,294,967,295)
64	0	$2^{64} - 1$ (18,446,744,073,709,551,615)

binary arithmetic

some familiar operations

Counting in binary

Binary numerals

- ▣ 0
- ▣ 1
- ▣ 10
- ▣ 11
- ▣ 100
- ▣ 101
- ▣ 110
- ▣ 111
- ▣ 1000
- ▣ 1001
- ▣ 1010
- ▣ 1011

Decimal equivalents

- ▣ 0
- ▣ 1
- ▣ 2
- ▣ 3
- ▣ 4
- ▣ 5
- ▣ 6
- ▣ 7
- ▣ 8
- ▣ 9
- ▣ 10
- ▣ 11

Addition and Multiplication Tables

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

Binary Arithmetic

- All the familiar methods work, but with only 1 and 0 for digits
- $1 + 1 = 10$, $10 - 1 = 1$, $10 + 1 = 11$, ...
- Example:

```
  1  1
   1010
+ 1010
-----
 10100
```

Notice: we need more bits for the answer than we did for the operands.

Overflow: the first difficulty

- Machine word only has k bits for some **fixed** k !
- If k is 4, then we have **overflow** in the following:

```
  1  1
  1010
+1010
-----
  10100
```

- The machine retains only 0100 (the “least significant” bits), so $(n+n) - n$ **not** always equal to $n + (n - n)$

Modular Arithmetic

- ❑ Dropping the overflow bit is **modular arithmetic**
- ❑ We can carry out any arithmetic operation modulo 2^k for the precision k . The example again for precision 4:

binary	decimal
1 0 1 0	= 10
+ 1 0 1 0	= 10
(1) 0 1 0 0	= 20 = 4 mod 16

overflow can be ignored or signaled as an error

negative integers

representing all the integers...

Representing a sign +/-

- A natural idea: reserve one of the bits to stand for a sign.
- E.g., 0 could stand for + and 1 could stand for –
 - unsigned “ten” is 1010
 - so “negative ten” would be 11010
- But someone had a cleverer idea...
 - first, we'd like to avoid “two zeroes”: +0 and -0
 - second, we'd like the same machinery to work for addition and subtraction

Two's Complement Negative Numbers

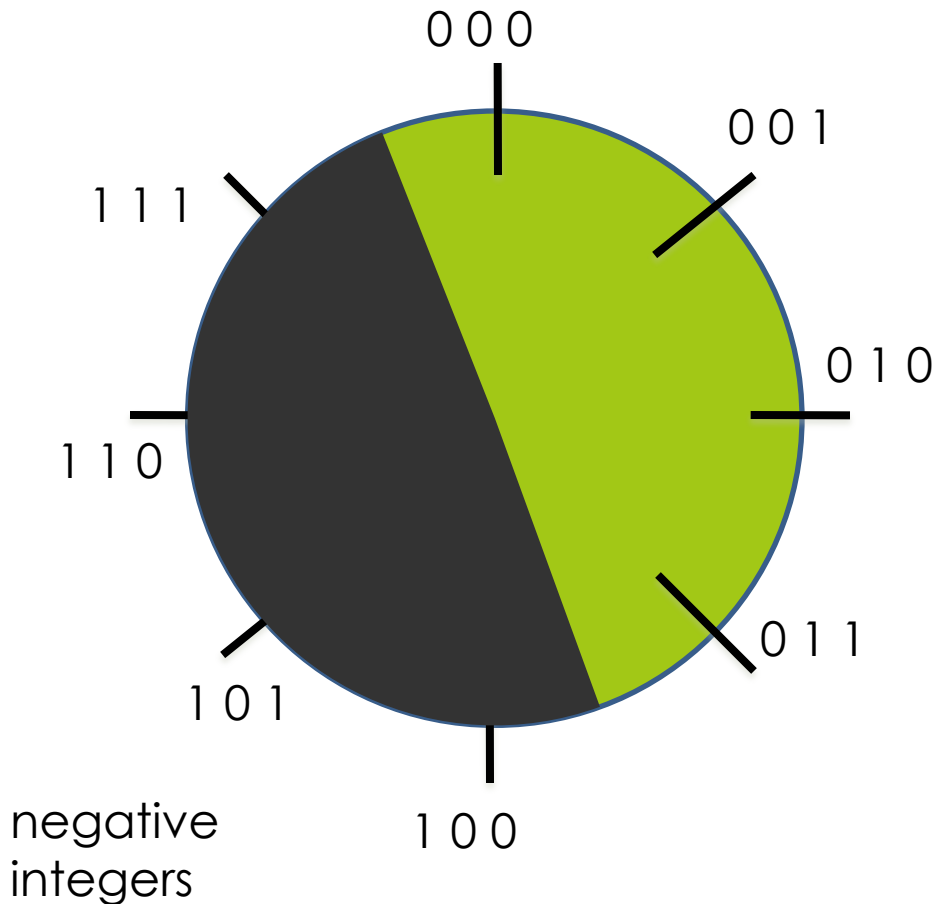
- ▣ A clever approach based on modular arithmetic
- ▣ Remember, with k bits, we do arithmetic mod 2^k
- ▣ We define negative numbers as *additive inverse*: $-x$ is the number y such that $x + y = 0 \pmod{2^k}$ – this is the **two's complement of x**
- ▣ Example with 4 bits: if 1 is 0001, what is -1 ?

<i>carry bits</i>		<i>1</i>	<i>11</i>	<i>111</i>	<i>1111</i>	
	0001	0001	0001	0001	0001	0001
+	????	+????1	+???11	+?111	+1111	+1111
	----	----	----	----	----	----
	0000	????0	???00	?000	0000	10000

representation for -1

↑ modular arithmetic discards overflow

All two's complement integers using 3 bits, arithmetic mod 8



Bit pattern	Decimal value
0 0 0	0
0 0 1	+ 1
0 1 0	+ 2
0 1 1	+ 3
1 0 0	- 4
1 0 1	- 3
1 1 0	- 2
1 1 1	- 1

Adding + n to - n gives 0
For example: 011 + 101 = 000

Great! but how do we “read” two’s complement integers?

- **Sign:** look at leftmost bit
 - **1 means negative, 0 means positive**
e.g. with four bits 1010 represents a negative number
- **Magnitude:** if negative, compute the two’s complement
 - flip each bit (one’s complement)
e.g. flip 1010 to get 0101
 - then add 1
e.g. $0101 + 0001 = 0110$, or
 $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 = 6$
 - **voilà! 1010 represents negative six**

Two's complement is an approach for representing negative integers

- Define negative by addition: $-x$ is value added to x to get 0
- Process:
 1. Write out the number in binary
 2. Invert the bits
 3. Add 1
- From and To two's complement use an identical process
- How does this work? Overflow...

Another Example

What value is this 8-bit signed integer?

sign bit

	1	1	0	0	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
	0	0	1	1	0	0	1	1
	Flip each bit							
+	0	0	0	0	0	0	0	1
	Add one							
	0	0	1	1	0	1	0	0

two's complement

$$2^5 \quad 2^4 \quad \quad \quad 2^2$$
$$32 + 16 + \quad \quad \quad 4 = 52$$

So 11001100 represents -52

so we can “decode” binary
signed integers, now for

encoding signed integers

Signed Integers: encoding negative values

Example: How do you store -52 in 8 bits?

Start by encoding +52:

One way to do it: by repeated integer division

$$52 // 2 = 26 \text{ r } 0$$

$$26 // 2 = 13 \text{ r } 0$$

$$13 // 2 = 6 \text{ r } 1$$

$$6 // 2 = 3 \text{ r } 0$$

$$3 // 2 = 1 \text{ r } 1$$

$$1 // 2 = 0 \text{ r } 1$$

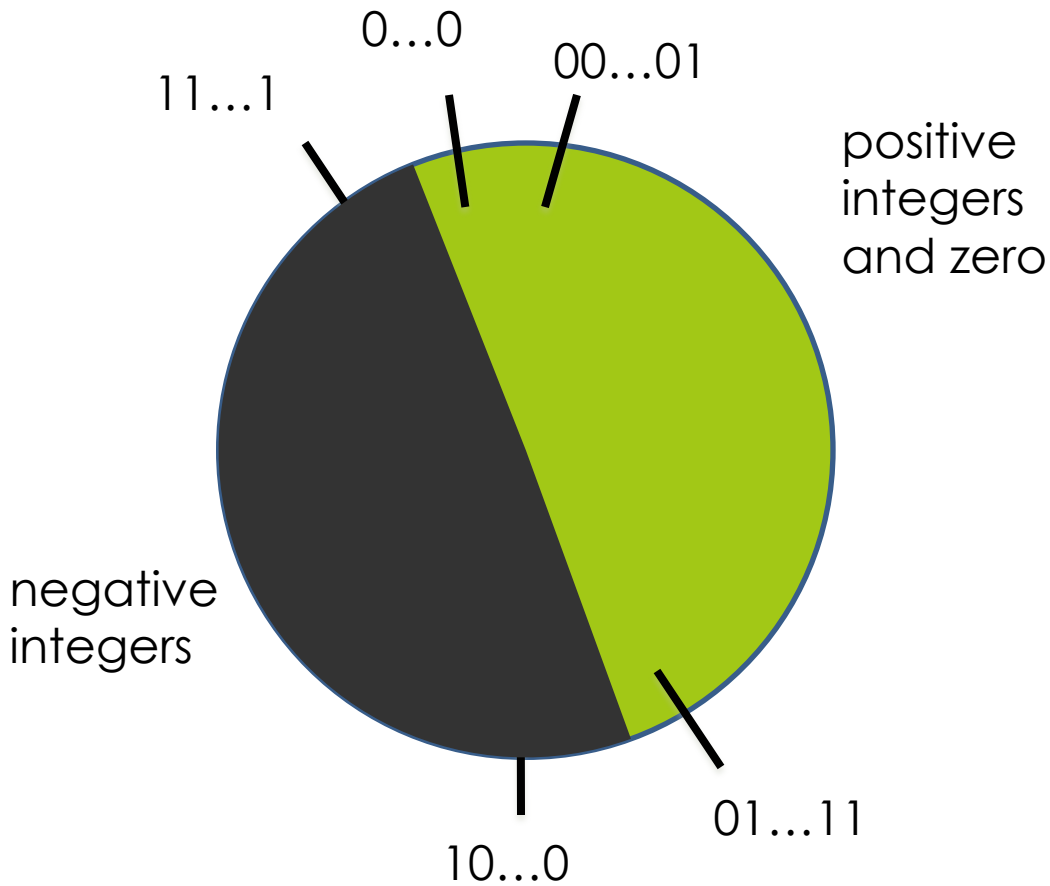
00110100

Another way: find the powers of two that add up to 52:

52 =

			32	+	16	+		4		
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
	0	0	1	1	0	1	0	0		

Range of Two's Complement Representations (for k bits)



Bit pattern	Decimal value
00...00	0
00...01	+1
...	
01...11	$+2^{k-1}-1$
10...00	-2^{k-1}
...	
11...11	-1

Range Examples

<u>bits</u>	<u>minimum value</u>	<u>maximum value</u>
8	$-2^7 = -128$ 10000000	$2^7 - 1 = +127$ 01111111
16	$-2^{15} = -32,768$	$2^{15} - 1 = +32,767$
32	-2^{31} $= -2,147,483,648$	$2^{31} - 1$ $= +2,147,483,647$
64	-2^{63} $= -9,223,372,036,854,775,808$	$2^{63} - 1$ $= +9,223,372,036,854,775,807$

From whole numbers to rational numbers

Real Numbers in the Machine?

- Real numbers measure **continuous** quantities; can we represent them exactly in the machine?
- Not possible with a fixed number of bits
- Can only approximate by rational numbers using **floating point representations**
- e.g. $\pi \approx 3.14159$

Binary and fractions

- Decimal 5.75 can be represented in binary as follows, because $.75 = \frac{1}{2} + \frac{1}{4} = 2^{-1} + 2^{-2}$

$$5.75 = 5 + 0.75$$

$$= 101 + 0.11 \text{ (i.e. } 2^{-1} + 2^{-2}\text{)}$$

$$= 101.11 = 1.0111 \times 10^{10}$$

decimal

binary

In binary floating point the mantissa is a binary fraction, exponent is a binary integer, and the base of the exponent is always 2

101.11 has *mantissa* 1.0111 and *exponent* 10

Some Floating Point Anomalies

- Rounding error
 - remember, floating point with a fixed number of digits is an *approximation, no matter what base is used!*
 - in addition, there is no finite base two representation for $1/10$

- Resolution

- Accumulation of errors: repeated operations may get further and further from the “true” value

Rounding in any base

- Floating point works with a finite fixed number of digits
- No matter what the base, some numbers can only be approximated
 - π , e , other irrationals
 - but also rationals needing more digits than we have in a machine word

Rounding in binary

```
>>> x = 1/10
```

```
>>> x
```

```
0.1
```

```
>>> y = 2/10
```

```
>>> y
```

```
0.2
```

```
>>> x + y
```

```
0.30000000000000004
```

```
>>> from decimal import Decimal
```

```
>>> Decimal(x)
```

```
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

```
>>> Decimal(y)
```

```
Decimal('0.2000000000000000011102230246251565404236316680908203125')
```

```
>>> Decimal(x+y)
```

```
Decimal('0.30000000000000000444089209850062616169452667236328125')
```

```
>>>
```

python prints a rounded value

Ack!
Whyyyy?

the actual value looks like
this (in decimal)!

Why is $1/10$ not exactly $.1$?

Let's compute $1/10$ using binary long division:

$$\begin{array}{r}
 .000110011\dots \\
 1010 \overline{) 1.00000000\dots} \\
 \underline{1010} \\
 1100 \\
 \underline{1010} \\
 10000 \\
 \underline{1010} \\
 1100 \\
 \underline{1010} \\
 10\dots
 \end{array}$$

we get a repeating series of digits 11001100...

same

Resolution

- Tiny example: *suppose we use a binary floating point notation like this (4 bits):*

$d_1.d_2d_3 \times 2^e$, where $-1 \leq e \leq 2$ and $d_1 = 1$ unless $e=0$



- **Representable values get sparser as we go to bigger and bigger numbers!**

Image source: “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, by David Goldberg. *Computing Surveys*, 1991

Floating point: the bottom line

For serious work like simulating the weather or the economy,
hire an expert! (or be an expert)

You should be able to

- Count in unsigned binary
0, 1, 10, 11, 100, ...
- Add in binary and know what overflow is
- Determine the sign and magnitude of an integer represented in two's complement binary
- Determine the two's complement binary representation of a positive or negative integer

Some Helpful Python functions

```
>>> bin(10)
```

```
'0b1010'
```

```
>>> hex(10)
```

```
'0xa'
```

```
>>> from decimal import Decimal
```

```
>>> Decimal(.2)
```

```
Decimal('0.2000000000000000000111022302462515654042363  
16680908203125')
```