Extended Introduction to Computer Science
CS1001.py

Chapter B Integer Representation
Lecture 6a (in binary and other bases)

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

עדכונים קצרים

- כפי שהודענו במייל, ניתן להגיש קבצי py במודל
- תרגיל בית 2 פורסם, ל- 3 שבועות
- שבוע הבא: שבוע השלמת פערים. לא לומדים חומר חדש.
  - אני אפרסם בקרוב תקציר של החומר שנלמד עד כה עם דגשים ללמידה – מה קריטי להמשך ומה אפשר לדחות קצת. יעזור למי שזקוקים להכוונה ומיקוד להשלמת פערים
  - יום ראשון: אגיע פיזית לכיתה + זום (כרגיל) לשעת קבלה. הפעם יהיה אפשר להשתתף בזום.
  - אם זקוקים לשיחה פרטית להכוונה נוספת – לא להסס לפנות אליי

Plan for This Lecture

"God created the natural numbers;
all the rest is the work of man"

(Leopold Kronecker, 1823 –1891)

1. From hardware to bits

2. From bits to the Naturals
   - Naturals in binary (base 2)
   - Large `int`s in Python
   - Binary arithmetic ($+$, $*$)

3. Naturals in other bases

4. Negative integers

Deep Inside the Computer Circuitry

- Most of the logic and memory hardware inside the computer are electronic devices containing a huge number of transistors
  - The transistor was invented in 1947, Bell Labs, NJ, and won the inventors a Nobel Prize in 1956
  - 3-10 nm (1 nanometer = 10^{-9} meter)

- At any given point in time, the voltage in each of these tiny devices can be in two distinct states, for example either +5v or 0v. So transistors can operate as binary switches, and are combined to form highly complex and functional circuitry.
- This means that data in the computer are represented in binary.
- An extremely useful abstraction is to ignore the underlying electric details, and view the voltage levels as bits (binary digits):
  - 0v is viewed as 0, +5v is viewed as 1

Bits, Bytes and Beyond

1 bit = 0/1

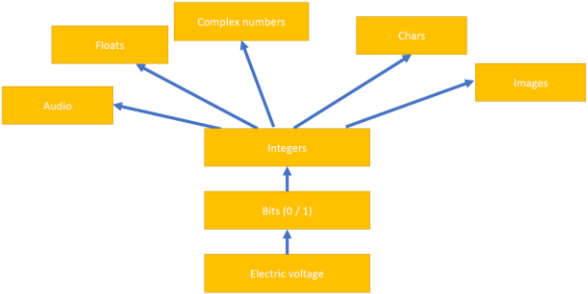
1 byte = 8 bit

1 KB (Kilobyte) = 2^{10} bytes = 1024 bytes
1 MB (Megabyte) = 2^{20} bytes = 1024 KB
1 GB (Gigabyte) = 2^{30} bytes = 1024 MB
1 TB (Terabyte) = 2^{40} bytes = 1024 GB

…

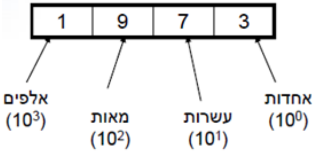
From Bits to Numbers and Beyond - Hierarchy of Abstraction Levels

- The next conceptual step is arranging these bits so they can represent natural numbers.
- Then, we will strive to arrange natural numbers so they can represent other types of numbers - negative integers, real numbers, complex numbers, and furthermore characters, text, pictures, audio, video, etc.

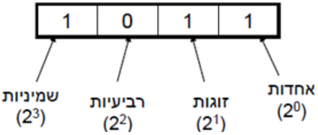


- We will begin at the beginning: From bits to integers.

Natural Numbers in Binary



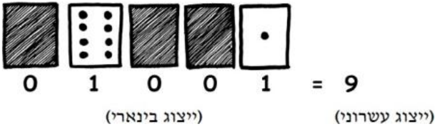
Decimal



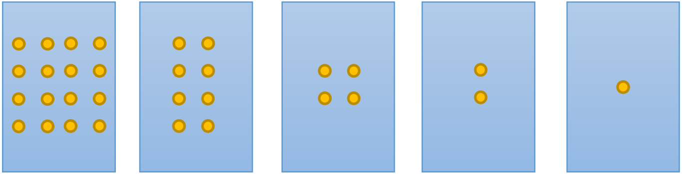
Binary

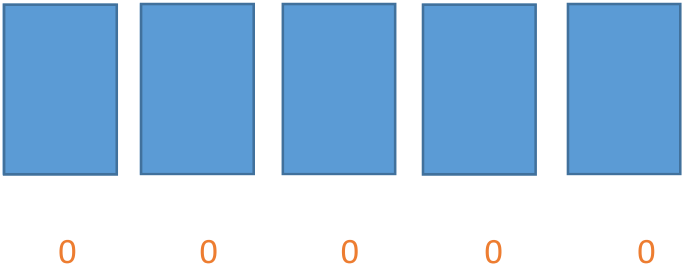
Natural Numbers in Binary

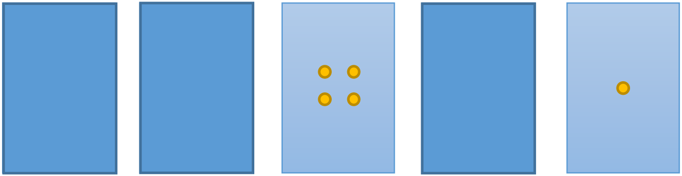
- Explanation using cards *



* Inspired by [Computer Science Unplugged](#) / [Hebrew version](#)

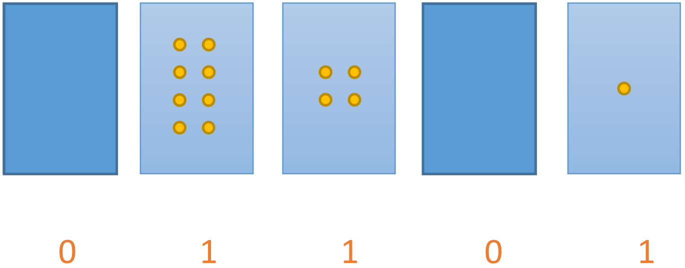



0
0
0
0
0

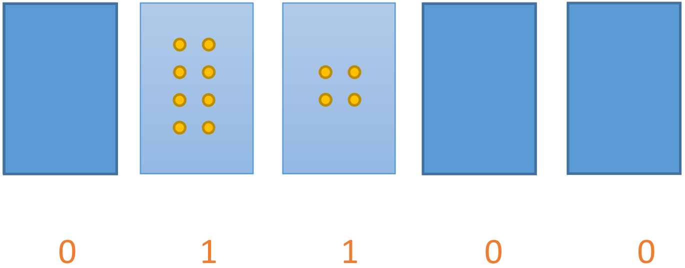


0 0 1 0 1

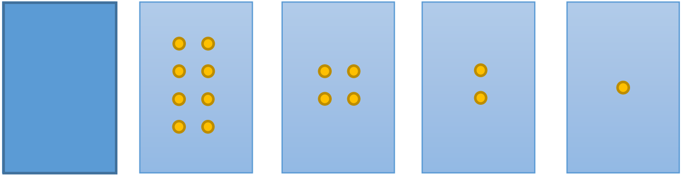
$$5_{(10)} = 101_{(2)}$$



$$13_{(10)} = 1101_{(2)}$$

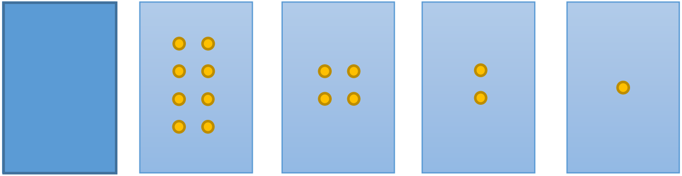


$$12_{(10)} = 1100_{(2)}$$



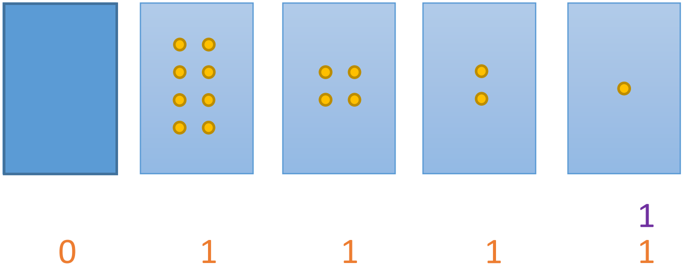
0 1 1 1 1

$$15_{(10)} = 1111_{(2)}$$

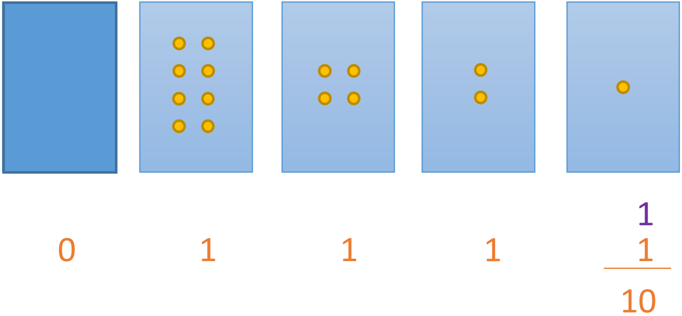


0 1 1 1 1

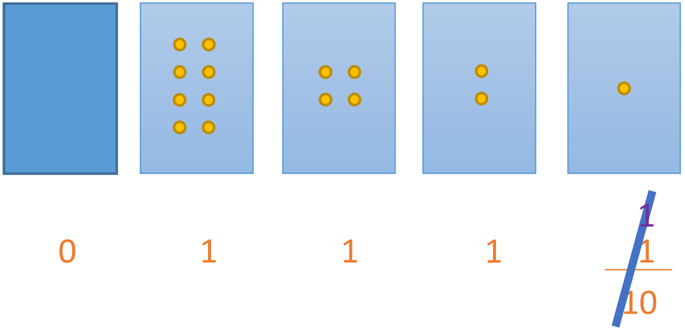
Adding 1 (to 15)



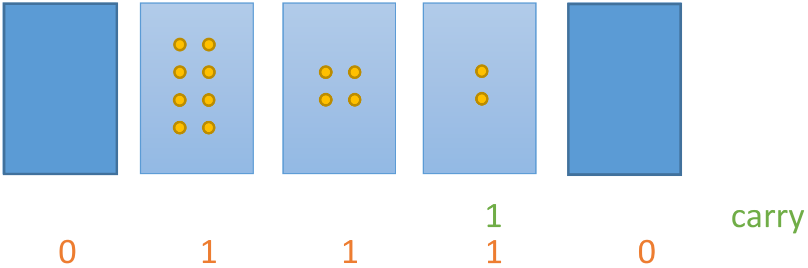
Adding 1 (to 15)



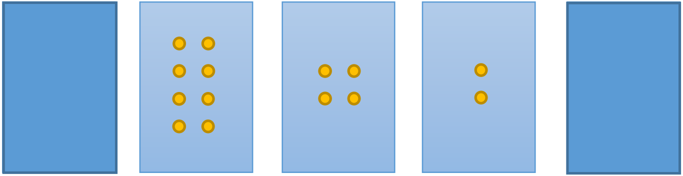
Adding 1 (to 15)



Adding 1 (to 15)



Adding 1 (to 15)

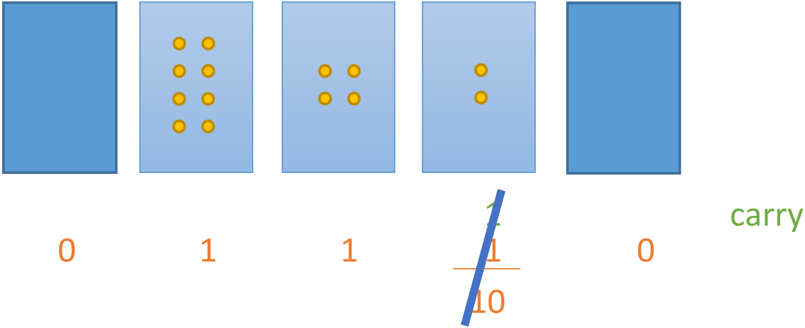


1 carry

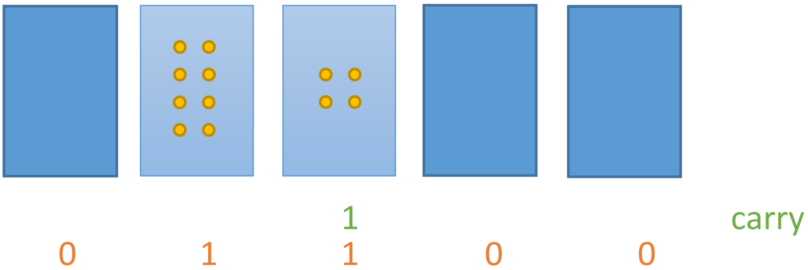
0 1 1 1 0

10

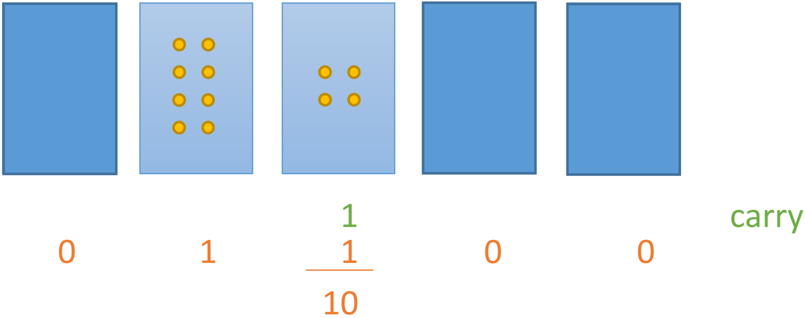
Adding 1 (to 15)



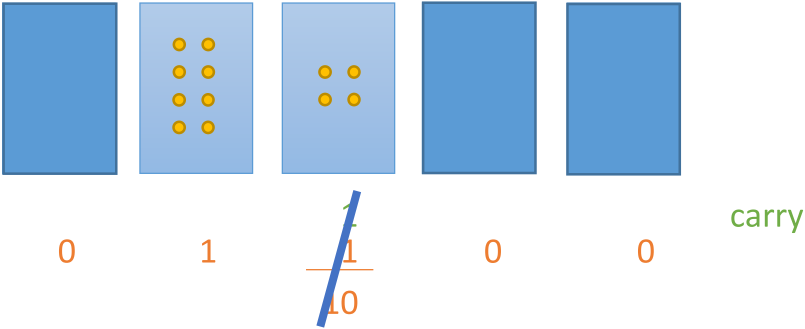
Adding 1 (to 15)



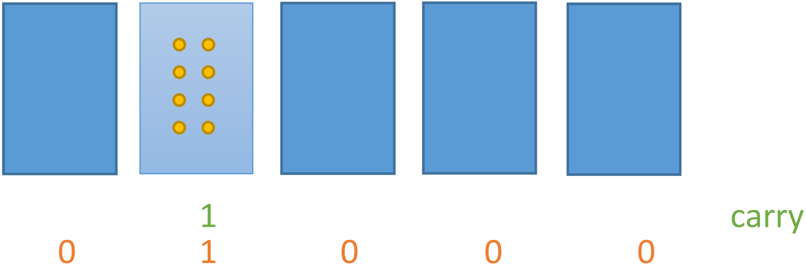
Adding 1 (to 15)



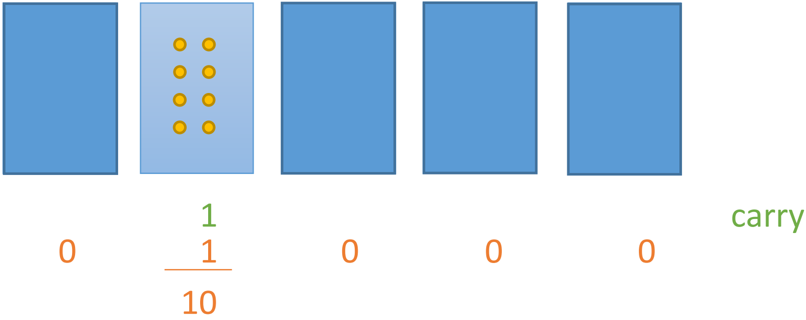
Adding 1 (to 15)



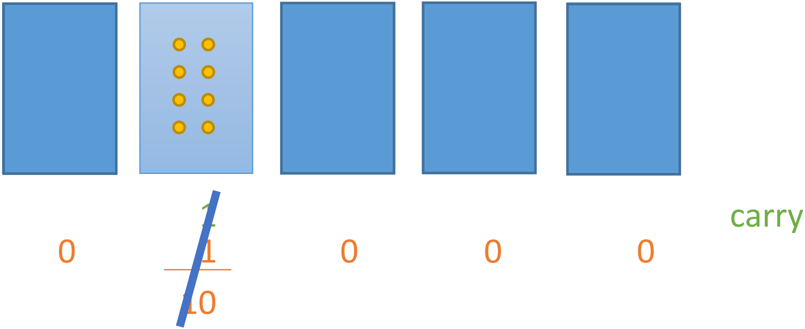
Adding 1 (to 15)



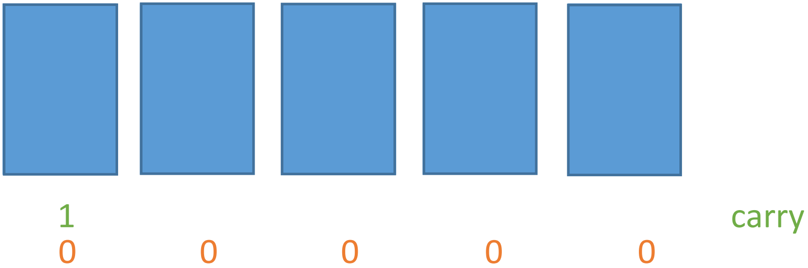
Adding 1 (to 15)



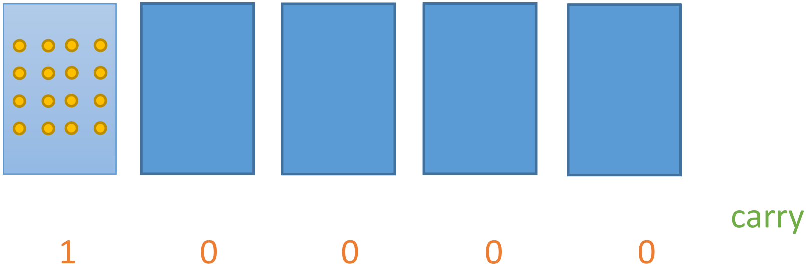
Adding 1 (to 15)



Adding 1 (to 15)



Adding 1 (to 15)



Adding 1 (to 15)

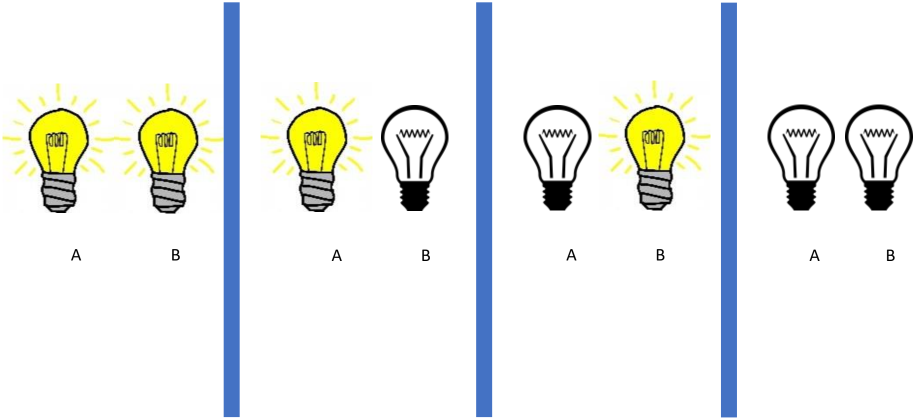
$$16_{(10)} = 10000_{(2)}$$

Naturals in Binary – an Important Property

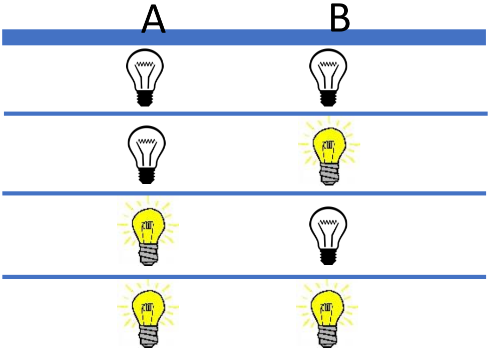
- Q: How many distinct values can be represented using n bits?
- A: 2^n

מתג אחד – ניתן להבחין בין שני מצבים

שני מתגים – ניתן להבחין בין ארבעה מצבים

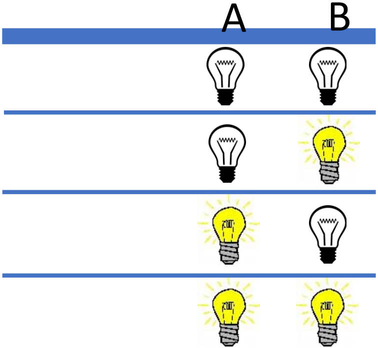


שני מתגים



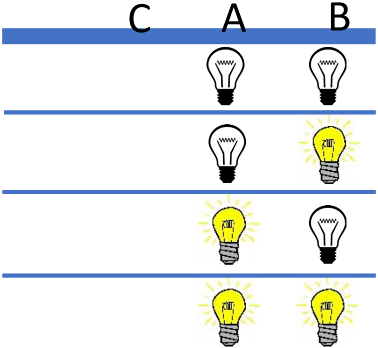
כמה מצבים שונים יוצרים שלושה מתגים?

כמה מצבים שונים יוצרים שלושה מתגים?



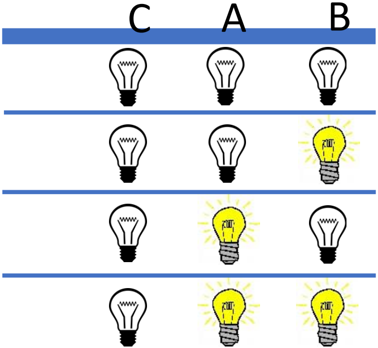
אלו המצבים שיוצרים שני מתגים

כמה מצבים שונים יוצרים שלושה מתגים?

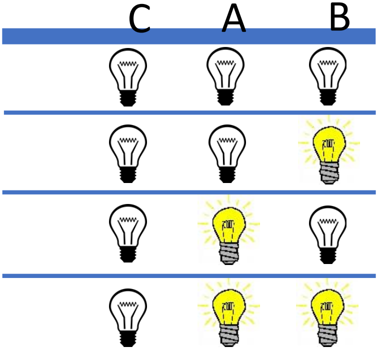


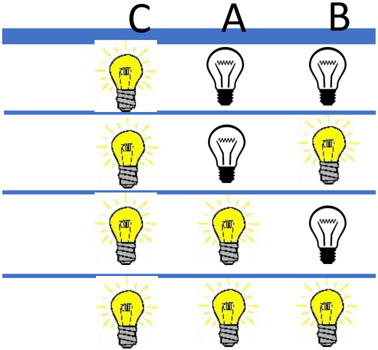
נוסיף מתג שלישי

כמה מצבים שונים יוצרים שלושה מתגים?



כמה מצבים שונים יוצרים שלושה מתגים?





כמה מצבים שונים יצרו n מתגים?

- לכל מתג יש שתי אפשרויות (כבוי/דולק או 0 / 1)
- מספר האפשרויות הכולל הוא 2^n

Naturals in Binary – an Important Property

- Q: How many distinct values can be represented using n bits?
- A: 2^n

- For example, 64-bit sequence can represent 2^{64} different values.
- Used to represent the Naturals, the range is $0, 1, \dots, 2^{64} - 1.$
  - $\underbrace{000 \dots 0}_{64\ bits}$ represents 0

    $\dots$

  - $\underbrace{111 \dots 1}_{64\ bits}$ represents $2^0 + 2^1 + \cdots + 2^{63} = 2^{64} - 1$

Limits to Natural Number Representation

- Modern computers are arranged by words, groups of fixed number of bits that correspond to size of registers, units of memory, etc.
- Word size is uniform across a computer, and depends both on the hardware (processors, memory, etc.) and on the operating system.
- Typical word sizes are 8, 16 (Intel original 8086), 32, or 64 bits (most probably used by your PC or iMAC).
- In many programming languages, integers are represented by either a single computer word, or by two computer words (e.g. types `int` and `long` in Java).
- This means that the range of representable integers is limited.
- Things are quite different in Python, as we shall now see.

Handling Large Integers in Python

- To "bypass" the word-size limit, several words should be manipulated together correspondingly (to represent higher powers of 2).
- This is either done explicitly by the user/programmer, or provided directly by the programming language.
- Python takes care of large integers internally, even if they are way over the word size.

```
>>> 2**199 #200 bits
803469022129495137770981046170581301261101496891396417650688

>>> 2**299 #300 bits
1018517988167243043134222844204689080525734196832968125318070224677190649881668353091698688

>>> 3**97 - 2**121 + 17
19088056320749371083854654541807988572109959828
```

Complexity Issues

- Still, when manipulating large integers, one should think of the computational resources involved:
  - Time: How many basic operations (or, clock cycles) are needed for the various arithmetic operations? Time will grow as n, the number of bits in the numbers we operate on, grows. How time grows as a function of n is important since this will make the difference between fast and slow tasks, and even between feasible and infeasible tasks.
  - Space: There is no difficulty to represent $2^{2^{10}}$, or, in Python `2**(2**10)` (as the result takes four lines on my Python Shell window, I skip it).
    - But don't try $2^{2^{100}}$ (why?)!
- As already mentioned, We will define these notions precisely soon, but you should start paying attention to such considerations.

Bit Addition and Multiplication

- Addition and multiplication of single bits are not different from the decimal operations we are used to:

b1	b2	b1+b1	b1*b2
1	1	10	1
1	0	1	0
0	1	1	0
0	0	0	0

Binary Addition

- Suppose we have two n-bit natural numbers in binary, A and B.
- Computing $A + B$ is done "bitwise", with possibly a carry bit in each position.
- This logic can be implemented by hardware circuitry

```
  (1)(1)
      1 1 0 0    (A = 12₁₀)
+     1 1 0 1    (B = 13₁₀)
    ---------
  = 1 1 0 0 1    (A+B = 25₁₀)
```

- Property 1: The maximal length of the output is $n + 1$ bits
- Property 2: At most $2n - 1$ bit-additions needed

Binary Multiplication

- Suppose we have two n-bit natural numbers in binary, A and B.
- Computing $A * B$ can also be done "bitwise", just like decimal multiplication from elementary high-school
- This logic can be implemented by hardware circuitry

```
          1 1 0 0    (A = 12₁₀)
        × 1 1 0 1    (B = 13₁₀)
        ---------
          1 1 0 0     ← Corresponds to the rightmost 'one' in B
  +     0 0 0 0       ← Corresponds to the next 'zero' in B
  +(1) 1 1 0 0
  + 1 1 0 0
  ---------------
= 1 0 0 1 1 1 0 0    (A·B = 156₁₀)
```

- Property 1: The maximal length of the output is $2n$ bits
- Property 2: exactly n^2 bit-multiplications, $< 4n^2$ bit-additions

Try proving both at home!

Binary Subtraction and Division

- Subtraction and division are also performed in a similar way to decimal, but we will not show it.
- You may be asked about this in your HW

Unary Representation of Naturals

- Consider the natural number nineteen (19 in decimal)
  - In binary, it is represented as 10011
  - In unary (base 1) it is represented as 0000000000000000000

- The two representations refer to the same entity, nineteen. However, the lengths of the representations are substantially different:

  The unary representation is exponentially longer than the binary.

- To see this, consider the natural number 2^n.
  - In unary it is represented by 2^n digits
  - In binary it is represented by a single '1', followed by n '0's.

Representation of Naturals in Base 10

- A natural number N can be represented in base 10, as a polynomial, whose coefficients are natural numbers smaller than 10.

$$N = a_k \cdot 10^k + a_{k-1} \cdot 10^{k-1} + \ldots + a_1 \cdot 10^1 + a_0$$

(for each i, $0 \leq a_i < 10$)

- The coefficients of the polynomial are the digits of N in its base 10 representation:

$$\boxed{N_{(10)} = a_k a_{k-1} \ldots a_1 a_0}$$

- **Claim**: The natural number N represented as a polynomial of degree k (has $k+1$ digits, $a_k \neq 0$) in base 10 satisfies $10^k \leq N < 10^{k+1}$.
- Do you see why?

Representation of Naturals in Base 2

- A natural number N can be represented in base 2, as a polynomial, whose coefficients are natural numbers smaller than 2.

$$N = a_k \cdot 2^k + a_{k-1} \cdot 2^{k-1} + \ldots + a_1 \cdot 2^1 + a_0$$

(for each i, $0 \leq a_i < 2$)

- The coefficients of the polynomial are the digits of N in its base 2 representation:

$$\boxed{N_{(2)} = a_k a_{k-1} \ldots a_1 a_0}$$

- **Claim**: The natural number N represented as a polynomial of degree k (has $k+1$ digits, $a_k \neq 0$) in base 2 satisfies $2^k \leq N < 2^{k+1}$.
- Do you see why?

Representation of Naturals in Base $b > 1$

- A natural number N can be represented in base b ($b > 1$, an integer), as a polynomial, whose coefficients are natural numbers smaller than b.

$$N = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \ldots + a_1 \cdot b^1 + a_0$$

(for each i, $0 \leq a_i < b$)

- The coefficients of the polynomial are the digits of N in its base b representation:

$$\boxed{N_{(b)} = a_k a_{k-1} \ldots a_1 a_0}$$

- **Claim**: The natural number N represented as a polynomial of degree k (has $k+1$ digits, $a_k \neq 0$) in base b satisfies $b^k \leq N < b^{k+1}$.
- Do you see why?

Representation of Naturals in Base > 1

Concllusion (proof in the recitations): The number of digits, d, required for representing the natural number N in base b is

$$d = \lfloor log_b N \rfloor + 1$$

- For example, $1024 = 2^{10}$ requires 11 bits (10000000000)

  $2^{2^{10}}$ requires $2^{10}+1$ = 1025 bits.

Other Bases > 1

- Beside the commonly used decimal (base 10) and binary (base 2) representations, other representations are also in use. In particular the ternary (base 3), octal (base 8) and hexadecimal (hex, 0x, base 16) are well known.

- The lengths of representations in different bases differ. However, the lengths in any two bases $b > 1$ and $c > 1$ are related linearly.
- A number represented with d digits in base $b > 1$ will take at most $\lceil d \cdot \log_c b \rceil$ digits in base $c > 1$.
  - For example, a number represented with d digits in base 10 will take at most $\lceil d \cdot \log_2 10 \rceil$ digits in base 2 (=bits).
    - 9 → 1001
    - 99 → 1100011

- You may prove this in the Tirgul/HW

Different Base Representations in Python

- Python has built in functions for converting a number from decimal (base 10) to binary, octal (base 8), and hexadecimal (base 16).

```
>>> bin(1000)
'0b1111101000'

>>> oct(1000)
'0o1750'

>>> hex(1000)
'0x3e8' #hexadedimal digits: 0,1,2,...,9,a,b,c,d,e,f

>>> type(bin(1000))
<class 'str'>
```

- The returned values are strings, whose prefixes `0b,0o,0x` indicate the bases 2, 8, 16, respectively.

Hexadecimal Representations in Python

- In hex, the letters a , b , . . . , f indicate the “digits” 10, 11, . . . , 15, respectively.

```
>>> hex(10)
'0xa'
>>> hex(15)
'0xf'
>>> hex(62)
'0x3e'          🡐 62 = 3*16 + 14
```

- Recitation (”tirgul”): Conversion to “target bases” $\neq$ 2, 8, 16.

Converting to Decimal in Python

- Python has a built-in function, int, for converting a number from base b representation to decimal representation. The input is a string, which is a representation of a number in base b (for the power of two bases, `0b`, `0o`, `0x` prefixes are optional), and the base, b, itself .

```
>>> int("0110",2)
6
>>> int("0b0110",2)
6
>>> int("f",16)
15
>>> int("fff",16)
4095
>>> int("fff",17)
4605
>>> int("ben",24)
6695
>>> int("ben",23)
Traceback (most recent call last):
File "<pyshell#16>", line 1, in
<module> int("ben",23)
ValueError: invalid literal for int() with base 23:'ben'
```

"a" for 10, "b" for 11, ... ,"m" for 22
No "n" in base 23

Comic Relief *

* אני מזמין אתכם לשלוח לי הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Negative Integers

- An interesting issue is how negative integers can be represented.
- Suppose we use n bits. We would like to divide the range of 2^n values represented by those bits such that about half would be used for positive and about half for negative integers.
- For example, 32 bits can be used to represent any integer, k, in the range $-2^{31} \leq k \leq 2^{31} - 1$.
- We would also like arithmetic operations to be efficient.

Negative Integers - ~~the Sign Bit Option~~

- We could simply assign one bit (say, the leftmost) to denote the sign: 0 for +, 1 for −
- Example with 8 bit numbers: +1=00000001, -1=10000001

- Disadvantages?

  - 0 has 2 representations (+0 and -0)
  - arithmetical operations (e.g. addition) involving negative numbers require slightly different algorithms (and computer circuitry). For example, suppose we deal with 8 bit numbers, and add +1 (00000001) and -1 (10000001) in this representation. Just adding the bits wll yield 10000010, which is -2...

- So the sign bit method is not in use.
- The method most often used is Two's Complement

Two's Complement (for reference only)

- For the sake of completeness, we'll briefly explain how negative integers are represented in the two's complement representation.

- Suppose we have a k bit, non negative integer, M .
- To represent $-M$, we compute $2^k - M$, and drop the leading (leftmost) bit.
- For the sake of simplicity, suppose we deal with an 8 bit number:

100000000	2^8	100000000	2^8
- 00000001	$M = 1$	- 00001110	$M = 14$
11111111	-1	11110010	-14

- It turns out that if non negative integers have 0 as their leading (leftmost) bit, then negative integers will have 1 as their leading (leftmost) bit.
- So the leading bit practically behaves as a sign bit, with about half the numbers positive and half negative.
- Main advantage of 2's Complement over sign bit method: operations require no distinction between positive and negative numbers.

Highly Recommended Sources

- ”Computer Science Field Guide” chapter at
  https://www.csfieldguide.org.nz/en/chapters/data-representation/numbers/

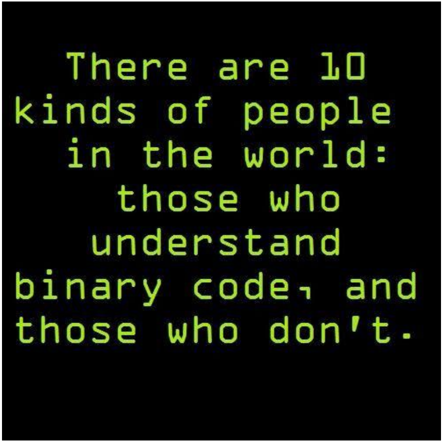
- A nice explanation on counting bases from a wonderful blog by Dr. Gadi Aleksandrowicz called “לא מדוייק“:
  https://gadial.net/2017/06/11/number_bases/

- A very thorough explanation on binary numbers (with some nice demos):
  http://courses.cs.vt.edu/csonline/NumberSystems/Lessons/index.html

- Two's complement method for representing negative integers, for those interested: https://youtu.be/4qH4unVtJkE

- Computer Science Unplugged / Hebrew version

Comic Relief *



* אני מזמין אתכם לשלוח לי הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר