Middlebury

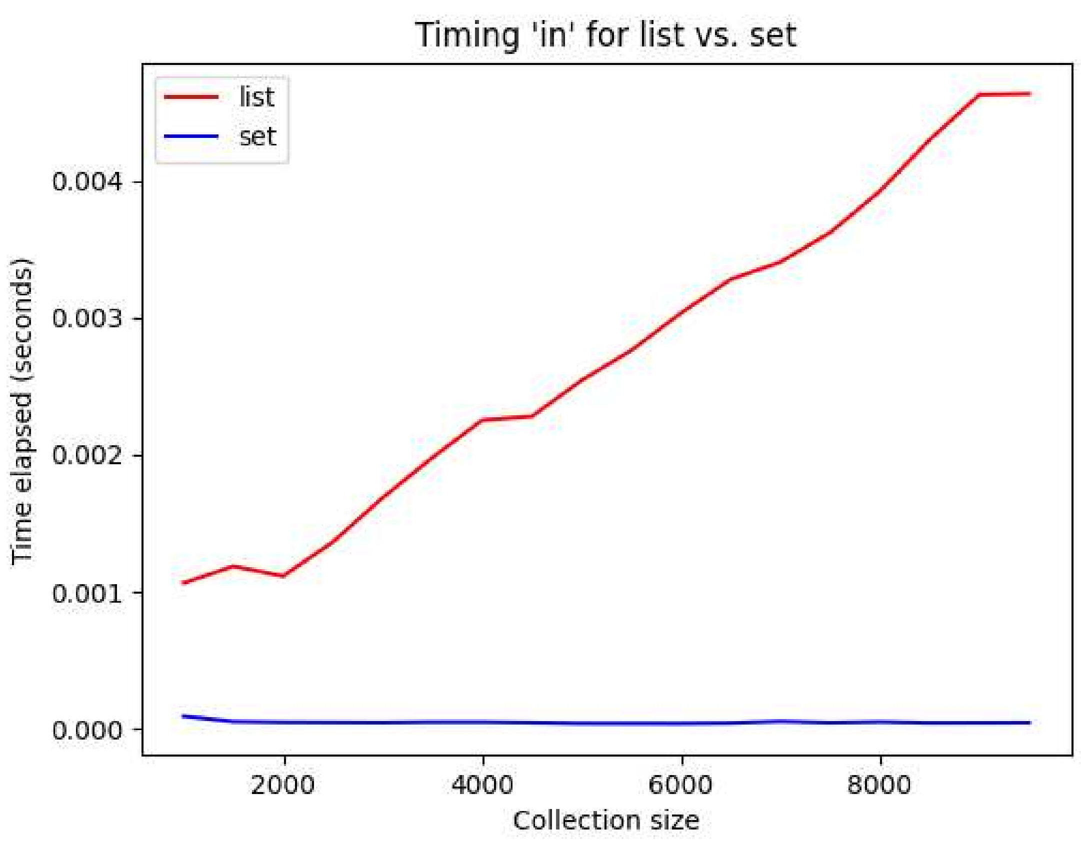
CSCI 146: Intensive Introduction to Computing

Fall 2025

Lecture 16: Complexity analysis and numerical representation

Goals for today

- Informally define **asymptotic complexity** with a focus on **runtime**.
- Describe the purpose and calculation of **Big-O notation**.
- Compare (e.g. rank) constant, logarithmic, linear, quadratic, and cubic complexities.
- Predict the runtime of an algorithm based on its Big-O complexity.
- Explain how integers and other types are represented in the computer.
- Perform binary addition of unsigned integers.
- Be aware of twos-complement representation.
- Be aware of the floating point representation and some of its limitations.



Analyzing the `in` operator.

Possible implementation:

"contains"

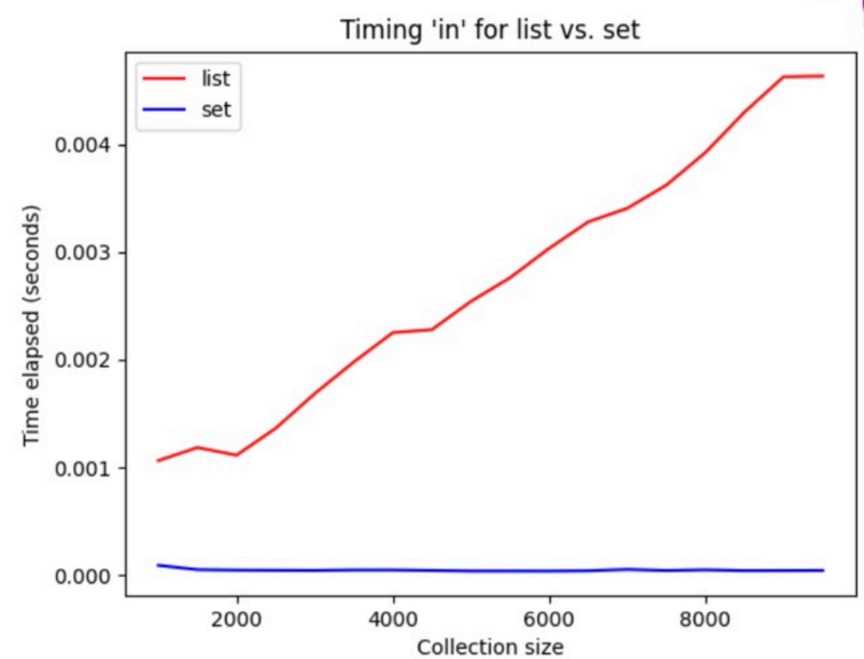
```python
def list_in(lst, x):
  """
  Should be the same as:
    >>> x in list
  """
  for value in lst:
    if value == x:
      return True
  return False
```

how many == performed?

$n = \text{len}(lst)$ $\approx n$ ==

$\# \text{ops} = an + b$

constants



Big-O notation allows us to describe the *asymptotic complexity* (runtime, memory) of an algorithm (not a specific implementation).

For the `in` operator on `list`s: (assume a `list` of `len` n)

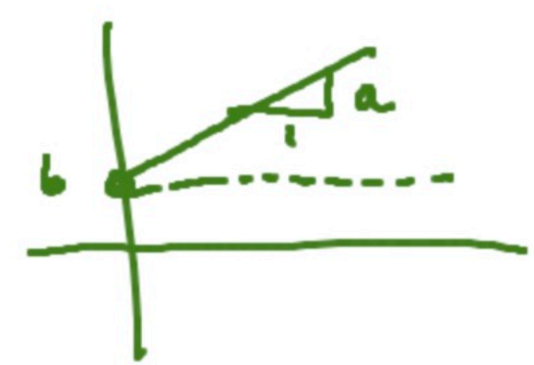
$$\rightarrow \qquad \texttt{runtime}(n) = an + b, \quad \text{where} \quad a, b \quad \text{are constants.}$$

Main idea: how does the runtime (or memory) grow as the size of the inputs grows? $n \to \infty$

- Big-O places an *upper-bound* on the growth of the function.
- If the function is a sum of several terms, only the fastest growing term is kept.
- Constant terms (i.e. those independent of the size of the input n) are omitted.

runtime

$$f(n) = \underbrace{an}_{} + b$$



$O(n)$

big-O

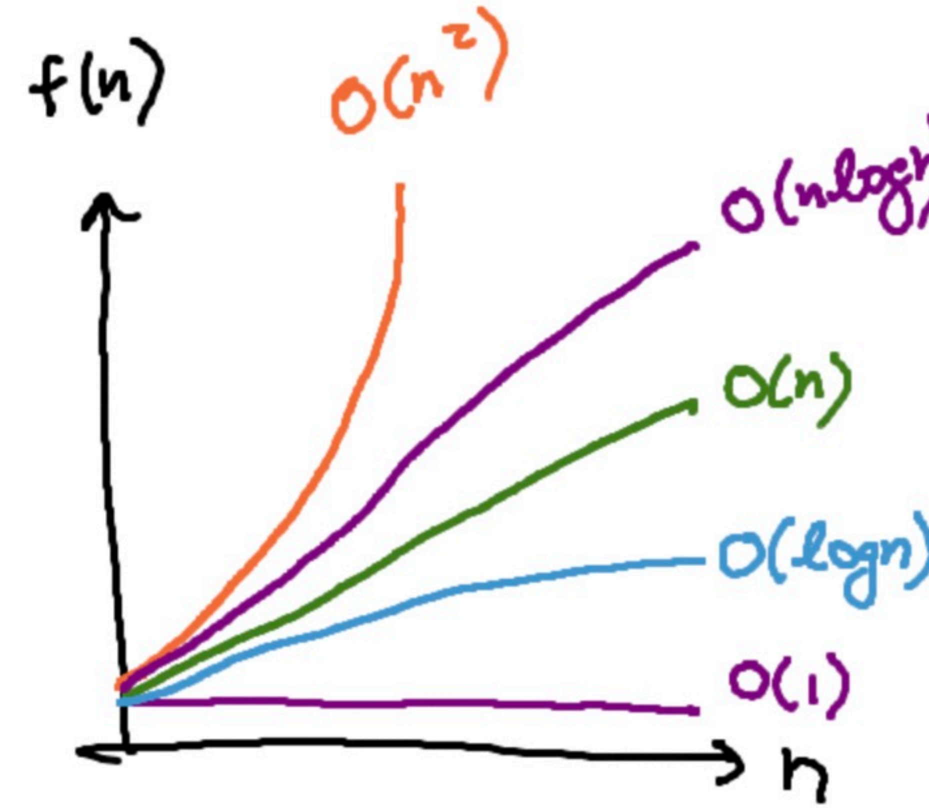
Example: determine a Big-O bound on the following function:

$$f(n) = \underbrace{\cancel{3}n^3} + \cancel{6n^2} + \cancel{n} + \cancel{5}.$$

$f(n)$ is $O(n^3)$

Common Big-O functions

Complexity	Description
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Linearithmic
$O(n^2)$	Quadratic



Estimating runtime using Big-O

Suppose that we had an algorithm with $O(n^2)$ complexity that takes 5 seconds to run on our computer when *n* is 1000. If the input increased to *n* of 3000, about how long would it take to run?

$$t = an^2$$

$$t_0 = 5 = a(1000)^2 \quad (1)$$

$$t_1 = a(3000)^2 \quad (2)$$

$$\frac{(2)}{(1)} = \frac{\cancel{a}(3000)^2}{\cancel{a}(1000)^2} = \frac{t_1}{t_0} = 9$$

$$t_1 = 9 * 5 = \boxed{45 \text{ sec}}$$

What is the complexity of these two implementations of a standard deviation function?

n-1 additions for sum → O(n)

```python
def mean(data):
    """
    Return mean of iterable data
    """
    return sum(data) / len(data)

def stddev(data):
    """
    Return standard deviation for iterable data
    """
    result = 0.0
    average = mean(data)
    for elem in data:
        result += (elem - average) ** 2
    return math.sqrt(result / (len(data) - 1))

def stddev2(data):
    """
    Return standard deviation for iterable data
    """
    result = 0.0
    for elem in data:
        result += (elem - mean(data)) ** 2
    return math.sqrt(result / (len(data) - 1))
```

O(n) $a_1 n + b_1$

$a_2 n + b_2$

$t = a_1 n + b_1 + a_2 n + b_2$

$= (a_1 + a_2) n + (b_1 + b_2)$

O(n)

$a_2 (a_1 n + b_1) n + b_2$

O(n^2)

Question 1: What is the big-O asymptotic bound of the following function?

$$f(n) = \cancel{3}\underbrace{n^2} + \cancel{4n} + \cancel{2}$$

$O(n^2)$

A. $O(n)$

B. $\boxed{O(n^2)}$

C. $O(n^2 + n)$

D. $O(3n^2 + 4n + 2)$

A B C D E ✋ ❤️ 👍 👎 😳 🤔 😊 🐦 🐢 🐍

0 40 0 0 0 0 4 0 0 0 0 0 2 4 1

Question 2: Which of the following functions has the same big-O asymptotic bound as the following function

$$f(n) = 2n^3 + 5n + 2$$

A. $2n^2 + 5n + 2$

B. $\boxed{n^3 + 2n^2}$

C. $n(2n^3 + 5n + 2)$

D. 2

A	B	C	D	E	✋	❤️	👍	👎	😳	🤔	😊	🐦	🐢	🐍
0	43	0	0	0	1	0	11	3	0	29	1	55	34	66

Question 3: What is the time complexity of the Python max function (example below):

[0, 1, 2, ..., 99]

```
>>> numbers = list(range(100))
>>> the_max = max(numbers)
```

A. $O(1)$

B. $O(100)$

C. $\boxed{O(n)}$

D. $O(n \log n)$

E. $O(n^2)$

```
m = numbers[0]
for n in numbers:
    if n > m:
        m = n
return m
```

Question 4: The code below implements matrix multiplication between two $n \times n$ matrices (stored as `list`s of `list`s). What is the time complexity of this algorithm?

n×n×n

n×n

n

```
for i in range(n):
    for j in range(n):
        val = 0.0
        for k in range(n):
            val += x[i][k] * y[k][j]
        res[i][j] = val
```

how many times does this line execute ? ❤️

A. $O(1)$

B. $O(n)$

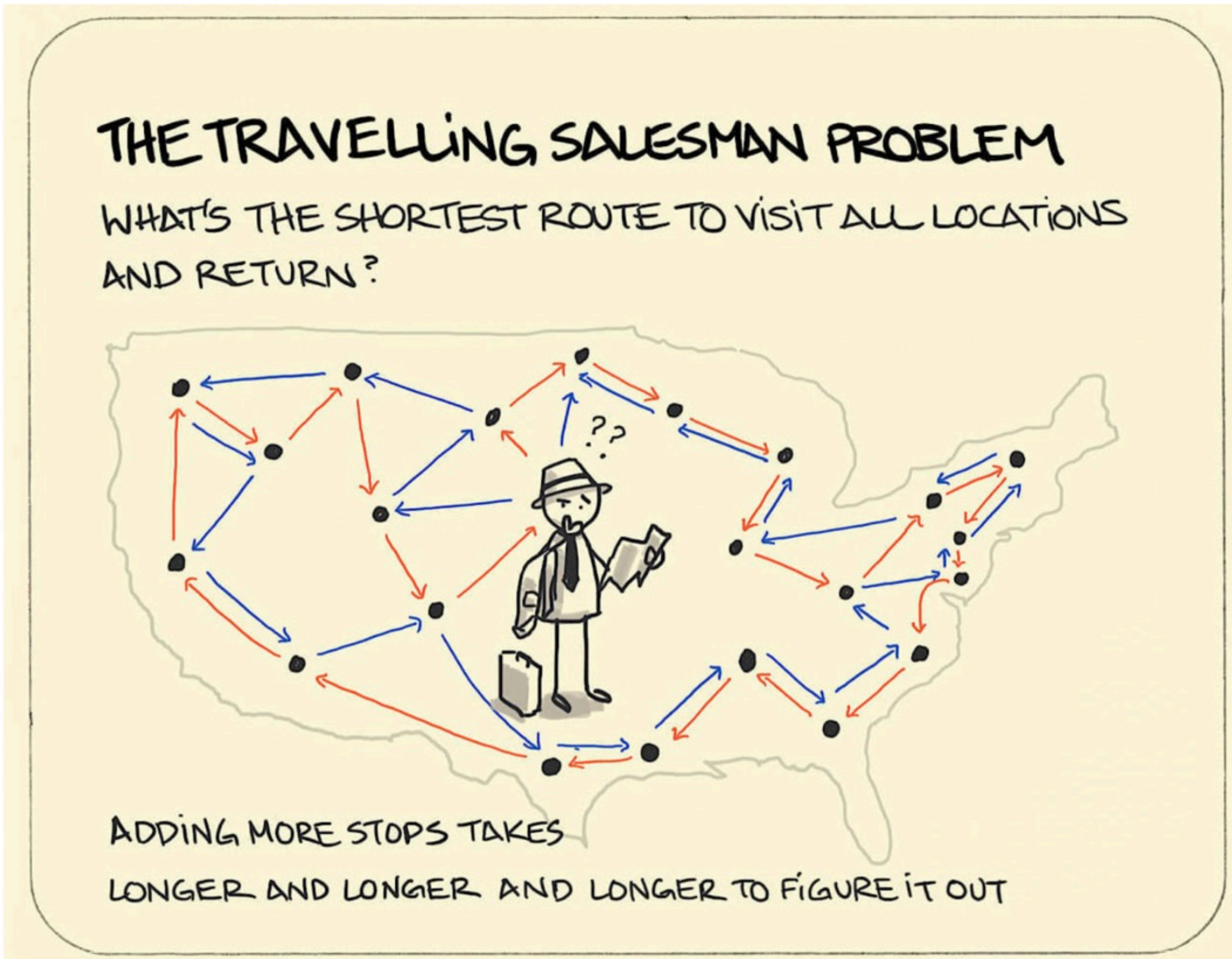
C. $O(n \log n)$

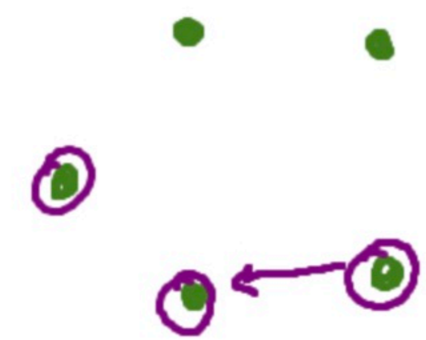
D. $O(n^2)$

E. $\boxed{O(n^3)}$

The traveling salesperson problem doesn't have a polynomial time algorithm.

Polynomial time problems: we can find a solution that runs in $O(n^k)$.





$n \times (n-1) \times (n-2) \ldots 3 \times 2 \times 1$

↳ all possible paths

$O(n!)$

source: https://github.com/ramoneas/travelling-salesman-problem-solver

The halting problem: can we write a program to determine whether an arbitrary program (with given inputs) *halts*?

Alan Turing proved that such a program does not exist.


```python
def will_this_halt():
  # suppose `halts` solves the halting problem for a given function object
  if halts(will_this_halt):
      while True:
          print("hello from an infinite loop")
```

When analyzing complexity, we can count mathematical, relational and logical operators. How does the computer perform these operations?

```
>>> x = 17
```

Computers use a binary (base-2) representation for numbers,
i.e. a sequence of *bits* (**bi**nary digi**ts**) which are either `0` or `1`.

decimal: $17 = 1 \times 10^1 + 7 \times 10^0$

binary: instead of powers of 10, use powers of 2

...	0	0	1	0	0	0	1
		2^5	2^4	2^3	2^2	2^1	2^0
		32	16	8	4	2	1

$\rightarrow 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 16 + 1 = 17$

Converting between binary and decimal.

Converting `1000011` to decimal:

$$\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

$$64+2+1 = \boxed{67}$$

Converting 437 to binary:

$$\begin{array}{ccccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \end{array}$$

$$\begin{array}{r} 437 \\ -\,256 \\ \hline 181 \end{array} \qquad \begin{array}{r} 181 \\ -\,128 \\ \hline 053 \end{array} \qquad \begin{array}{r} 53 \\ -\,32 \\ \hline 21 \end{array} \qquad \begin{array}{r} 21 \\ 16 \\ \hline 5 \end{array}$$

```
>>> bin(437)
```

Adding and subtracting with binary arithmetic.

Example: computing $\underbrace{23 + 5}_{28}$.

```
  1 1 1
 1 0 1 1 1
+0 0 1 0 1
----------
 1 1 1 0 0
16 8 4  → 28
```

Example: computing $17 - 10$.

```
 0 1 0 1 0 1 0 1
   1 0 0 0 1
-    1 0 1 0
------------
     0 1 1 1
```

10 − 01

```
  0  11 1
    x 2 5
      3 8
    -----
     1
      8 7
  +   3 8
    -----
    1 2 5
```

Question 5: What is the minimum number of bits (binary digits) to represent the decimal number 32?

A. 3
B. 4
C. 5
D. $\boxed{6}$
E. 7

Question 6: How many distinct numbers can be represented with 5 bits (binary digits) ?

A. 15

B. 16

C. 31

D. $\boxed{32}$

E. 63

0 or 1 0 or 1 0 or 1

$2 \times 2 \times 2 \ldots \quad 2^5$

Question 7: What is the result of adding 01101 and 00100?

A. 01001

B. 01101

C. 10000

D. $\boxed{10001}$

E. 10101

$$
\begin{array}{cccccc}
 & 1 & 1 & & & \\
 & 0 & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 1 & 0 & 0 \\
\hline
 & 1 & 0 & 0 & 0 & 1
\end{array}
$$

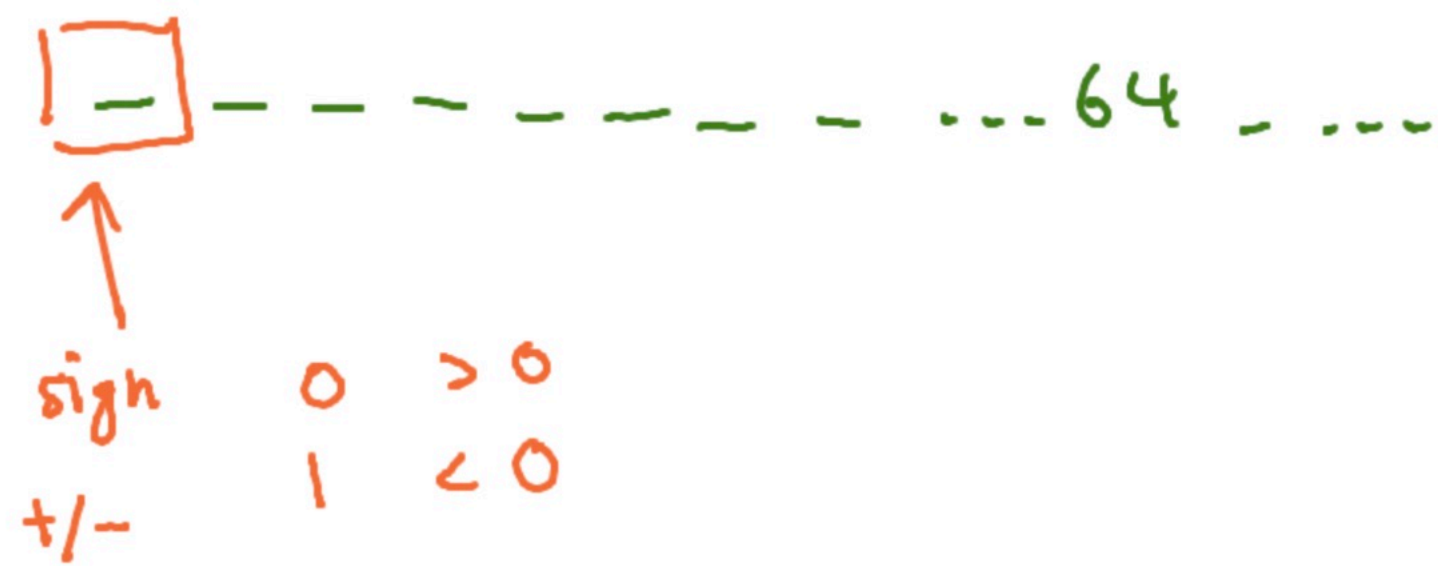
Python uses `sys.maxsize` as the maximum size of a container (and hence maximum index).

On most modern computers, this max size is represented by an integer with 64 bits. So what's `sys.maxsize`?

```
>>> import sys
>>> sys.maxsize
9223372036854775807
```

$2^{63} - 1$

negative numbers.



Summary and Reminders

- Big-O is concerned with finding an upper bound for our functions:
    1. Find fastest growing terms.
    2. Ignore constants.
- More Big-O analysis applied to searching and sorting algorithms on Wednesday.
- Use the built-in `bin` function to convert a decimal number to binary and check your work.
- Be careful with floating-point expressions: don't use equality checks, instead use tolerances.
- Programming Assignment 7 initial due date on Thursday.
- Quiz 8 this Friday includes retakes from Quizzes 4 - 7 + new Quiz 8 topics + Midterm 1 retakes of 2-3 questions (TBD).
- Use "Regrade Requests" form on the website. See Gradescope comments by clicking on **Code**.