Algorithms and Data Structures for Data Science

Arrays and Asymptotic Efficiency

CS 277
Brad Solomon

February 6, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Exam 1

Exams will be proctored by the CBTF: https://cbtf.engr.illinois.edu/

(That link will have a link to **Prairietest**, where you can sign up for exam 1)

Reservations open on February 2nd @ 9 AM

You must take the exam sometime between 2/14 and 2/16!

See website for expected content:

https://courses.grainger.illinois.edu/cs277/sp2023/exams/

Learning Objectives

Conceptualize the array list implementation

Introduce the concept of asymptotic efficiency

Compare list implementations using big O

List Implementations

1. Linked List

2. Array List

Creating an array list

```
myList = []
l = [0]*5
matrix = numpy.zeros([2, 3])
```

Multi-dimensional list memory

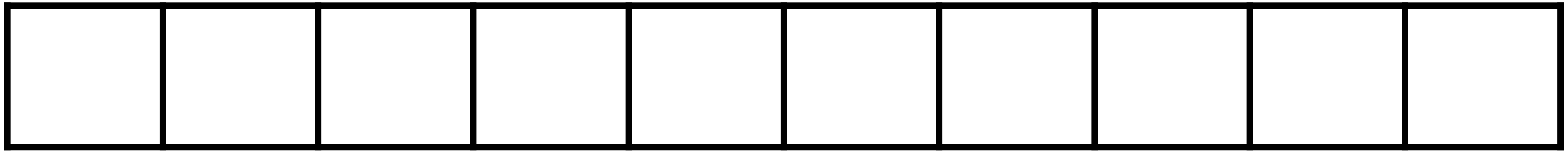
```
import numpy as np

r1 = [7, 8, 9]
r2 = [4, 5, 6]
r3 = [1, 2, 3]
plist = [r1, r2, r3]

myNP = np.array(plist)

print(myNP)

myNP = myNP.reshape(1, 9)

print(myNP)
```

Array __len__()

```
len(l)

numpy.shape(matrix)
```

Array __getitem__()

```
npArray[3]

npArray[9000]
```



Array __getitem__()

```
l[3]

l[9000]
```

Python List Size Implementation

```c
PyObject *
PyList_GetItem(PyObject *op, Py_ssize_t i)
{
    if (!PyList_Check(op)) {
        PyErr_BadInternalCall();
        return NULL;
    }
    if (i < 0 || i >= Py_SIZE(op)) {
        if (indexerr == NULL)
            indexerr = PyString_FromString(
                "list index out of range");
        PyErr_SetObject(PyExc_IndexError, indexerr);
        return NULL;
    }
    return ((PyListObject *)op) -> ob_item[i];
}
```

This is C code!

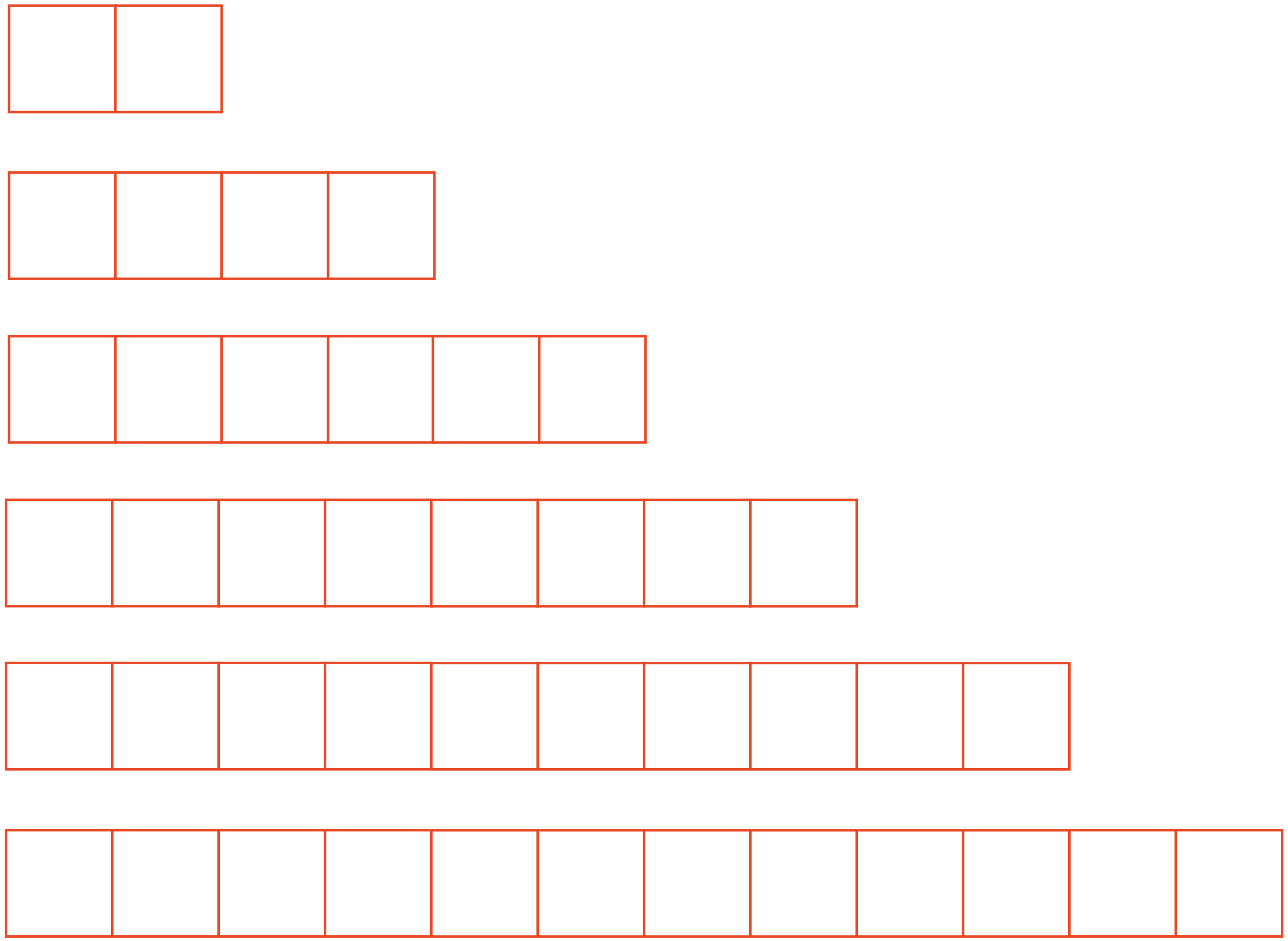
Array 'Add'

```
l.append(1)

l.insert(0, 2)
```

Array Capacity vs Array Size

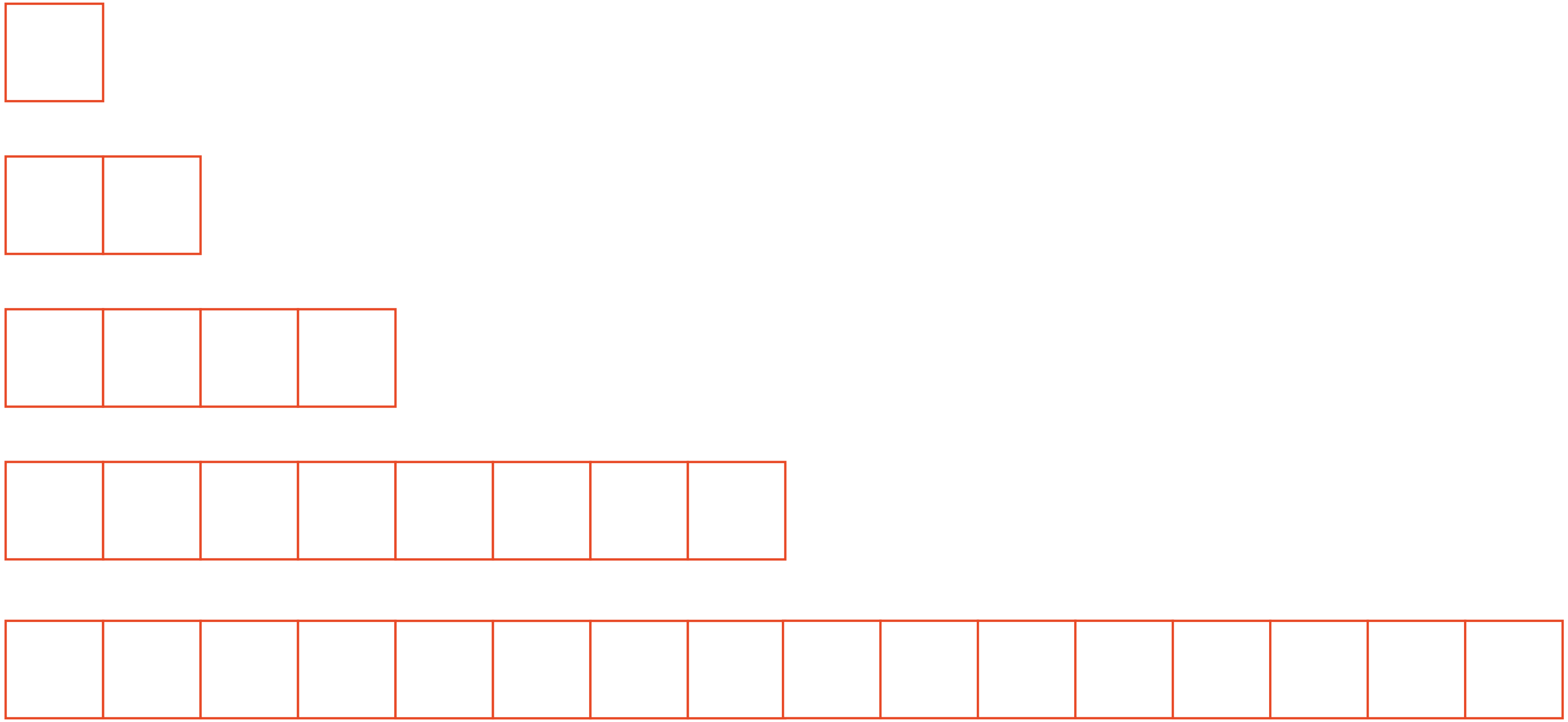
We want to minimize the number of times we have to rebuild an array!

Resize Strategy: +2 elements every resize



Resize Strategy: +2 elements every resize

Resize Strategy: x2 elements every resize



Resize Strategy: x2 elements every resize

Python List Size Implementation

```
memory_size = {}

for length in range(50):
    lst = []
    for length_loop in range(length):
        lst.append(length_loop)
    memory_size[length] = sys.getsizeof(lst)

print(memory_size)
```

```
{0: 56, 1: 88, 2: 88, 3: 88, 4: 88, 5: 120, 6: 120, 7: 120, 8: 120, 9: 184, 10: 184, 11:
184, 12: 184, 13: 184, 14: 184, 15: 184, 16: 184, 17: 248, 18: 248, 19: 248, 20: 248, 21:
248, 22: 248, 23: 248, 24: 248, 25: 312, 26: 312, 27: 312, 28: 312, 29: 312, 30: 312, 31:
312, 32: 312, 33: 376, 34: 376, 35: 376, 36: 376, 37: 376, 38: 376, 39: 376, 40: 376, 41:
472, 42: 472, 43: 472, 44: 472, 45: 472, 46: 472, 47: 472, 48: 472, 49: 472}
```

Python List Size Implementation

```
memory_size = {}

for length in range(50):
    lst = []
    for length_loop in range(length):
        lst.append(length_loop)
    memory_size[length] = sys.getsizeof(lst)

print(memory_size)
```

```
{0: 56, 1: 88, 2: 88, 3: 88, 4: 88, 5: 120, 6: 120, 7: 120, 8: 120, 9: 184, 10: 184, 11:
184, 12: 184, 13: 184, 14: 184, 15: 184, 16: 184, 17: 248, 18: 248, 19: 248, 20: 248, 21:
248, 22: 248, 23: 248, 24: 248, 25: 312, 26: 312, 27: 312, 28: 312, 29: 312, 30: 312, 31:
312, 32: 312, 33: 376, 34: 376, 35: 376, 36: 376, 37: 376, 38: 376, 39: 376, 40: 376, 41:
472, 42: 472, 43: 472, 44: 472, 45: 472, 46: 472, 47: 472, 48: 472, 49: 472}
```

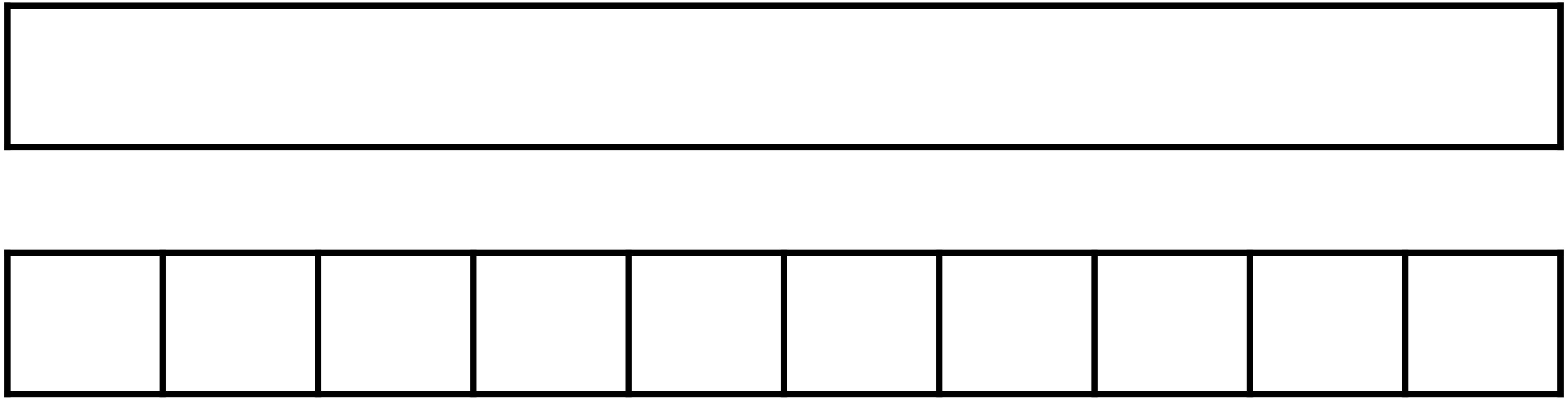
Trivia: Numpy append is really bad!

```
nms = {}
for length in range(50):
    npa = np.array([])
    for length_loop in range(length):
        npa = np.append(npa, length)
    nms[length] = sys.getsizeof(npa)

print(nms)
```

```
{0: 112, 1: 120, 2: 128, 3: 136, 4: 144, 5: 152, 6: 160, 7: 168, 8: 176, 9: 184, 10: 192,
11: 200, 12: 208, 13: 216, 14: 224, 15: 232, 16: 240, 17: 248, 18: 256, 19: 264, 20: 272,
21: 280, 22: 288, 23: 296, 24: 304, 25: 312, 26: 320, 27: 328, 28: 336, 29: 344, 30: 352,
31: 360, 32: 368, 33: 376, 34: 384, 35: 392, 36: 400, 37: 408, 38: 416, 39: 424, 40: 432,
41: 440, 42: 448, 43: 456, 44: 464, 45: 472, 46: 480, 47: 488, 48: 496, 49: 504}
```

Array remove()

```
l = [1, 2, 3, 4, 5, 6, 7]
l.pop()

l.remove(1)

l.pop(3)


```



Python List Remove Implementation

```
test = [1, 2, 3, 4, 5]
print(test, sys.getsizeof(test))
test.remove(3)
print(test, sys.getsizeof(test))
test.remove(4)
print(test, sys.getsizeof(test))
```

```
[1, 2, 3, 4, 5] 104
[1, 2, 4, 5] 104
[1, 2, 5] 104
```

List Abstract Data Type

We've now seen the LinkedList and Array versions of:

Constructor:	`__init__()`	
Insert:	`append(x)`	`insert(i, x)`
Delete:	`remove(x)`	`pop()`
Index	`__getitem__()`	`index(x)`
Size()	`len(list)`	

Which implementation is better?

What do we care about when we write code?

How do we analyze an algorithm?

```
for i in range(n):
    time.sleep(30)
    doStuff()

```

```
for i in range(n):
    for j in range(n):
        doStuff()

```

How do we analyze an algorithm?

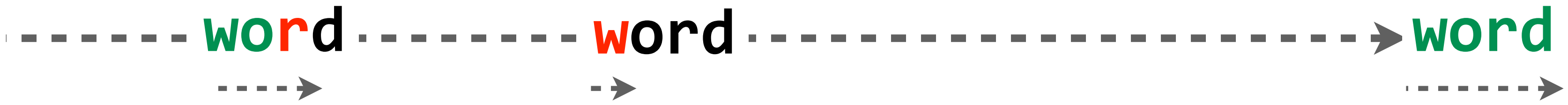
```
for i in range(n):
    for j in range(n/2):
        doStuff()

```

```
for i in range(n):
    for j in range(n):
        doStuff()

```

How do we analyze an algorithm?

P: `word`

T: `There would have been a time for such a word`



How do we analyze an algorithm?

P: **`word`**

T: **`There would have been a time for such a word`**

`word word word word word word word word` **`word`**

`word word word word word word word word`

`word word word word word word word word`

`word word word word word word word word`

`word word word word word word word word`

How do we analyze an algorithm?

P: **aaa**

T: **bbbbbbb**

```
aaa
 aaa
  aaa
   aaa
    aaa
```

P: **bbb**

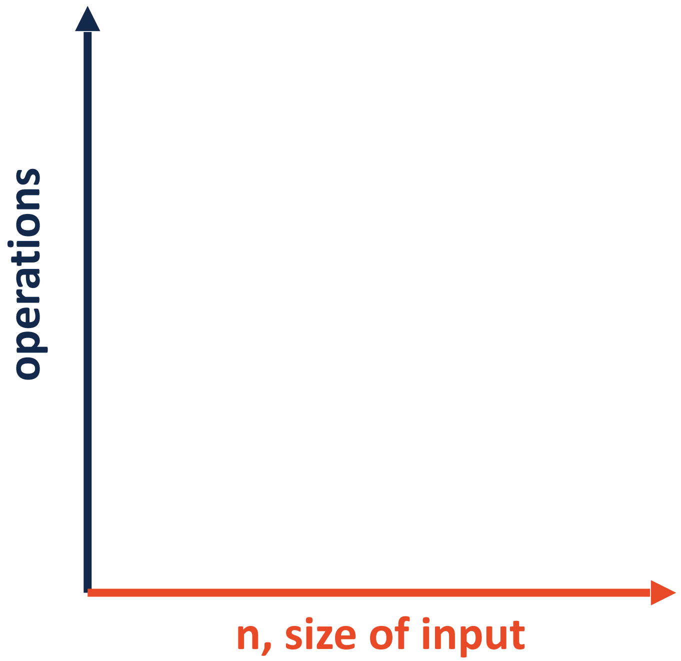
T: **bbbbbbb**

```
bbb
 bbb
  bbb
   bbb
    bbb
```

How do we analyze an algorithm?

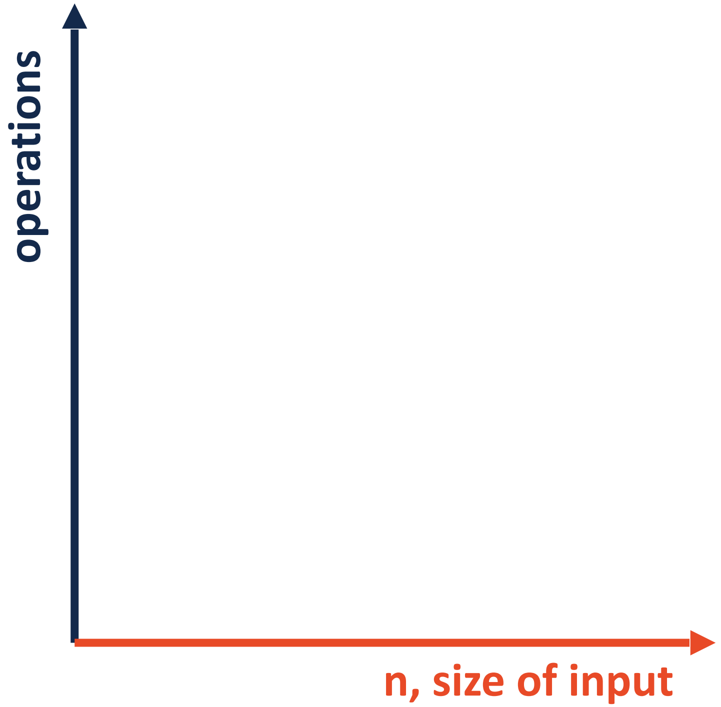
Big-O notation

$f(n)$ is $O(g(n))$ iff $\exists c, k$ such that $f(n) \leq cg(n)\ \forall n > k$



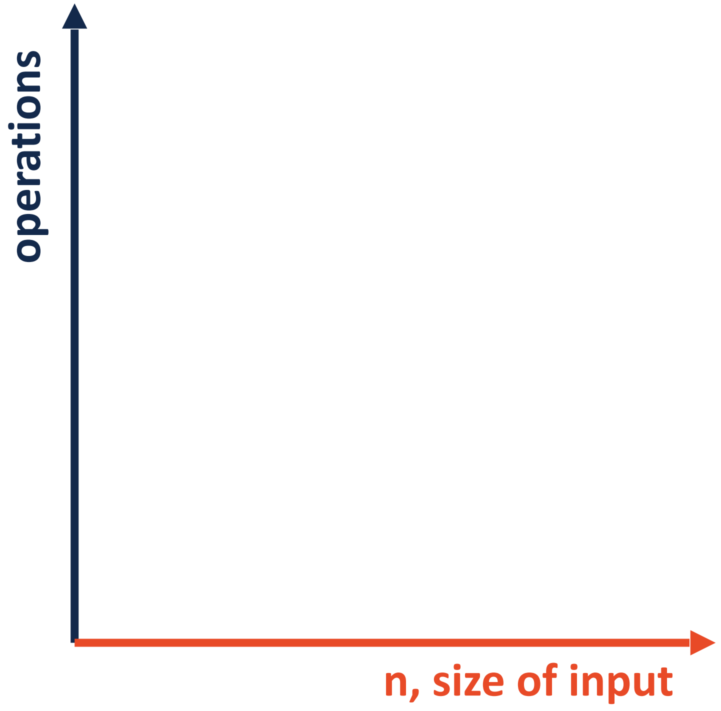
Constant Time, O(1)

```
def constant(n):
    ops = 0
    for i in range(10):
        ops+=1
    return ops

print(constant(5))
print(constant(9001))
```



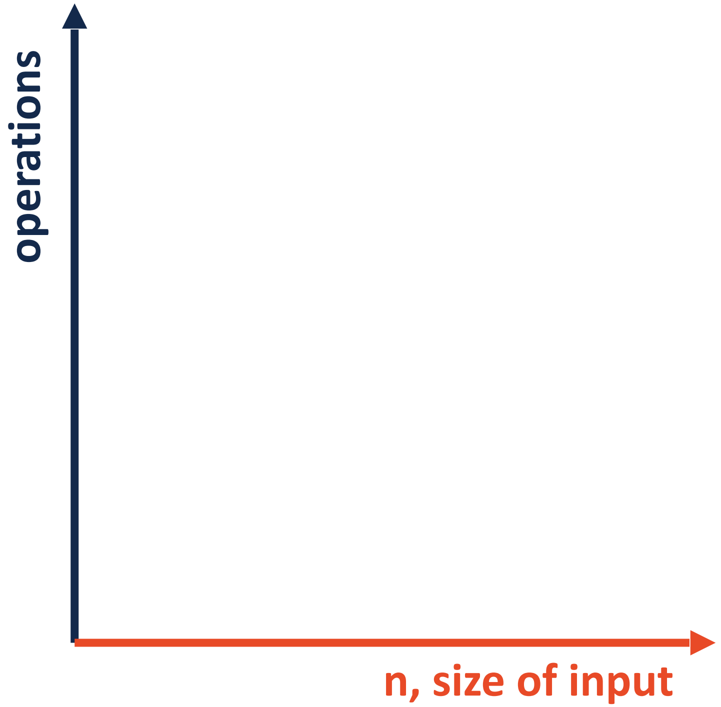
Logarithmic Time, O(log n)

```python
import math
def logarithmic(n):
    ops = 0
    for i in range(int(math.log2(n))):
        ops+=1
    return ops

print(logarithmic(5))
print(logarithmic(9001))
```



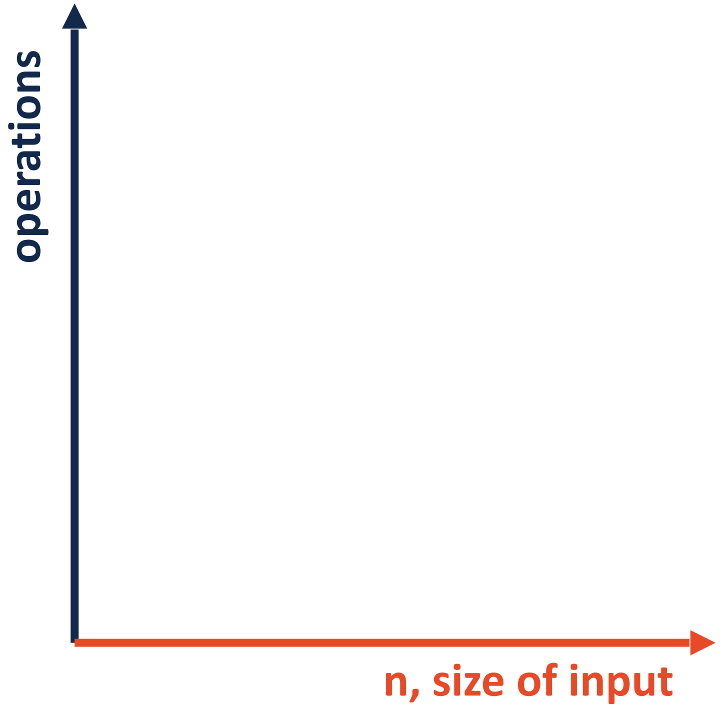
Linear Time, O(n)

```python
def linear(n):
    ops = 0
    for i in range(n):
        ops+=1
    return ops

print(linear(5))
print(linear(9001))
```

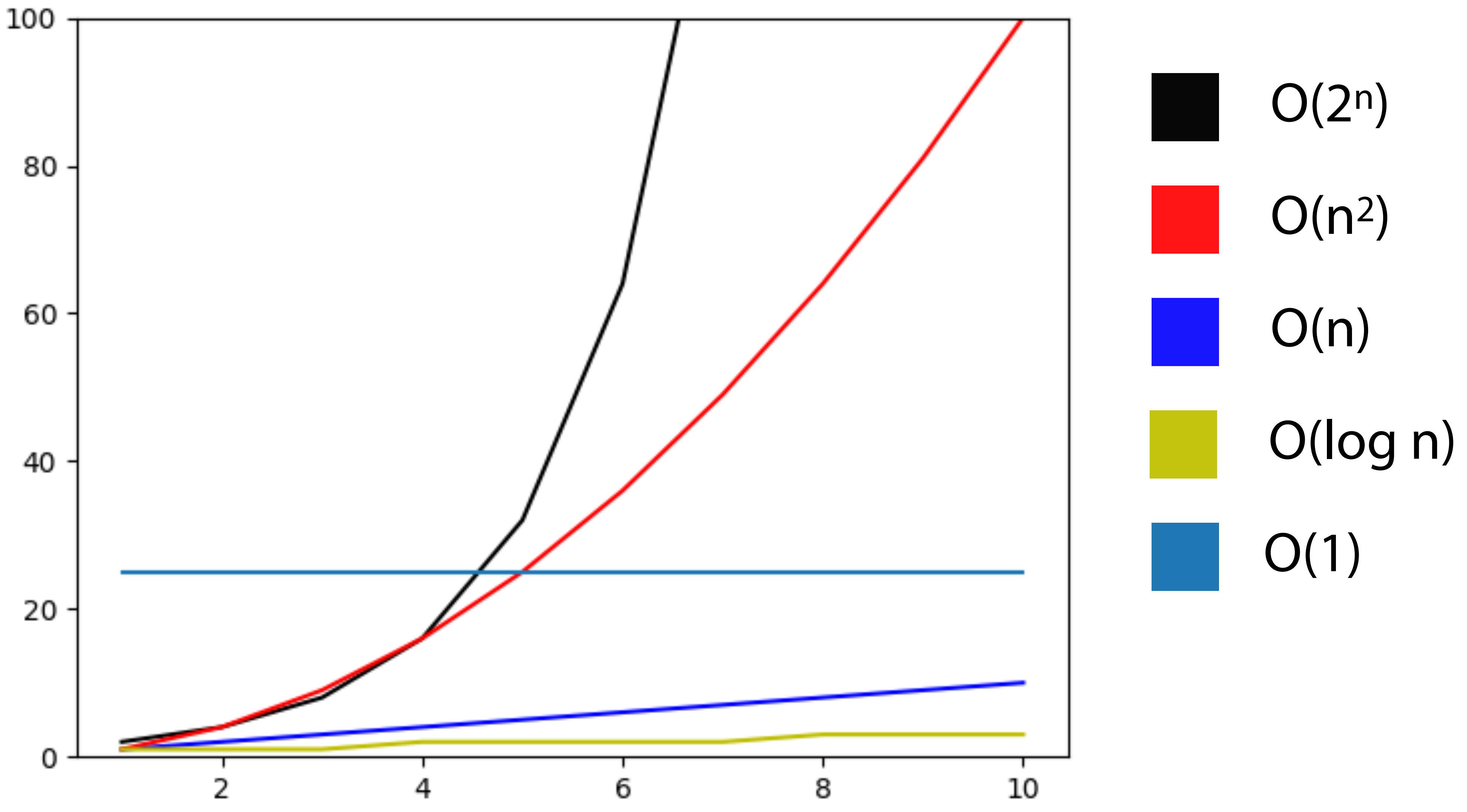


Quadratic Time, $O(n^2)$

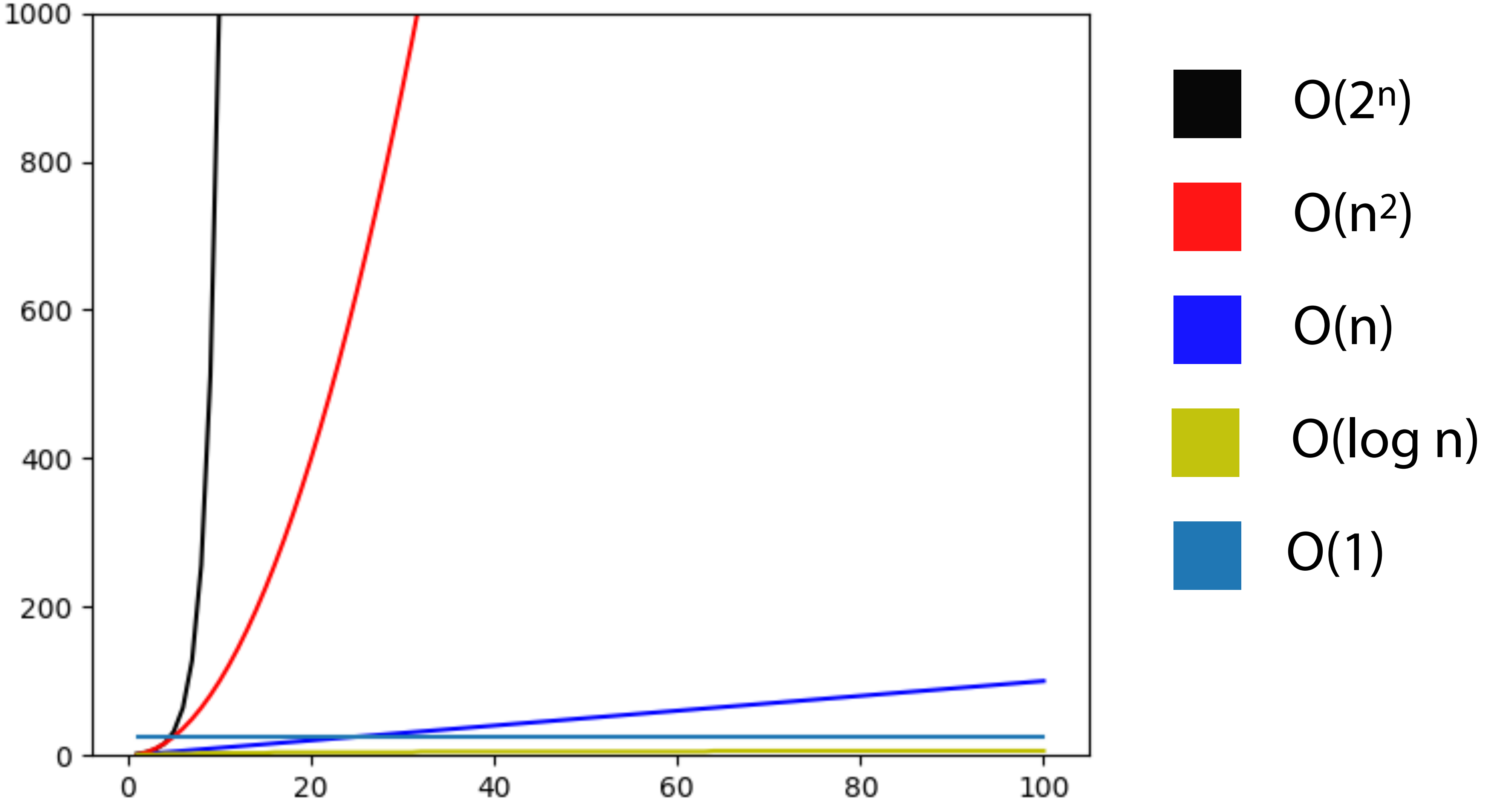
```
# Quadratic Time
def quadratic(n):
    ops = 0
    for i in range(n):
        for j in range(n):
            ops+=1
    return ops

print(quadratic(5))
print(quadratic(9001))
```



Big-O Complexity Classes



Big-O Complexity Classes



Not every operation is created equal

	Time x1 billion	Like
L1 cache reference	0.5 seconds	Heartbeat 💓
Branch mispredict	5 seconds	Yawn 😲
L2 cache reference	7 seconds	Long yawn 😲 😲 😲
Mutex lock/unlock	25 seconds	Make coffee ☕
Main memory reference	100 seconds	Brush teeth
Compress 1K bytes	50 minutes	TV show 📺
Send 2K bytes over 1 Gbps network	5.5 hours	(Brief) Night's sleep 🛌
SSD random read	1.7 days	Weekend
Read 1 MB sequentially from memory	2.9 days	Long weekend
Read 1 MB sequentially from SSD	11.6 days	2 weeks for delivery 📦
Disk seek	16.5 weeks	Semester
Read 1 MB sequentially from disk	7.8 months	Human gestation 🐣
Above two together	1 year	🌍 ☀️
Send packet CA->Netherlands->CA	4.8 years	Ph.D. 🎓

(Care of https://gist.github.com/hellerbarde/2843375)