01: Coding for Performance

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University

Some material taken from:
https://github.com/trekhleb/homemade-machine-learning/tree/master/homemade/neural_network
http://cs231n.github.io/neural-networks-1/

Course Website

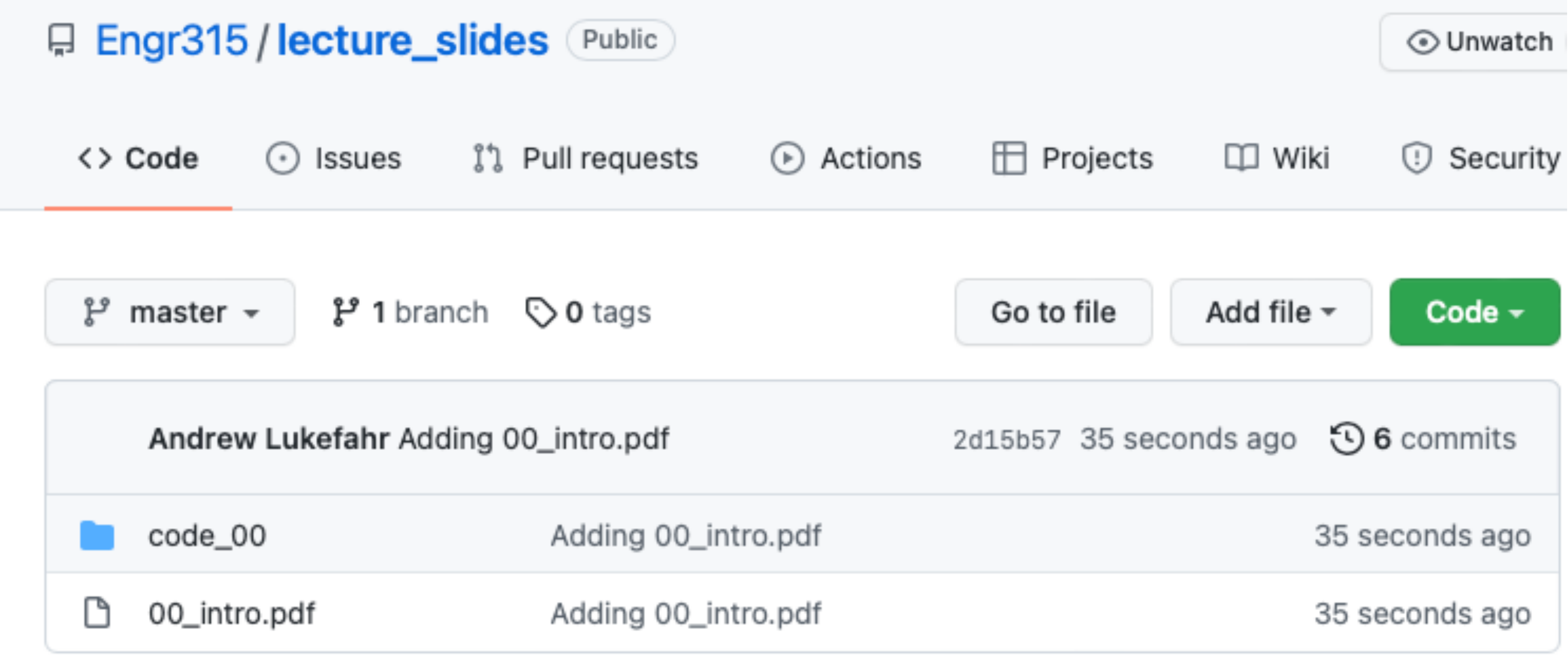
engr315.github.io

Write that down!

Slack? - https://engr-315.slack.com

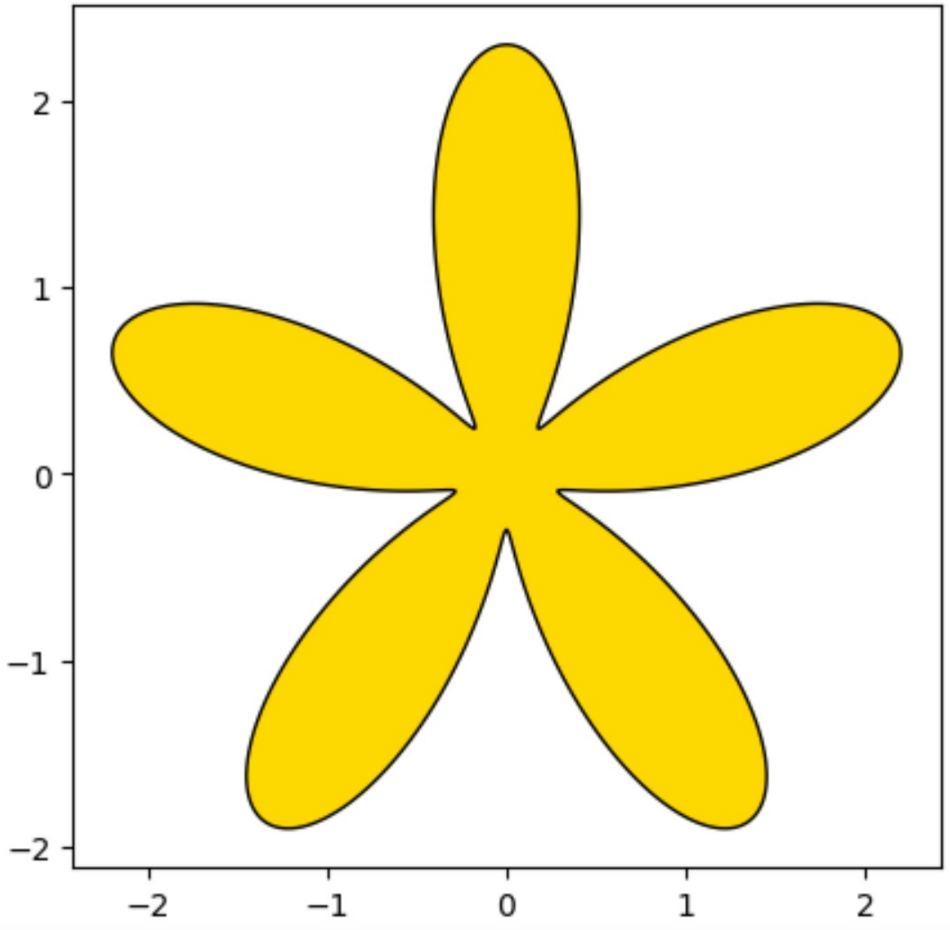
- ~~Thanks Joel~~

I try to post all the code I use in class

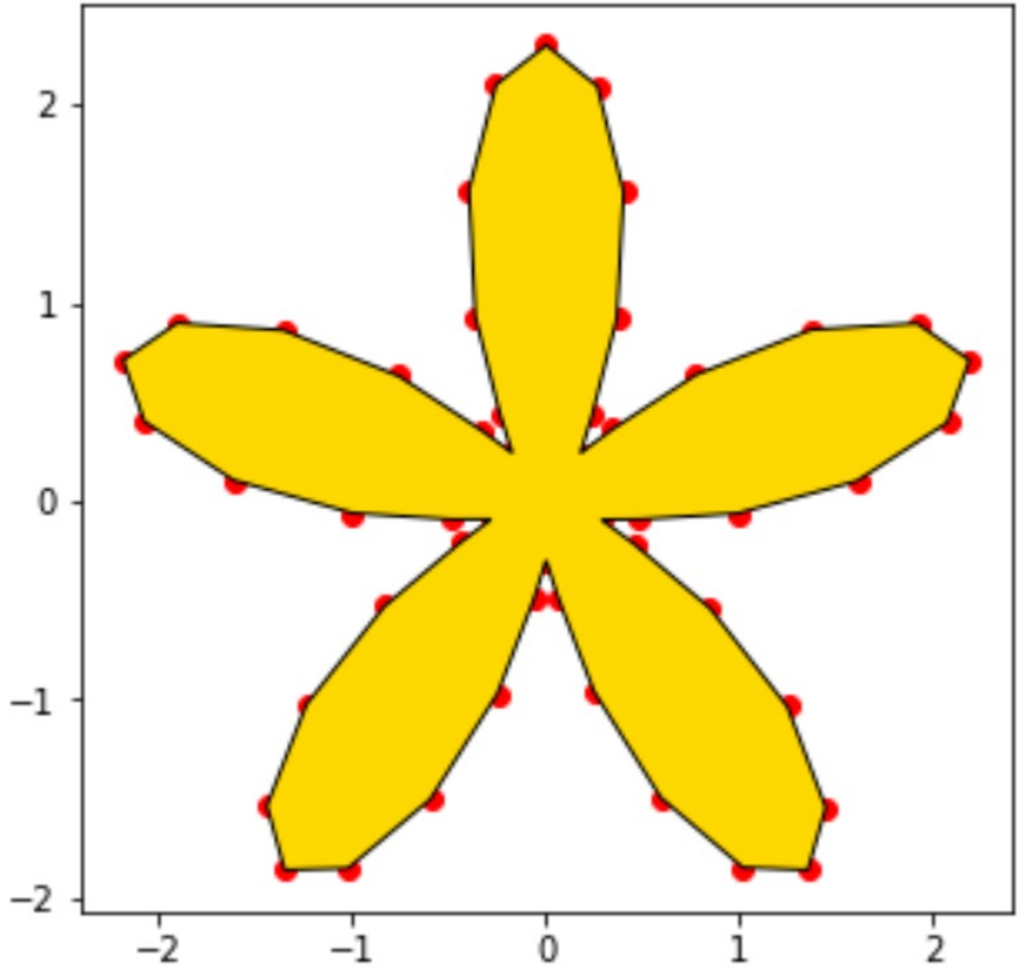


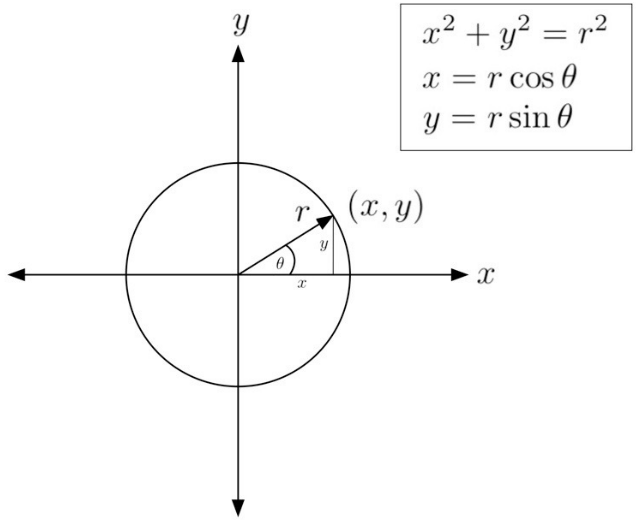
Remind me if (when) I forget.

Project 1: Optimization

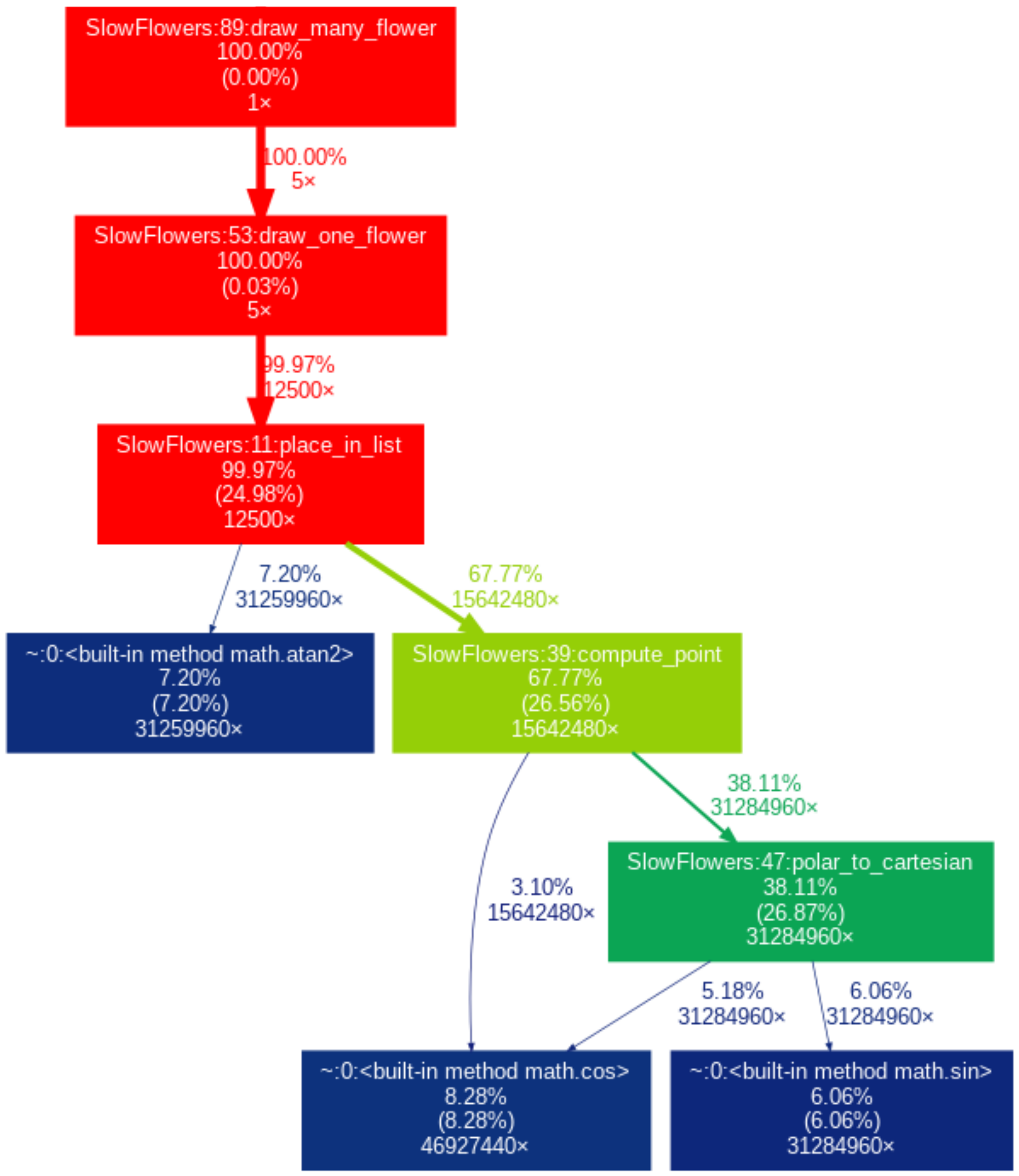


Project 1: Flowers





Project 1: Flowers



Project 1: Flowers

- This is optimized already:

```
oldTheta = atan2(point[0]-x_c, point[1]-y_c)
          r = (a * cos(5 * (theta - (pi/2)))) + (a * 1.3)
```

- You aren’t going to accelerate math functions. Don’t try.
- Figure out how to call it less.

Project 1: “Bonus”

- Bonus: Gets you bonus points!
  - My old best time
- Better Bonus: Gets you even more bonus points!
  - (best time from student)

Good luck!

Code Profiling

- In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization. [Wiki]

Code Profiling can measure

- Program Runtimes
- Function Call Numbers/Runtimes
- Memory Usage
- Instruction Usage
- Others

Profiling guide us on ***where to look*** to reduce runtime

```python
def squares(n):
    if n <= 1:
        return [1]
    else:
        seq = squares(n-1)
        seq.append(n*n)
        return seq
```

```
         40002 function calls (20003 primitive calls) in 0.021 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  20000/1    0.019    0.000    0.021    0.021 <ipython-input-8-50d13c5dd8df>:1(squares)
        1    0.000    0.000    0.021    0.021 <string>:1(<module>)
        1    0.000    0.000    0.021    0.021 {built-in method builtins.exec}
    19999    0.002    0.000    0.002    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Conclusion #1: Function calls are not free!

- Setup/Return overheads with function calls
  - Small
- Recursion: small overheads *x* many calls
  - Can add notable overheads
- Only use recursion if don't care about performance

Cutting recursion buys us ~2x

```python
def squares(n):
    if n <= 1:
        return [1]
    else:
        seq = squares(n-1)
        seq.append(n*n)
        return seq
```

```python
def squares2(n):
    if n <= 1:
        return [1]
    else:
        seq = []
        for i in range(1,n):
            seq.append(i*i)
        return seq
```

```python
import time
import sys
sys.setrecursionlimit(21000)

start_time = time.time()
squares(20000)
end_time = time.time()

# at the end of the program:
print("%f seconds" % (end_time - start_time))
```

```
0.009825 seconds
```

```python
import time

start_time = time.time()
squares2(20000)
end_time = time.time()

# at the end of the program:
print("%f seconds" % (end_time - start_time))
```

```
0.004209 seconds
```

Can we make it go even faster?

```python
def squares2(n):
    if n <= 1:
        return [1]
    else:
        seq = []
        for i in range(1,n):
            seq.append(i*i)
        return seq
```

```python
import time

start_time = time.time()
squares2(20000)
end_time = time.time()

# at the end of the program:
print("%f seconds" % (end_time - start_time))
```

```
0.004209 seconds
```

```python
import numpy as np
def squares3(n):

    seq = np.zeros(n, dtype=np.int)
    for i in range(1, n+1):
        seq[i-1] = i * i
    return seq
```

```python
import time

start_time = time.time()
squares3(20000)
end_time = time.time()

# at the end of the program:
print("%f seconds" % (end_time - start_time))
```

```
0.003960 seconds
```

… And I'm bested!

```python
#Thanks Drason!
def squares4(n):
    return [i * i for i in range(1, n+1)]

start_time = time.time()
squares4(20000)
end_time = time.time()

# at the end of the program:
print("%f seconds" % (end_time - start_time))
```

```
0.003010 seconds
```

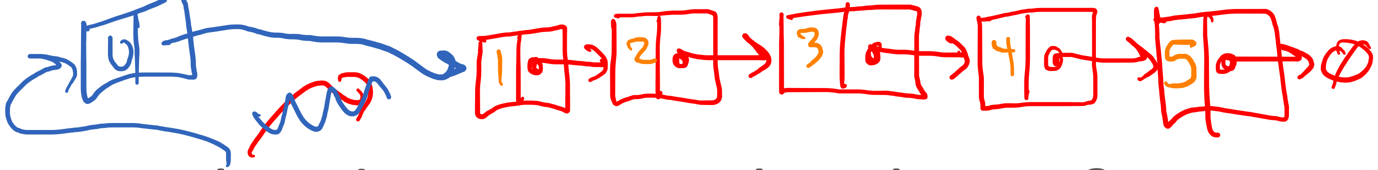
Conclusion #2: memory preallocation *should be* faster

- Numpy reallocates a large contiguous block ☺
  - But also zeros it out. ☹

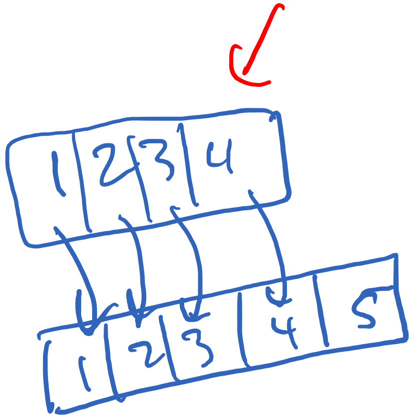
np.empty

- List.append() allocates new memory as needed

- Python's "List Comprehension" is weird.



Array vs. Linked List: Which is faster?



- Randomly accessing a specific element? array
- Appending new values? -> list*

Array vs. Linked List: Random Access

```python
lst = collections.deque(nums)
arr = np.array(nums)
print (lst)
print (arr)
```

```
deque([5, 1, 9, 0, 3, 2, 6, 4, 8, 7])
[5 1 9 0 3 2 6 4 8 7]
```

```python
def traverse( thing, times):
    idx = 0
    for i in range(times):
        nidx = thing[idx]
        print (i, ':', idx, '->', nidx)
        idx = nidx
```

```python
trips = 10
traverse(lst, trips)
```

```
0 : 0 -> 5
1 : 5 -> 2
2 : 2 -> 9
3 : 9 -> 7
4 : 7 -> 4
5 : 4 -> 3
6 : 3 -> 0
7 : 0 -> 5
8 : 5 -> 2
9 : 2 -> 9
```

Array vs. Linked List: Random Access

```python
start_time = time.time()
traverse(lst, trips)
end_time = time.time()

# at the end of the program:
print("True List: %f seconds" % (end_time - start_time))

start_time = time.time()
traverse(arr, trips)
end_time = time.time()

# at the end of the program:
print("Array: %f seconds" % (end_time - start_time))
```

```
0 : 0 -> 5
1 : 5 -> 2
2 : 2 -> 9
3 : 9 -> 7
4 : 7 -> 4
5 : 4 -> 3
6 : 3 -> 0
7 : 0 -> 5
8 : 5 -> 2
9 : 2 -> 9
True List: 0.001251 seconds
0 : 0 -> 5
1 : 5 -> 2
2 : 2 -> 9
3 : 9 -> 7
4 : 7 -> 4
5 : 4 -> 3
6 : 3 -> 0
7 : 0 -> 5
8 : 5 -> 2
9 : 2 -> 9
Array: 0.006385 seconds
```

Array vs. Linked List: Random Access

```python
def traverse( thing, times):
    idx = 0
    for i in range(times):
        idx = thing[idx]

random.seed(1)
sz = 1000000
nums = [x for x in range(sz)]
random.shuffle(nums)
random.shuffle(nums)
lst = collections.deque(nums)
arr = np.array(nums)
trips = 1000

start_time = time.time()
traverse(lst, trips)
end_time = time.time()
print("True List: %f seconds" % (end_time - start_time))

start_time = time.time()
traverse(arr, trips)
end_time = time.time()
print("Array: %f seconds" % (end_time - start_time))

start_time = time.time()
traverse(nums, trips)
end_time = time.time()
print("Python List: %f seconds" % (end_time - start_time))
```

```
True List: 0.037878 seconds
Array: 0.000312 seconds
Python List: 0.000410 seconds
```

Python’s “List” isn’t actually a ”List”

- It’s a list of arrays!

Array vs. Linked List: Sequential Insert

```python
def insert(thing, idx, values):
    print (thing)
    for value in values:
        thing.insert(idx, value)
    print (thing)

random.seed(1)
sz = 10
nums = [x for x in range(sz)]
random.shuffle(nums)
random.shuffle(nums)
lst = collections.deque(nums)
arr = np.array(nums)

idxs = int(sz/2)
insert(nums, idxs, [-1,-2,-3,-4])
```

```
[5, 1, 9, 0, 3, 2, 6, 4, 8, 7]
[5, 1, 9, 0, 3, -4, -3, -2, -1, 2, 6, 4, 8, 7]
```

Array vs. Linked List: Sequential Insert of 1M elements

```
Insert at:  0

True List: 0.000085 seconds
Array: 0.335853 seconds
Python List: 0.115629 seconds

Insert at:  750000

True List: 0.054327 seconds
Array: 0.336377 seconds
Python List: 0.022257 seconds
```

Big O Complexity

- Computational time complexity describes the change in the runtime of an algorithm, depending on the change in the input data's size.

- "How much does an algorithm’s performance change when the amount of input data changes?"

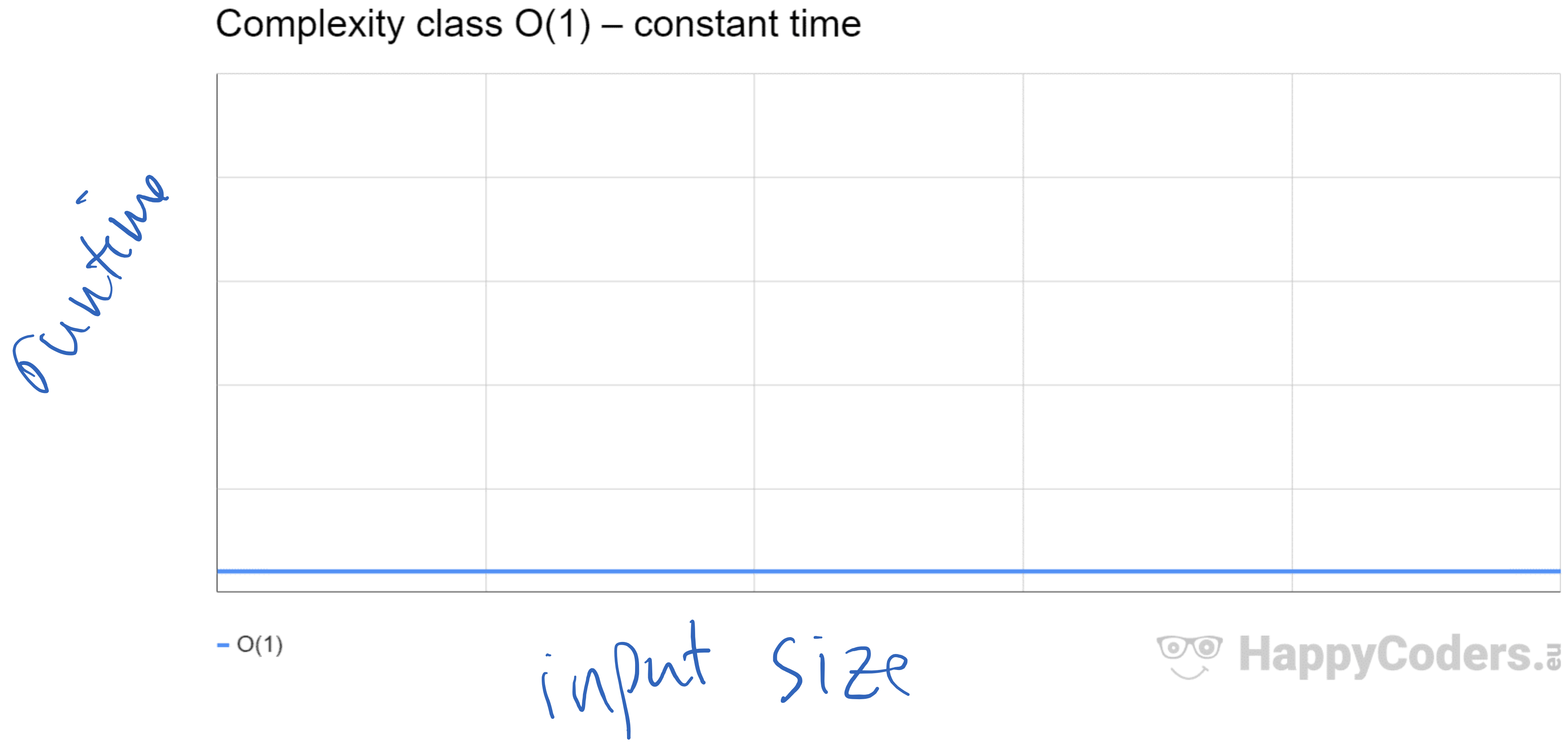
Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

O(1) – Constant Time

- “big O of 1”
- Runtime is constant, regardless of input size
- Example: x = array[n]

Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

O(1) – Constant Time



Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

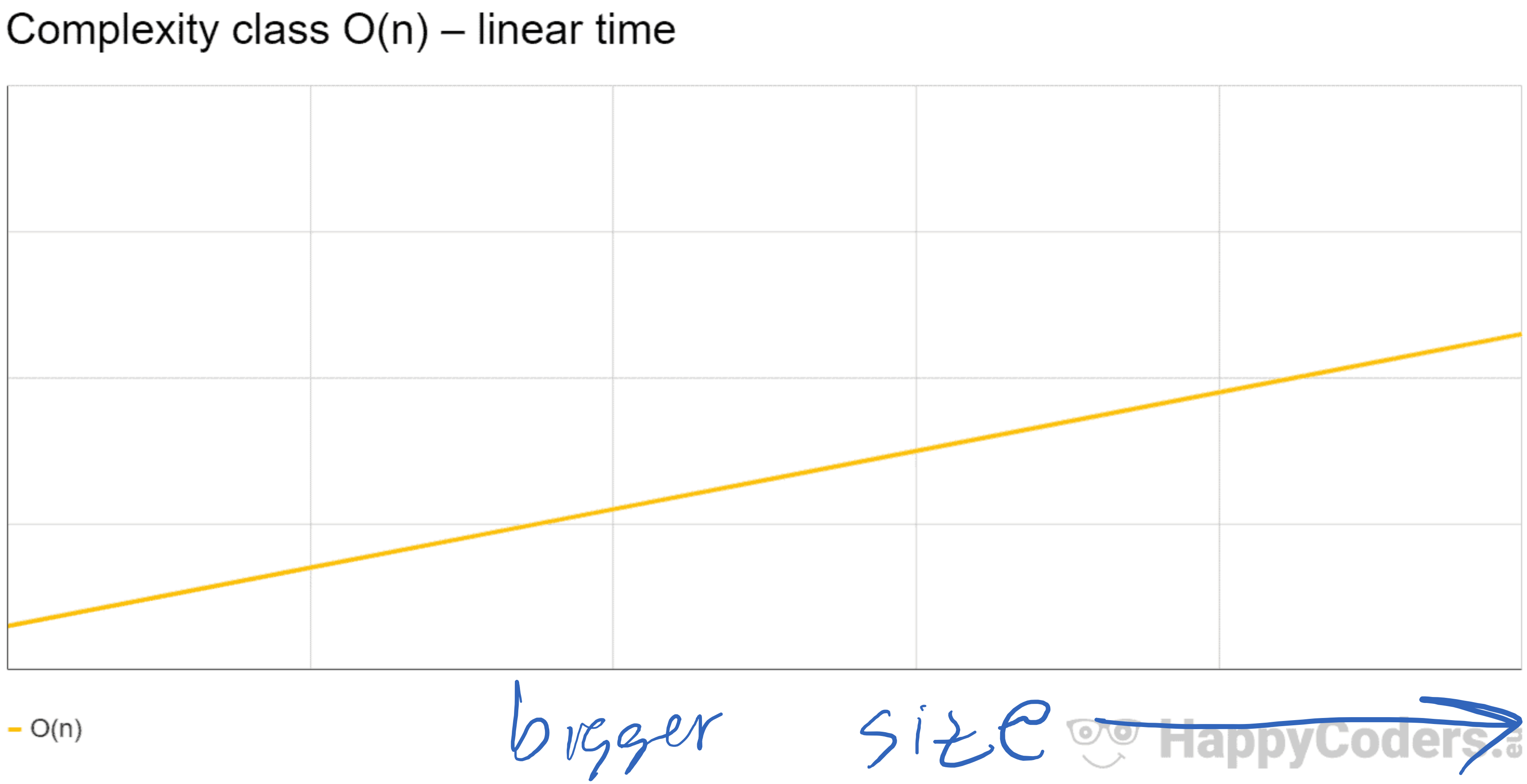
O(n) – Linear Time

- “big O of n”
- Runtime grows linearly with input size
- Example: Linked Lists!

Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

O(n) – Linear Time

more time



bigger size

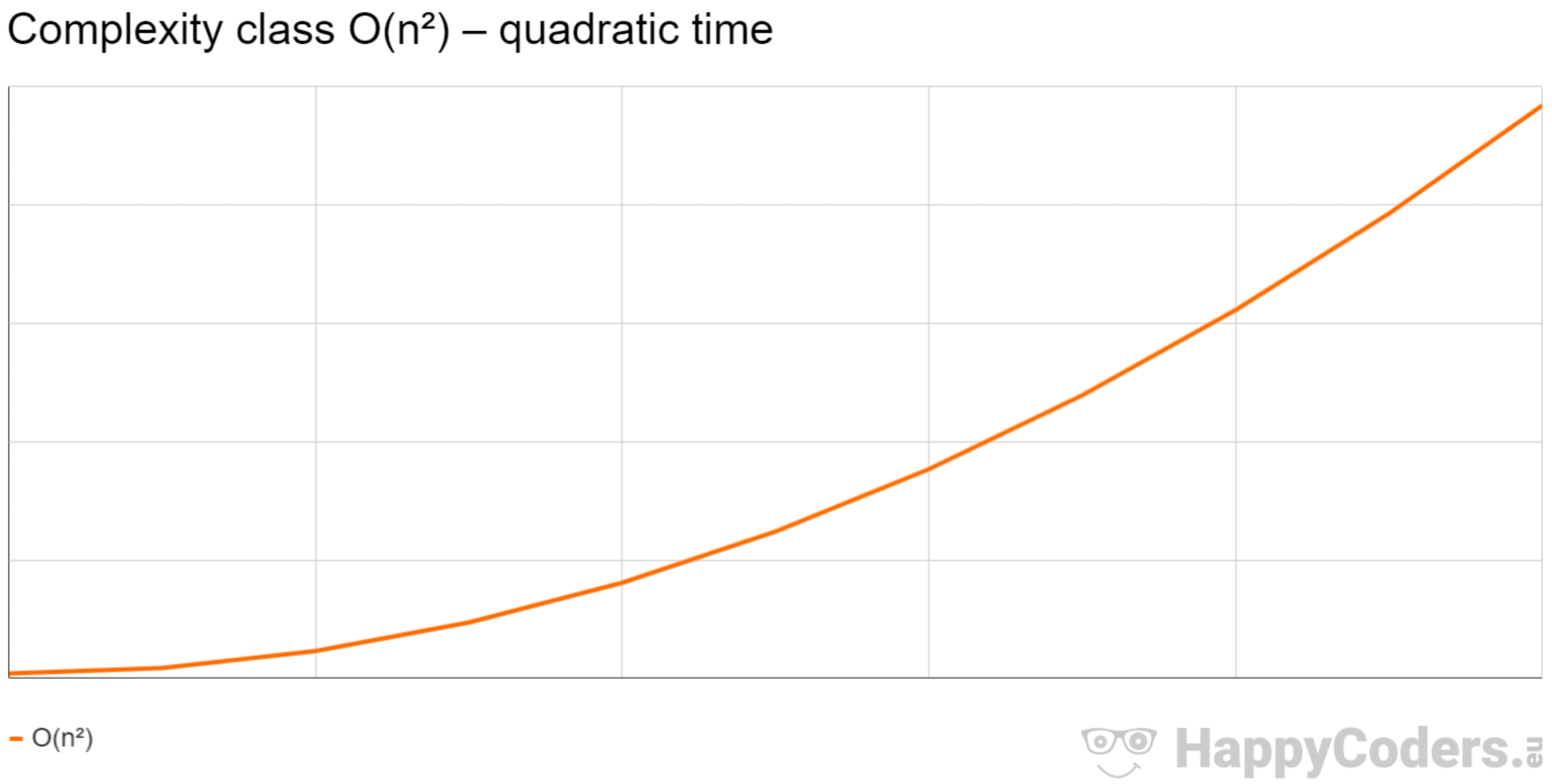
Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

O(n^2) – Quadratic Time

- “big O of n squared”
- Runtime grows linearly with square of the input size
- Example: Bubble Sort
  - haystack.sort(low->high)

Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

$O(n^2)$ – Quadratic Time



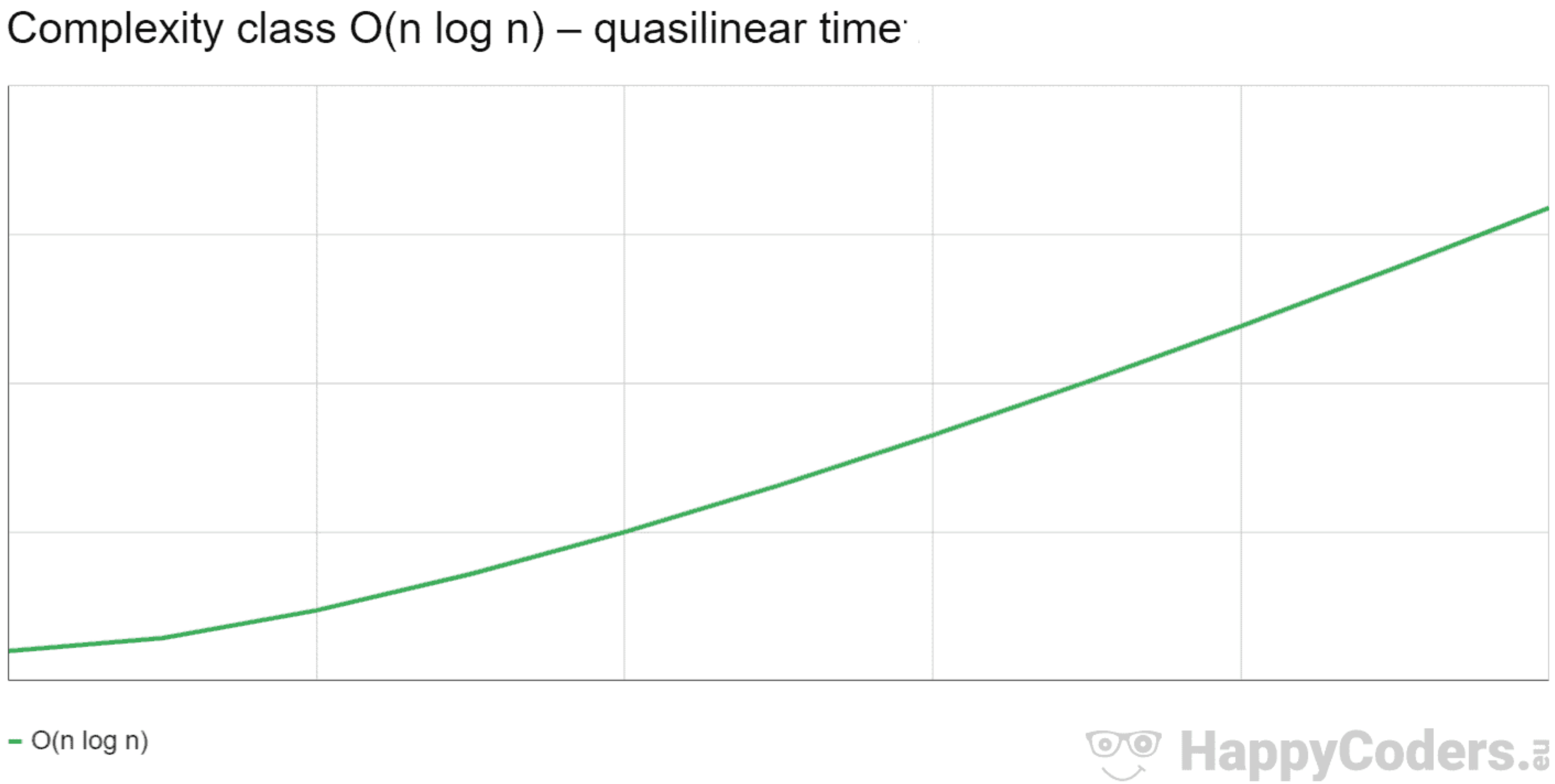
Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

O(n log n) – Quasilinear Time

- "big O of n log n"
- Runtime grows linearly and logarithmically with the input size
- Example: Good Sort
  - haystack.sort(low->high)

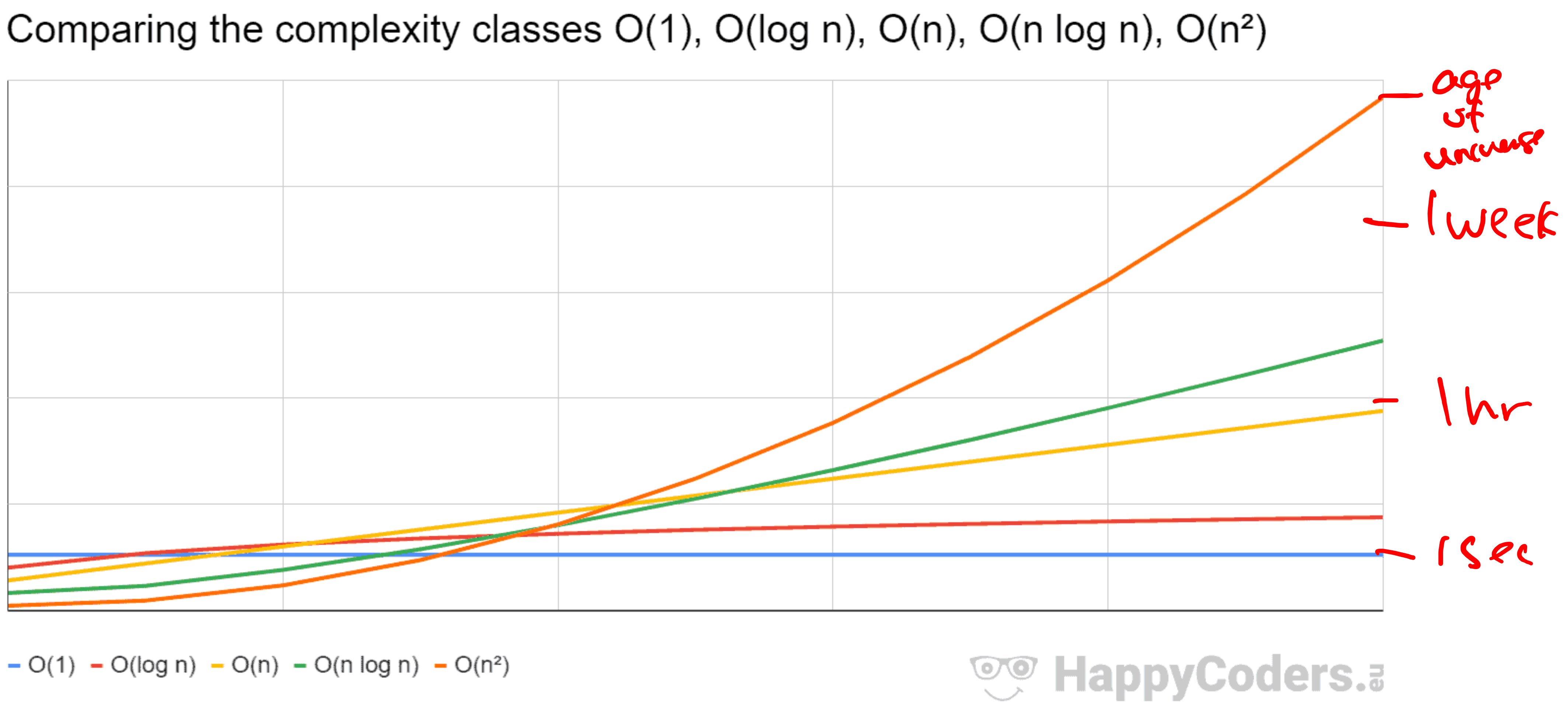
Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

O(n log n) – Quasilinear Time



Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

O() Complexities



Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

Conclusion #3: Think about your data structure!

- How will you be accessing your data?
  - Randomly? Sequentially?
- How will you up updating your data?

- Pick a data structure to minimize overheads for your access patterns

Find: The needle in the haystack.

Find: The needle in the haystack.

```python
def find_ignore_case( needle, haystack):
    results = []
    for hi in range(len(haystack)):
        match = True
        for ni in range(len(needle)-1):
            h = haystack[hi + ni].lower()
            n = needle[ni].lower()
            if h != n:
                match=False
        if match:
            results.append(hi)
    return results
```

- No libraries!

```python
sz=20
haystack = random_str(sz)
needle = haystack[int(sz/2):int(sz/2)+2]
results = find_ignore_case(needle, haystack)

print (needle)
print (haystack)
print (results)

```

```
sk
eiPPzDAnWiskaumnqYpl
[10]
```

Find: The needle in the haystack.

```python
def find_ignore_case( needle, haystack):
    results = []
    for hi in range(len(haystack)-len(needle)):
        match = True
        for ni in range(len(needle)):
            h = haystack[hi + ni].lower()
            n = needle[ni].lower()
            if h != n:
                match=False
        if match:
            results.append(hi)
    return results

random.seed(1)
sz=1000000
haystack = random_str(sz)
needle = haystack[int(sz/2):int(sz/2)+2]

start_time = time.time()
results = find_ignore_case(needle, haystack)
end_time = time.time()
print("True List: %f seconds" % (end_time - start_time))
```

```
True List: 1.109001 seconds
```

Find: The needle in the haystack.

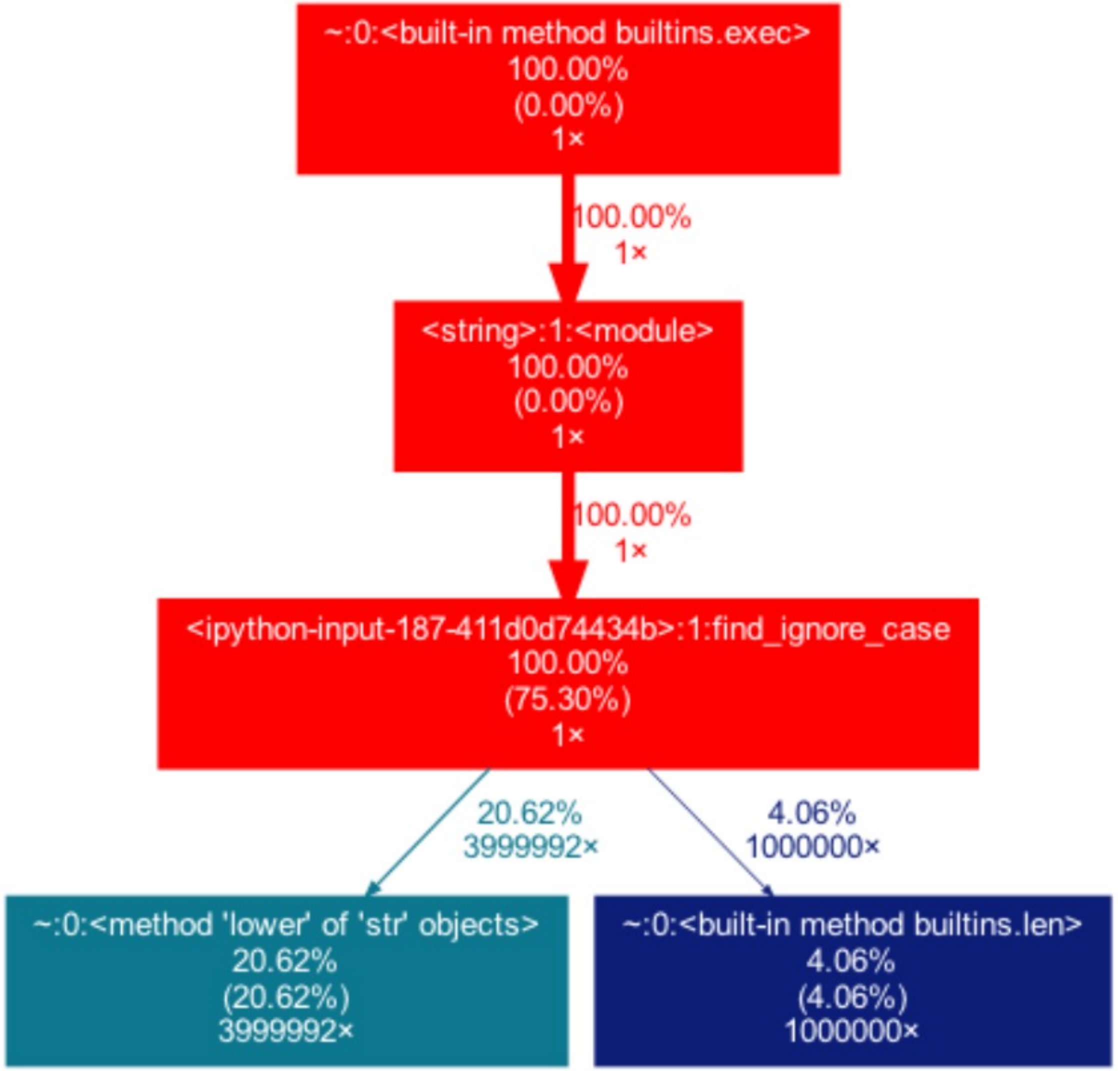
```
import cProfile
cProfile.run('find_ignore_case(needle, haystack)')
```

```
         5001486 function calls in 1.715 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.276    1.276    1.715    1.715 <ipython-input-187-411d0d74434b>:1(find_ignore_case)
        1    0.000    0.000    1.715    1.715 <string>:1(<module>)
        1    0.000    0.000    1.715    1.715 {built-in method builtins.exec}
  1000000    0.072    0.000    0.072    0.000 {built-in method builtins.len}
     1490    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
  3999992    0.367    0.000    0.367    0.000 {method 'lower' of 'str' objects}
```

Find: The needle in the haystack.



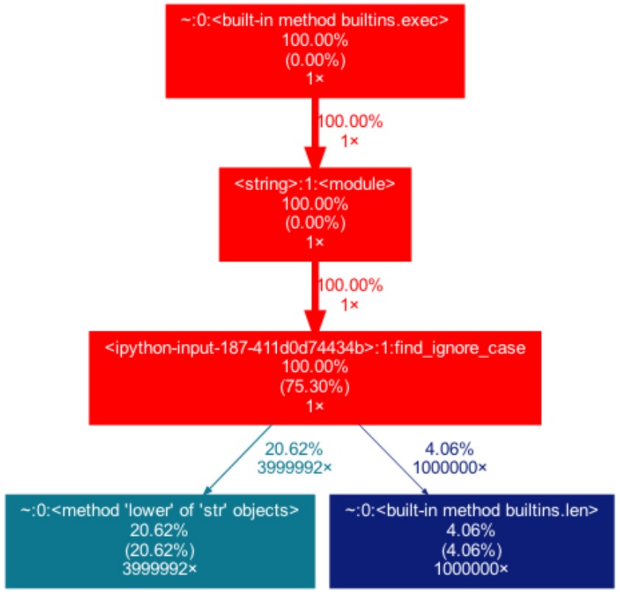
Find: The needle in the haystack.

```python
def find_ignore_case( needle, haystack):
    results = []
    for hi in range(len(haystack)-len(needle)):
        match = True
        for ni in range(len(needle)):
            h = haystack[hi + ni].lower()
            n = needle[ni].lower()
            if h != n:
                match=False
        if match:
            results.append(hi)
    return results

random.seed(1)
sz=1000000
haystack = random_str(sz)
needle = haystack[int(sz/2):int(sz/2)+2]

start_time = time.time()
results = find_ignore_case(needle, haystack)
end_time = time.time()
print("True List: %f seconds" % (end_time - start_time))
```

```
True List: 1.109001 seconds
```



- No libraries!

Find: The needle in the haystack.

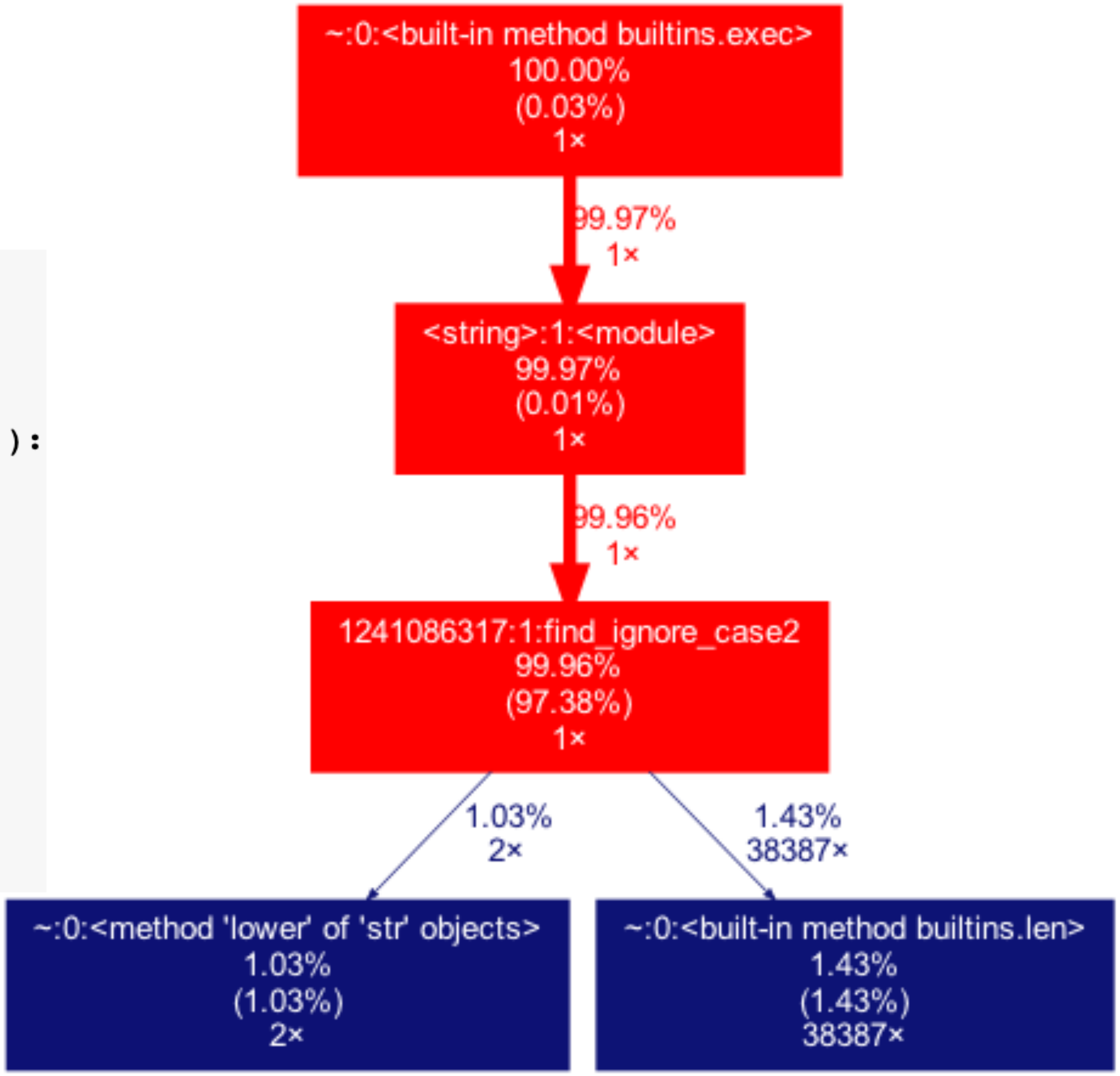
```python
def find_ignore_case( needle, haystack):
    results = []
    for hi in range(len(haystack)-len(needle)):
        match = True
        for ni in range(len(needle)):
            h = haystack[hi + ni].lower()
            n = needle[ni].lower()
            if h != n:
                match=False
        if match:
            results.append(hi)
    return results
```

```python
def find_ignore_case2( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    for hi in range(len(haystack)-len(needle)):
        match = True
        for ni in range(len(needle)):
            h = haystack[hi + ni]#.lower()
            n = needle[ni]#.lower()
            if h != n:
                match=False
                break # new
        if match:
            results.append(hi)
    return results
```

```
Find: 0.917540 seconds
Find2: 0.440155 seconds
```

Anything else?

```
def find_ignore_case2( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    for hi in range(len(haystack)-len(needle)):
        match = True
        for ni in range(len(needle)):
            h = haystack[hi + ni]#.lower()
            n = needle[ni]#.lower()
            if h != n:
                match=False
                break # new
        if match:
            results.append(hi)
    return results
```



```python
def find_ignore_case3( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    r = range(len(needle)) # new

    for hi in range(len(haystack)-len(needle)):
        match = True

        if haystack[hi] == needle[0]:
            for ni in r: # update
                h = haystack[hi + ni]#.lower()
                n = needle[ni]#.lower()
                if h != n:
                    match=False
                    break # new
            if match:
                results.append(hi)
    return results
```

```
Find: 0.370030 seconds
Find2: 0.057817 seconds
Find3: 0.053763 seconds
```

[New Mac Times]

```python
def find_ignore_case4( needle, haystack):
    results = []
    needle = needle.lower() # new
    haystack = haystack.lower() # new
    r = range(len(needle)-1) # new

    for hi in range(len(haystack)-len(needle)):
        #match = False

        if haystack[hi] == needle[0]:
            for ni in r: # update
                h = haystack[hi + ni]#.lower()
                n = needle[ni]#.lower()
                if h == n: # new
                    #match=False
                    results.append(hi) # new
                    break # new
            #if match:
                #results.append(hi)
    return results
```

```
Find: 0.259516 seconds
Find2: 0.057128 seconds
Find3: 0.053053 seconds
Find4: 0.048197 seconds
```

Q: How is this so much faster?

```
def find_ignore_case5( needle, haystack):
    return [haystack.find(needle)]
```

Using built-in libraries is usually the fastest…

```
def find_ignore_case5( needle, haystack):
    return [haystack.find(needle)]
```

```
Find: 0.259516 seconds
Find2: 0.057128 seconds
Find3: 0.053053 seconds
Find4: 0.048197 seconds
Find5: 0.000172 seconds
```

Coding for Performance

Engr 315: Hardware / Software Codesign
Andrew Lukefahr
Indiana University