02: C Interfaces

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University

Slack - https:///e315-fall2022.slack.com

- Thanks Nicole!

- https://join.slack.com/t/slack-9wx7802/shared_invite/zt-1ep5fpz2b-bNXM9830VdBlKQGcMb3Quw

Send as Canvas mesg.

Office Hours

- Andrew: M/W 11-12 in 2032 Luddy or 4111 Luddy

Nicole Miller - ncm1@iu.edu

Caleb Cook - cookce@iu.edu

Office Hours:

- Tuesday: 10am-1pm
- Wednesday: 5pm-8pm
- Thursday: 5pm-8pm
- Friday: 2pm-5pm

Location: 3111 Luddy Hall (one floor below 4111)
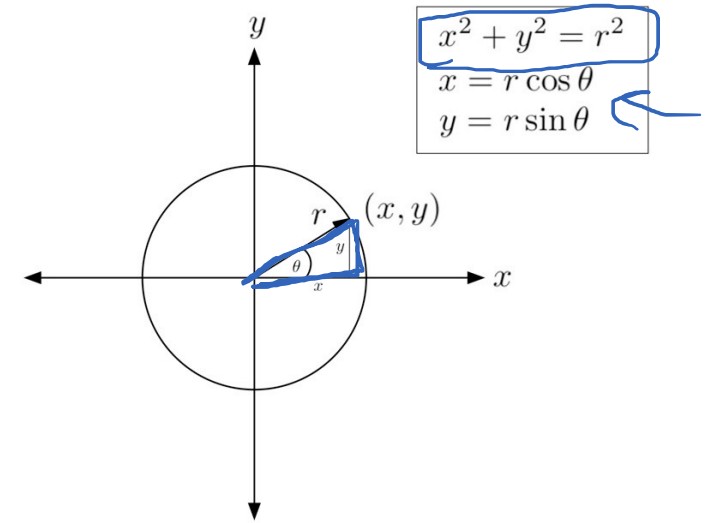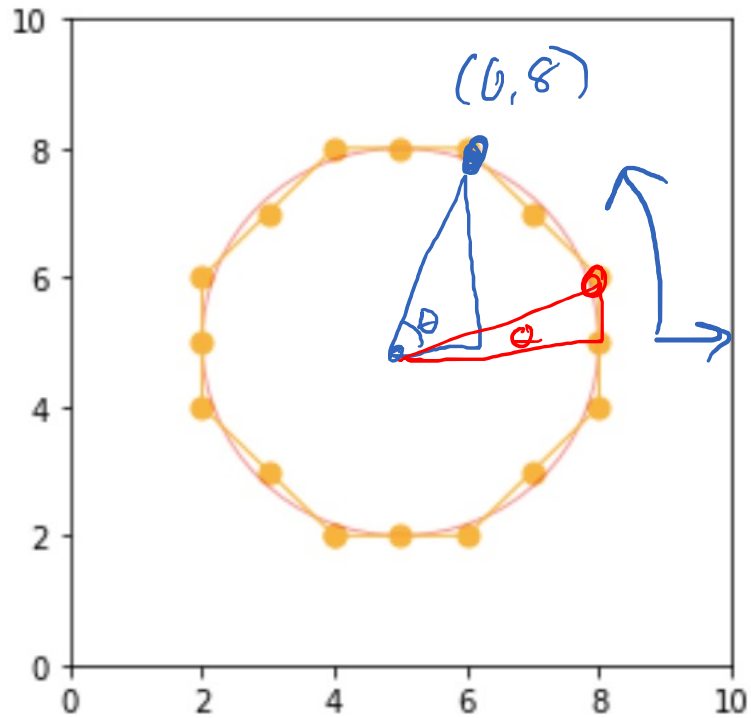
Syllabus

Course Website

engr315.github.io

Write that down!

Announcements

- Slack
  - Contact Joe josbella@iu.edu if questions
- Office Hours:
  - Me: 4-5 pm   T/H
  - Mike: M   5-7 pm
           F   ~~8-10~~
                    10-12 am

link in
Canvas
Chat

Project 1: Circles

$(0, 8)$

$$x^2 + y^2 = r^2$$
$$x = r \cos \theta$$
$$y = r \sin \theta$$

(x, y)

Project 1: Circles

- This is optimized already:

```python
def computeTheta(self, x,y, x_centre, y_centre):
    return math.atan2(x-x_centre, y-y_centre)
```

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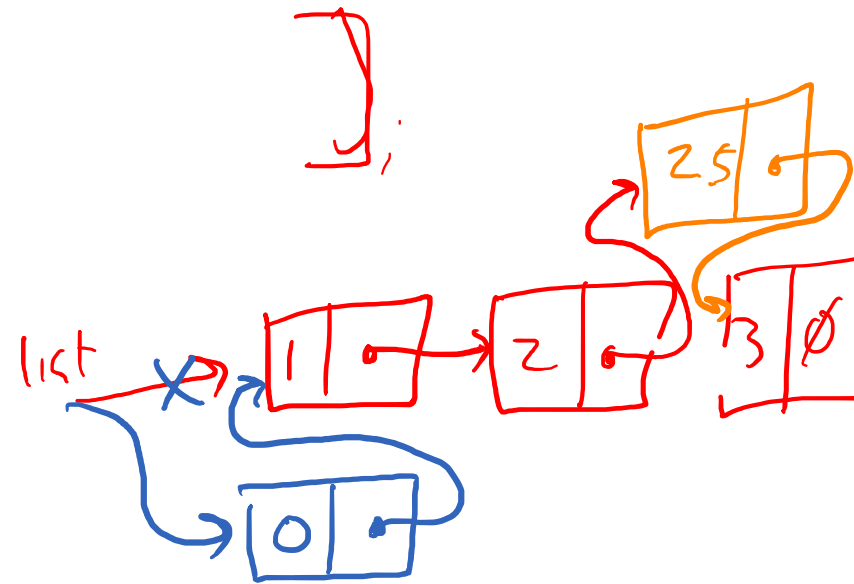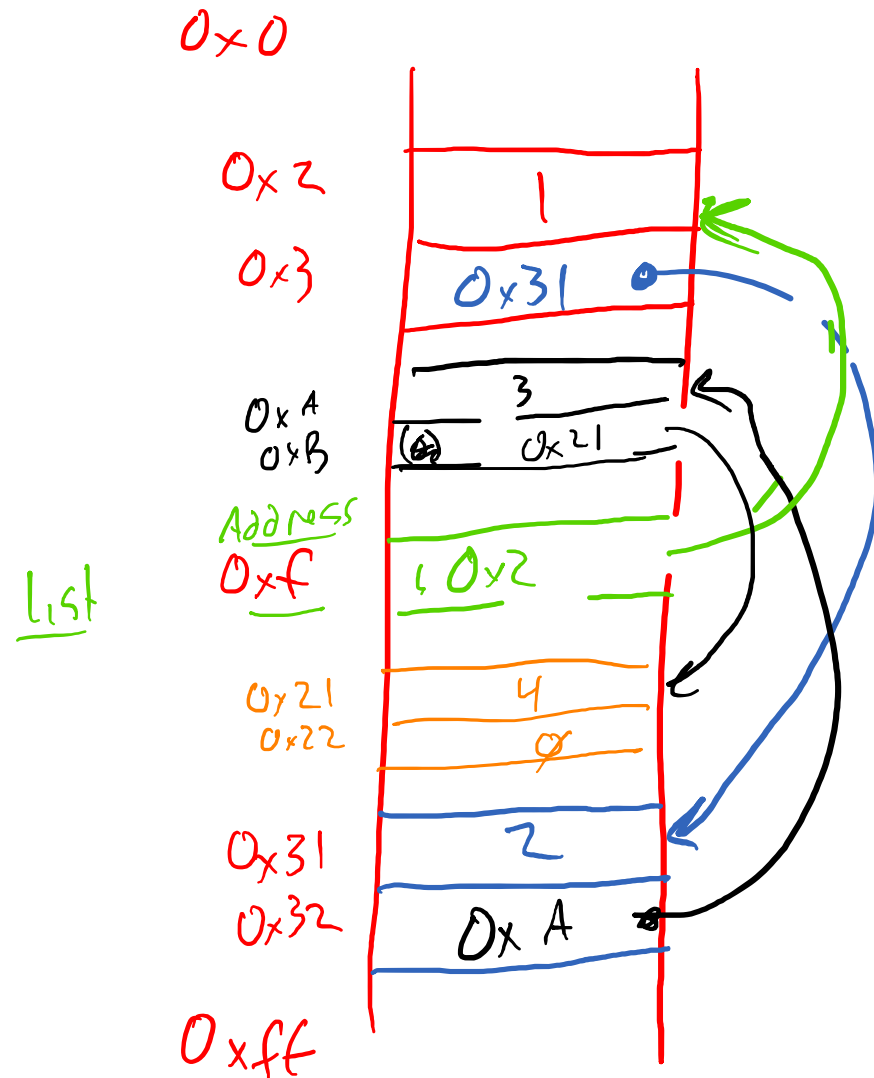- Figure out how to call it less.

Project 2: Accelerate Exp. Moving Avg.

- Need Pynq board for P2.

- Hope to have those ready by Wednesday.

Linked Lists

- Who was the professor when you learned about linked lists?

linked list = [1, 2, 3,];

0x0

0x2   1
0x3   0x31

0xA   3
0xB   0x21

Address
0xf   , 0x2

List

0x21   4
0x22   0

0x31   2
0x32   0x A

0xff

25

list
1 0   2 0   3 0

0

Array vs. Linked List: Random Access

```python
1  lst = collections.deque(nums)
2  arr = np.array(nums)
3  print (lst)
4  print (arr)
```

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

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

```python
1  trips = 10
2  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
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
```

Array vs. Linked List: Sequential Insert

```python
1  def insert(thing, idx, values):
2      print (thing)
3      for value in values:
4          thing.insert(idx, value)
5      print (thing)
6
7  random.seed(1)
8  sz = 10
9  nums = [x for x in range(sz)]
10 random.shuffle(nums)
11 random.shuffle(nums)
12 lst = collections.deque(nums)
13 arr = np.array(nums)
14
15 idxs = int(sz/2)
16 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

```
Insert at:   0

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


Insert at:   750000        ~3/2

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

Let's plot that.

X-axis:

Insert Times 1,10,100,1000(1e3), 1e4, 1e5,1e6,1e7

Y-axis:

Total runtime for list insert
Total runtime for array insert

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?"

O(1) – Constant Time

- "big O of 1"

- Runtime is constant, regardless of input size

- Example: x = array[n]

O(1) – Constant Time
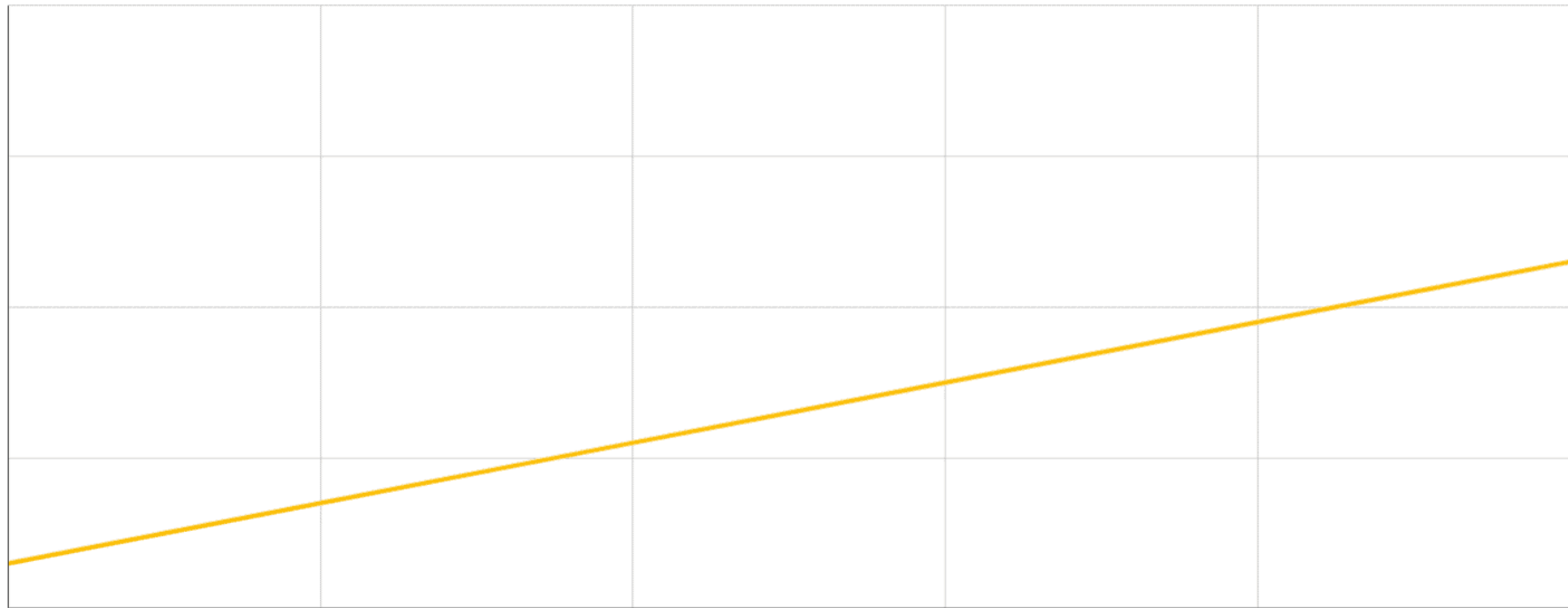

Complexity class O(1) – constant time

— O(1)

O(n) – Linear Time

- "big O of n"

- Runtime grows linearly with input size

- Example: Inserting into the head of a linked list

O(n) – Linear Time

Complexity class O(n) – linear time

more time

bigger size

— O(n)

Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/
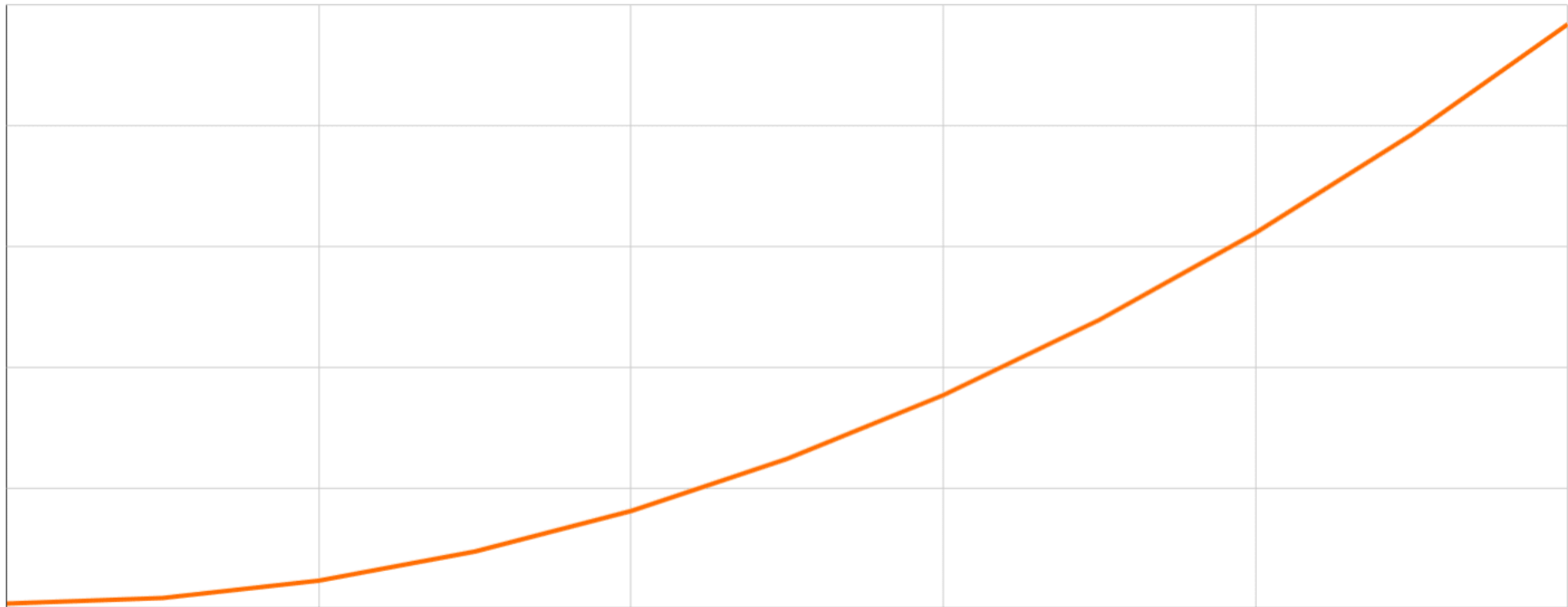
O(n^2) – Quadratic Time

- "big O of n squared"

- Runtime grows linearly with square of the input size

- Example: Bubble Sort
  - haystack.sort(low->high)

O(n²) – Quadratic Time

Complexity class O(n²) – quadratic time

─ O(n²)

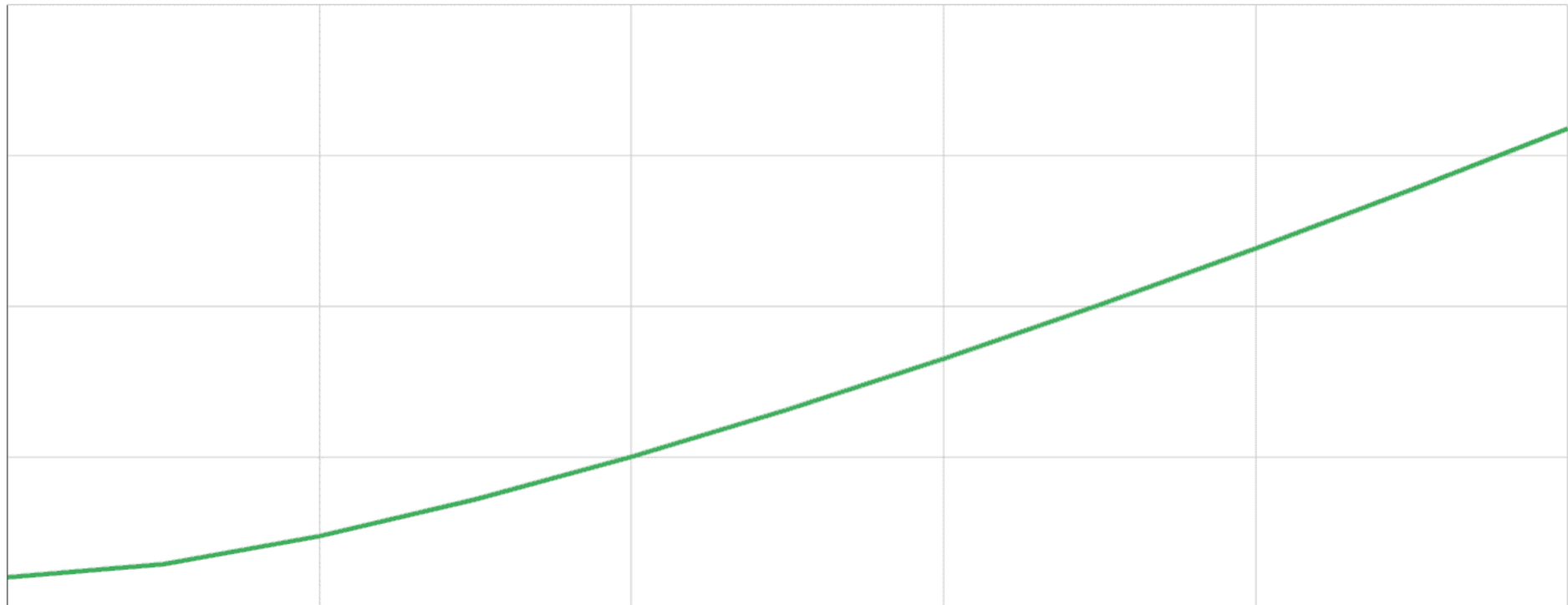Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/

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)

O(n log n) – Quasilinear Time

Complexity class O(n log n) – quasilinear time

– O(n log n)

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

O() Complexities

Comparing the complexity classes O(1), O(log n), O(n), O(n log n), O(n²)

O(1) O(log n) O(n) O(n log n) O(n²)

Material taken from: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/
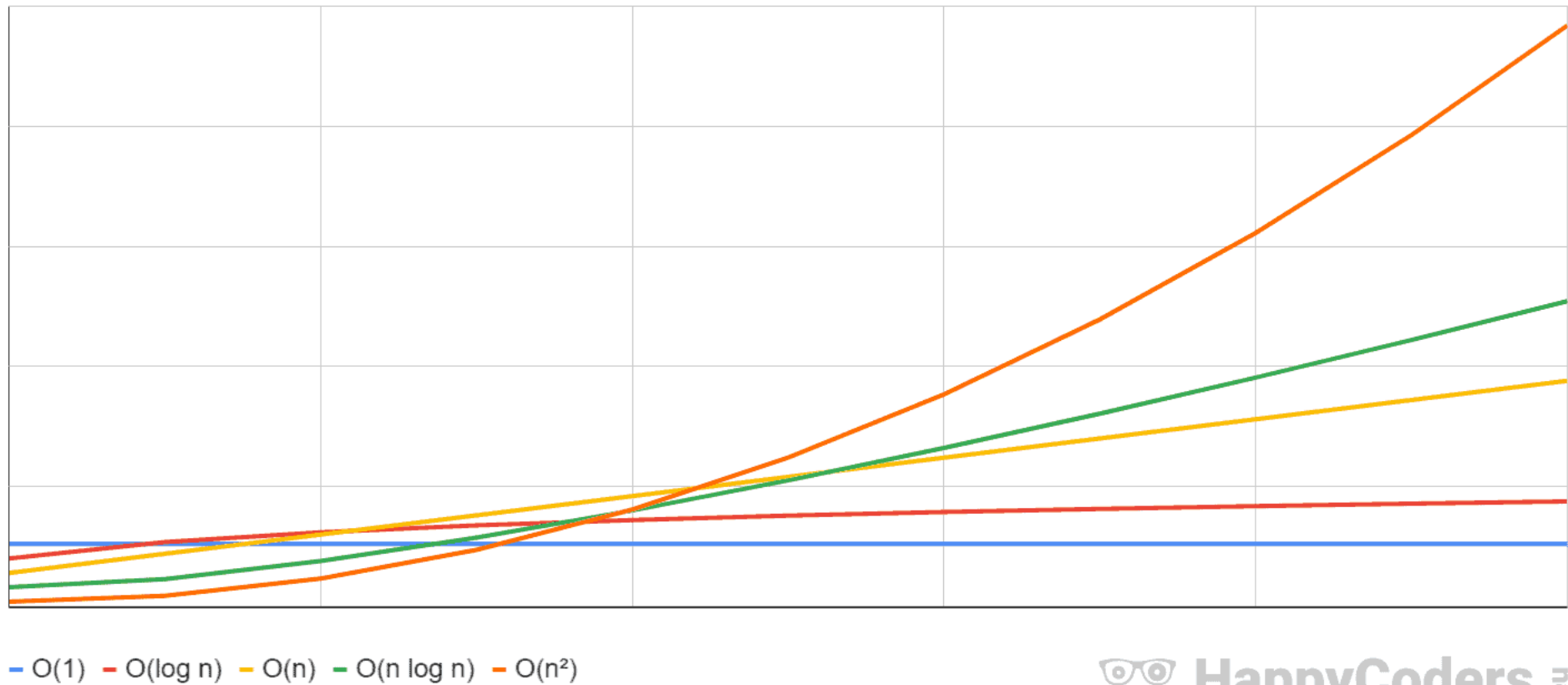
Conclusion #3: Think about your data structure!

- 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)):
            h = haystack[hi + ni].lower()
            n = needle[ni].lower()
            if h != n:
                match=False
                break
        if match:
            results.append(hi)
    return results
```

```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.

```
: 1  import cProfile
  2  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}
```
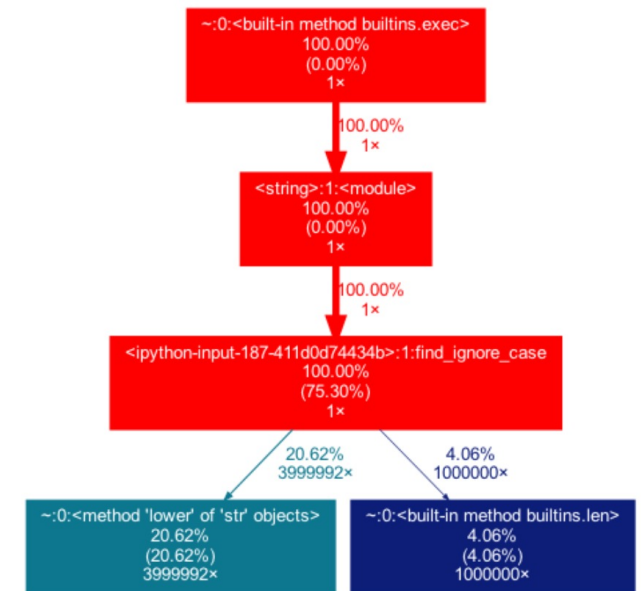
Find: The needle in the haystack.
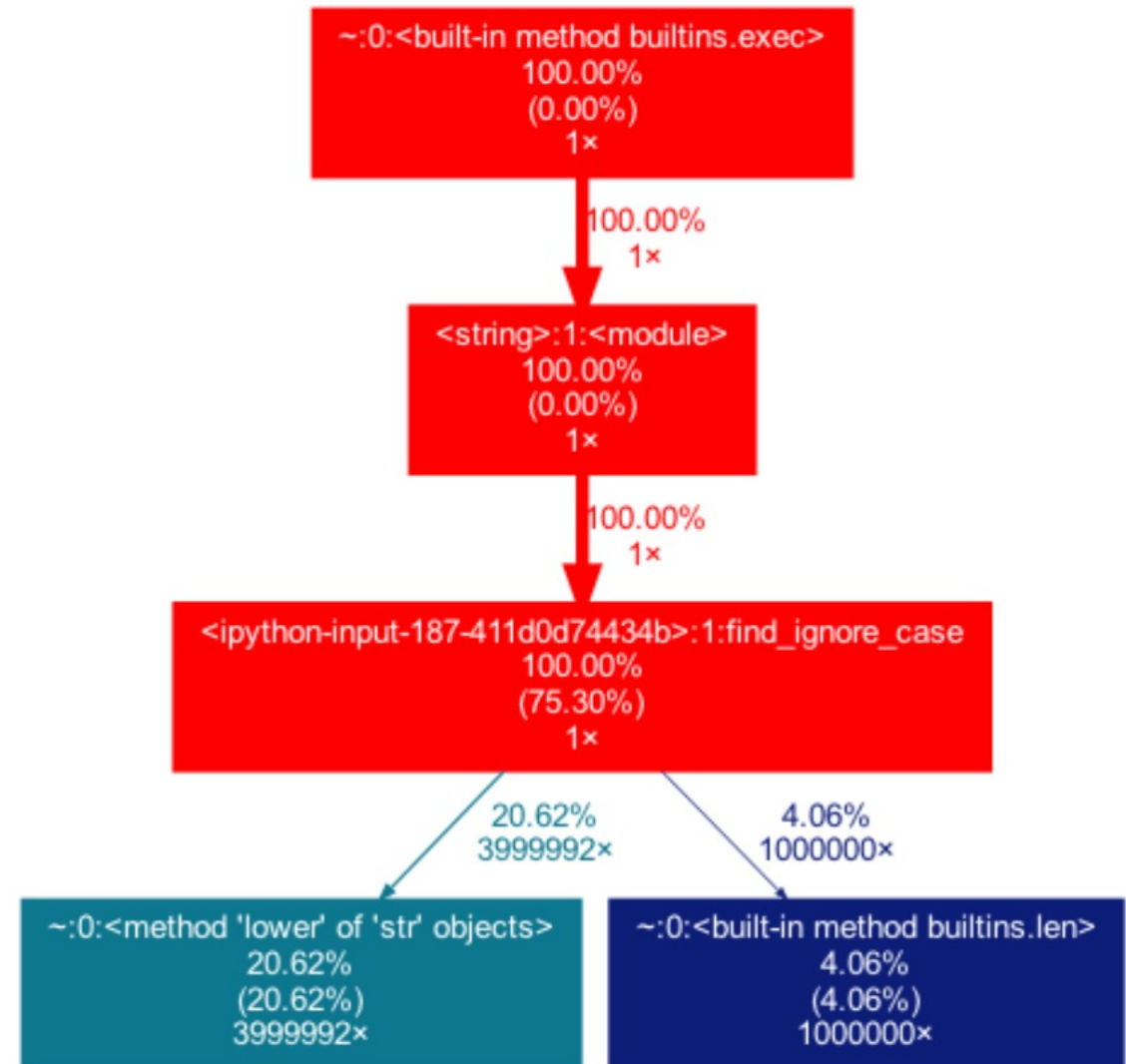
Find: The needle in the haystack.


```
1   def find_ignore_case( needle, haystack):
2       results = []
3       for hi in range(len(haystack)-len(needle)):
4           match = True
5           for ni in range(len(needle)):
6               h = haystack[hi + ni].lower()
7               n = needle[ni].lower()
8               if h != n:
9                   match=False
10          if match:
11              results.append(hi)
12      return results
13
14  random.seed(1)
15  sz=1000000
16  haystack = random_str(sz)
17  needle = haystack[int(sz/2):int(sz/2)+2]
18
19  start_time = time.time()
20  results = find_ignore_case(needle, haystack)
21  end_time = time.time()
22  print("True List: %f seconds" % (end_time - start_time))
```

True List: 1.109001 seconds

- No libraries!

37

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
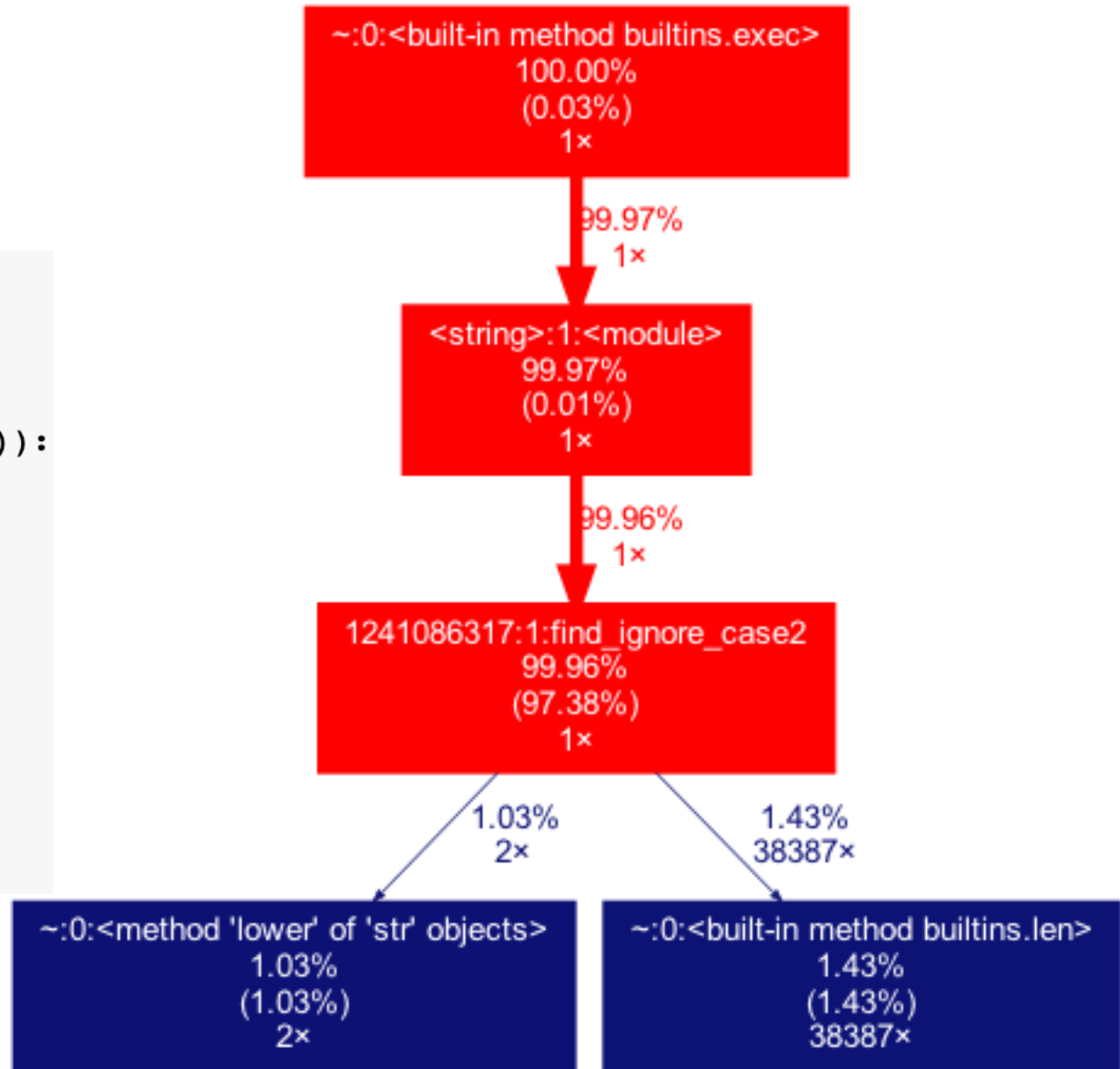Find2: 0.440155 seconds

Anything else?

```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
```


~:0:<built-in method builtins.exec>
100.00%
(0.03%)
1×

99.97%
1×

<string>:1:<module>
99.97%
(0.01%)
1×

99.96%
1×

1241086317:1:find_ignore_case2
99.96%
(97.38%)
1×

1.03%
2×

1.43%
38387×

~:0:<method 'lower' of 'str' objects>
1.03%
(1.03%)
2×

~:0:<built-in method builtins.len>
1.43%
(1.43%)
38387×

40

```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]

42

```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

43

Using built-in libraries is usually the fastest…

```python
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
```

Q: How is this so much faster?

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

Q: How is this so much faster?

```python
def find_ignore_case5( needle, haystack):
    return [haystack.find(needle)]
```
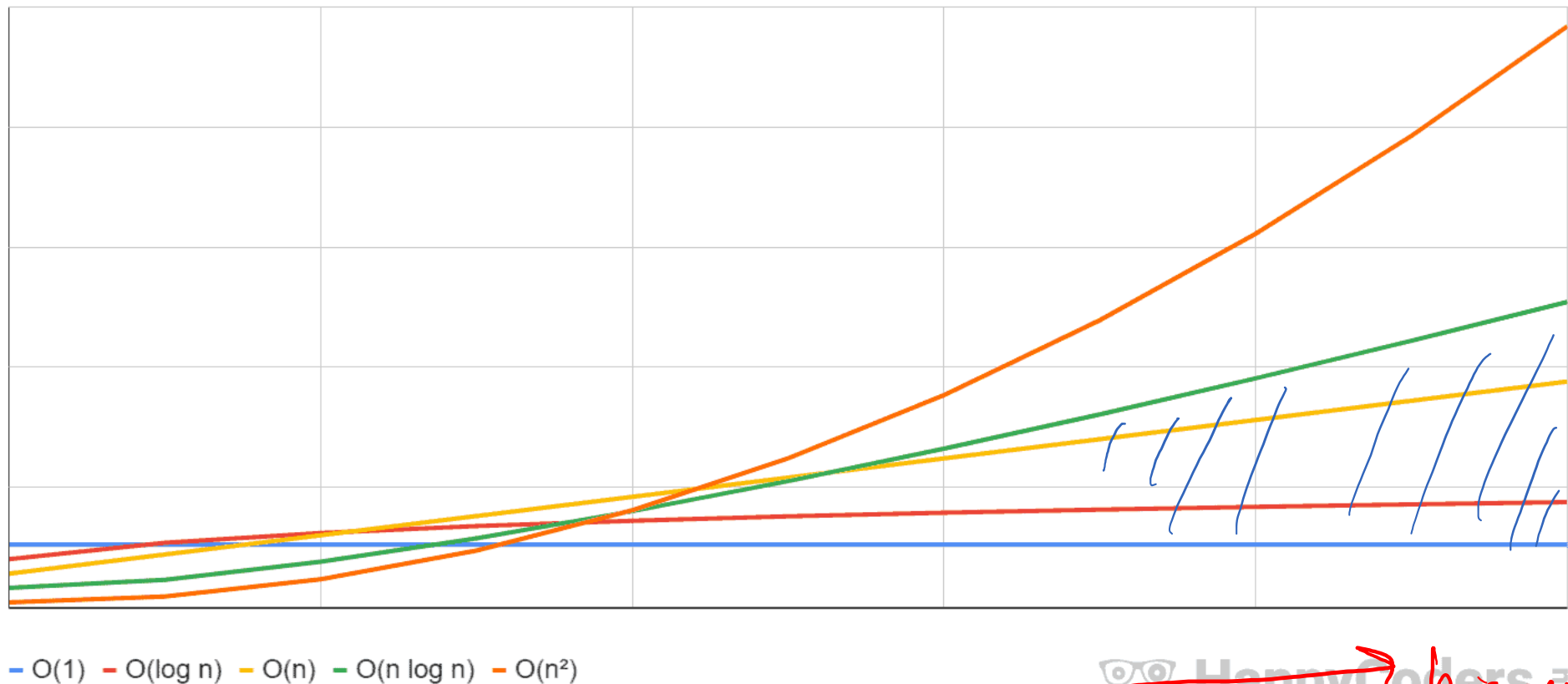
A: It built into Python. So it runs at C speeds!

Last Time: Big O Complexity

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

O() Complexities

Comparing the complexity classes O(1), O(log n), O(n), O(n log n), O(n²)

Legend: O(1) — O(log n) — O(n) — O(n log n) — O(n²)

Handwritten annotations: bigger runtimes ↑, bigger inputs →, $O(n^2)$, $O(n \log n)$, $O(n)$, $O(1)$

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

Last Time: 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_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
```

~ 8x faster

$\dfrac{0.37}{0.05}$

49

Using built-in libraries is usually faster...

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

```
Find: 0.270210 seconds
Find2: 0.061821 seconds
Find3: 0.054265 seconds
Find4: 0.051191 seconds
Find5: 0.000265 seconds
```

Numpy is written in C/Assembly. It's faster.

```python
import numpy
def find_ignore_case6(needle, haystack):
    return np.where(haystack==needle)
```

```
Find:  0.270210 seconds
Find2: 0.061821 seconds
Find3: 0.054265 seconds
Find4: 0.051191 seconds
Find5: 0.000265 seconds
Find6: 0.000052 seconds
```

Conclusions

1. Optimize algorithm if possible
2. Function calls are not free!
3. Preallocation (Bulk Allocation) is *usually* faster
4. Think about your data structure!
5. Use optimized libraries if possible

Popcount

- Count the number of binary 1's in a number

- 01000**1**00**1**0**1**0000**1**00**1**0000**1**00000000

- 7 total 1's

Popcount

```
def popcount(num):
    return bin(num).count('1')
```

value: 0 bin: 0b0 popcount: 0
value: 1 bin: 0b1 popcount: 1
value: 2 bin: 0b10 popcount: 1
value: 3 bin: 0b11 popcount: 2
value: 4 bin: 0b100 popcount: 1
value: 5 bin: 0b101 popcount: 2
value: 6 bin: 0b110 popcount: 2
value: 7 bin: 0b111 popcount: 3
value: 8 bin: 0b1000 popcount: 1
value: 9 bin: 0b1001 popcount: 2

5 → "0101"

$$X - Y = X + (-Y) = X + (\sim Y + 1)$$

popcount (slower, but no external calls)

```
def popcount2(num):
    w = 0
    while (num):
        w += 1
        num &= num - 1
    return w
```

$5 \rightarrow \text{``}0101\text{''}$

$4 \rightarrow 0100$

$3 \rightarrow 0011$

$0101 \qquad W = 1$

$\& \; 0100$

$= 0100 \qquad W = 2$

$\& \; 0011$

$0000 \Rightarrow$

$W = 2$

popcount (slower, but no external calls)

```
def popcount2(num):
    w = 0
    while (num):
        w += 1
        num &= num - 1
    return w

def popcount2_all(buf):
    return sum(map(bitcount2,buf))
```

1011 = 11

1010 (-1)   +1
1010

1001   +1
1000

0111   + +1
0000

3

`Popcount_all` is a helper function to run larger blocks of inputs

```
def popcount_all(buf):
    return sum(map(bitcount,buf))
def popcount2_all(buf):
    return sum(map(popcount2,buf))
```

popcount [handwritten annotation pointing to bitcount]

Big Bitcount

```python
np.random.seed(1)
buf = np.random.randint(0,1E9,int(1E6))

start_time = time.time()
sum_1s = popcount_all(buf)
end_time = time.time()
print("popcount: %f seconds (w/libs)"
        % (end_time - start_time))


start_time = time.time()
sum_1s = popcount2_all(buf)
end_time = time.time()
print("popcount2: %f seconds (w/o libs)"
        % (end_time - start_time))
```

```
popcount: 0.307169 seconds (w/libs)
popcount2: 1.853192 seconds (w/o libs)
```

How did the library go so much faster?

- Python called C.

- The computations happened in C. It's faster.

- Can we do that?

Let's find out.

Popcount in Python vs. C

Python

```python
def popcount2(num):

    w = 0
    while (num):
        w += 1
        num &= num - 1

    return w
```

C

```c
int popcount(uint64_t num)
{
    int w=0;
    while (num) {
        w +=1;
        num &= (num -1);
    }
    return w;
}
```

Popcount test?

```c
#include <stdio.h>
#include "popcount.h"

int main()
{
    int res;
    for (int i = 0; i < 20; ++i){
        res = popcount(i);
        printf ("i:%d i:0x%x res: %d\n", i, i, res);
    }
    return 0;
}
```

```
i:0 i:0x0 res: 0
i:1 i:0x1 res: 1
i:2 i:0x2 res: 1
i:3 i:0x3 res: 2
i:4 i:0x4 res: 1
i:5 i:0x5 res: 2
i:6 i:0x6 res: 2
i:7 i:0x7 res: 3
i:8 i:0x8 res: 1
i:9 i:0x9 res: 2
i:10 i:0xa res: 2
i:11 i:0xb res: 3
i:12 i:0xc res: 2
i:13 i:0xd res: 3
i:14 i:0xe res: 3
i:15 i:0xf res: 4
i:16 i:0x10 res: 1
i:17 i:0x11 res: 2
i:18 i:0x12 res: 2
i:19 i:0x13 res: 3
```

Let's see if we can wrap C `popcount` with Python

- https://realpython.com/build-python-c-extension-module/#packaging-your-python-c-extension-module

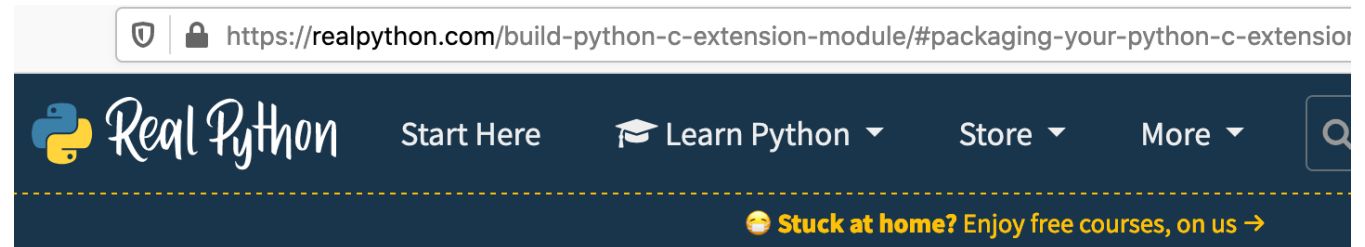- https://docs.python.org/3/extending/extending.html

Step 1: RTFM

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

Step 2: RTFM 2

https://realpython.com/build-python-c-extension-module/#packaging-your-python-c-extension

A minimal `setup.py` file for your module should look like this:

```python
from distutils.core import setup, Extension

def main():
    setup(name="fputs",
          version="1.0.0",
          description="Python interface for the fputs C library function",
          author="<your name>",
          author_email="your_email@gmail.com",
          ext_modules=[Extension("fputs", ["fputsmodule.c"])])

if __name__ == "__main__":
    main()
```

```c
20 static PyObject *
21 cPopcount(PyObject *self, PyObject *args)
22 {
23     uint64_t num;
24
25     if (!PyArg_ParseTuple(args, "l", &num))
26         return NULL;
27
28     //popcount!!!
29     uint64_t res = popcount(num);
30
31     return PyLong_FromLong(res);
32 }
```

```c
int popcount(uint64_t num)
{
    int w=0;
    while (num) {
        w +=1;
        num &= (num -1);
    }
    return w;
}
```

```python
import cPopcount
cPopcount.cPopcount(0xffff)
16
```

FIXME

```python
np.random.seed(1)
buf = np.random.randint(0,1E9,int(1E6))
buf = buf.tolist()

start_time = time.time()
sum_1s = popcount_all(buf)
end_time = time.time()
print("popcount: %f seconds (w/calls)"
       % (end_time - start_time))

start_time = time.time()
sum_1s = popcount2_all(buf)
end_time = time.time()
print("popcount2: %f seconds (w/o calls)"
       % (end_time - start_time))

start_time = time.time()
sum_1s = sum(map(cPopcount.cPopcount,buf))
end_time = time.time()
print("c_popcount: %f seconds (64-bits in C)"
       % (end_time - start_time))
```

popcount: 0.261108 seconds (w/calls)
popcount2: 0.881429 seconds (w/o calls)
c_popcount: 0.027510 seconds (64-bits in C)

```c
static PyObject *
cPopcount_all(PyObject *self, PyObject *args)
{
    PyObject *obj;
    int64_t res = 0;

    //parse the list argument
    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }

    //hope it's iteratable
    PyObject *iter = PyObject_GetIter(obj);
    if (!iter) {
        return NULL;// error not iterator
    }

    //loop over all elements in list
        ) {
        PyObject  next    PyIt  _Next(    );

        i   !ne    {
            //   hing le    the it  ato
            brea
        }
```

```c
        // conver to int64_t
        int64_t num = 0;
        if (PyLong_Check(next)) {
            num = PyLong_AsLong(next);
        } else {
            printf ("unsupported type\n");
            return NULL;
        }

        //now do popcount!
        res += popcount(num);// do something with foo

        /* release reference when done */
        Py_DECREF(next);
    }

    y_DECREF(iter);

    eturn PyLong_FromLong(res);
}
```

FIXME

67

```python
start_time = time.time()
sum_1s = sum(map(cPopcount.cPopcount,buf))
end_time = time.time()
print("c_popcount: %f seconds (64-bits in C)"
        % (end_time - start_time))


start_time = time.time()
sum_1s = cPopcount.cPopcount_all(buf)
end_time = time.time()
print("c_popcount: %f seconds (List in C)"
        % (end_time - start_time))
```

```
popcount: 0.261108 seconds (w/calls)
popcount2: 0.881429 seconds (w/o calls)
c_popcount: 0.027510 seconds (64-bits in C)
c_popcount: 0.007329 seconds (List in C)
```

Same algorithm. C vs. Python.

```
popcount: 0.261108 seconds (w/calls)
popcount2: 0.881429 seconds (w/o calls)
c_popcount: 0.027510 seconds (64-bits in C)
c_popcount: 0.007329 seconds (List in C)
```

When performance matters, use C.
When it doesn't, use Python.

02: C Interfaces

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University