BIG OH and THETA

(download slides and .py files to follow along)

6.100L Lecture 22

Ana Bell

TIMING

TIMING A PROGRAM

- Use time module
- Importing means bringing collection of functions into your own file
- **Start** clock
- **Call** function
- **Stop** clock

```
import time

def convert_to_km(m):
    return m * 1.609
t0 = time.perf_counter()
convert_to_km(100000)
dt = time.perf_counter() - t0
print("t =", dt, "s,")
```

More accurate timer, meaningful when used to get a time diff

EXAMPLE: `convert_to_km`, `compound`

```
def convert_to_km(m):
    return m * 1.609

def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
        total = total * interest + invest
    return total
```

- How long does it take to compute these functions?
- Does the time depend on the input parameters?
- Are the times noticeably different for these two functions?

CREATING AN INPUT LIST

Create a set of input sizes, each of which is 10 times larger than the previous one [1, 10, 100, 1000, …]

```
L_N = [1]
for i in range(7):
    L_N.append(L_N[-1]*10)

for N in L_N:
    t = time.perf_counter()
    km = convert_to_km(N)
    dt = time.perf_counter()-t
    print(f"convert_to_km({N}) took {dt} seconds ({1/dt}/sec)")
```

Measure time to compute (aka run function) for each input

Report time and how many times the fcn can run per sec

RUN IT!
`convert_to_km` OBSERVATIONS

Scientific notation, i.e.

$$1.44e{-}06 = 1.44 \times 10^{-6}$$

```
convert_to_km(1) took 4.30e-06 sec (232,558.14/sec)
convert_to_km(10) took 7.00e-07 sec (1,428,571.43/sec)
convert_to_km(100) took 4.00e-07 sec (2,499,999.99/sec)
convert_to_km(1000) took 3.00e-07 sec (3,333,333.33/sec)
convert_to_km(10000) took 3.00e-07 sec (3,333,333.33/sec)
convert_to_km(100000) took 4.00e-07 sec (2,499,999.99/sec)
convert_to_km(1000000) took 4.00e-07 sec (2,499,999.99/sec)
convert_to_km(10000000) took 3.00e-07 sec (3,333,333.33/sec)
convert_to_km(100000000) took 3.00e-07 sec (3,333,333.33/sec)
```

Observation: average time seems independent of size of argument

MEASURE TIME:
`compound` with a variable number of months

```
def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
        total = total * interest + invest
    return total
```

```
compound(1) took 2.26e-06 seconds (441,696.12/sec)
compound(10) took 2.31e-06 seconds (433,839.48/sec)
compound(100) took 6.59e-06 seconds (151,676.02/sec)
compound(1000) took 5.02e-05 seconds (19,938.59/sec)
compound(10000) took 5.10e-04 seconds (1,961.80/sec)
compound(100000) took 5.14e-03 seconds (194.46/sec)
compound(1000000) took 4.79e-02 seconds (20.86/sec)
compound(10000000) took 4.46e-01 seconds (2.24/sec)
```

Observation 1: Time grows with the input only when n_months changes

Observation 2: average time seems to increase by 10 as size of argument increases by 10

Observation 3: relationship between size and time only predictable for large sizes

MEASURE TIME: sum over L

```
def sum_of(L):
    total = 0.0
    for elt in L:
        total = total + elt
    return total

L_N = [1]
for i in range(7):
    L_N.append(L_N[-1]*10)

for N in L_N:
    L = list(range(N))
    t = time.perf_counter()
    s = sum_of(L)
    dt = time.perf_counter()-t
    print(f"sum_of({N}) took {dt} seconds ({1/dt}/sec)")
```

[0,1,2,…9] then [0,1,2,…99] etc

Observation 1: Size of the input is now the length of the list, not how big the element numbers are.

Observation 2: average time seems to increase by 10 as size of argument increases by 10

Observation 3: relationship between size and time only predictable for large sizes

Observation 4: Time seems comparable to computation of compound

MEASURE TIME: find element in a list

```
# search each element one-by-one
def is_in(L, x):
    for elt in L:
        if elt==x:
            return True
    return False

# search by bisecting the list (list should be sorted!)
def binary_search(L, x):
    lo = 0
    hi = len(L)
    while hi-lo > 1:
        mid = (hi+lo) // 2
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
    return L[lo] == x

# search using built-in operator
x in L
```

Integer division, round down

Measure "average" time. Search for the first, middle, and last element of sorted list, and average these 3 times.

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

Observation 3: binary search time seems *almost* independent of size

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

Observation 3: binary search time seems *almost* independent of size

Observation 4: binary search much faster than is_in, especially on larger problems

MEASURE TIME: find element in a list

```
is_in(10000000) took 1.62e-01 seconds (6.16/sec)
    9.57 times more than for 10 times fewer elements
binary(10000000) took 9.37e-06 seconds (106,761.64/sec)
    1.40 times more than for 10 times fewer elements
builtin(10000000) took 5.64e-02 seconds (17.72/sec)
    9.63 times more than for 10 times fewer elements

is_in(100000000) took 1.64e+00 seconds (0.61/sec)
    10.12 times more than for 10 times fewer elements
binary(100000000) took 1.18e-05 seconds (84,507.09/sec)
    1.26 times more than for 10 times fewer elements
builtin(100000000) took 5.70e-01 seconds (1.75/sec)
    10.11 times more than for 10 times fewer elements
```

Observation 1: searching one-by-one grows by factor of 10, when L increases by 10

Observation 2: built-in function grows by factor of 10, when L increases by 10

Observation 3: binary search time seems *almost* independent of size

Observation 4: binary search much faster than is_in, especially on larger problems

Observation 5: is_in is slightly slower than using Python's "in" capability

MEASURE TIME: find element in a list

```
def is_in(L, x):
    for elt in L:
        if elt==x:
            return True
    return False


def binary_search(L, x):
    lo = 0
    hi = len(L)
    while hi-lo > 1:
        mid = (hi+lo) // 2
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
    return L[lo] == x
```

So we have seen computations where time seems very different

- Constant time
- Linear in size of argument
- Something less than linear?

MEASURE TIME: diameter function

```
L=[(cos(0), sin(0)),
   (cos(1), sin(1)),
   (cos(2), sin(2)), ... ] #example numbers
```

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

`L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]`

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

```
L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]
```

MEASURE TIME: diameter function

```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

1st iter: len(L) - 1 passes
2nd iter: len(L) - 2 pases …
On average, len(L) / 2 passes

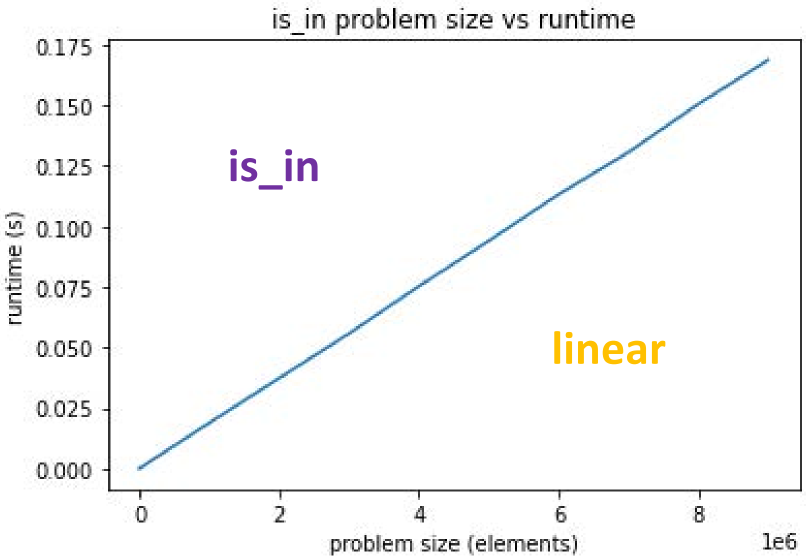
`L = [(cos(0),sin(0)), (cos(1),sin(1)), (cos(2),sin(2)), (cos(3),sin(3))]`

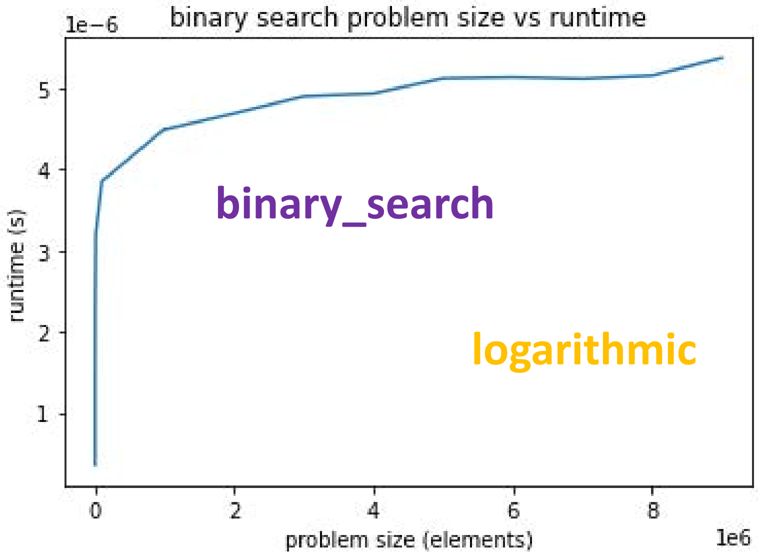
MEASURE TIME: diameter function

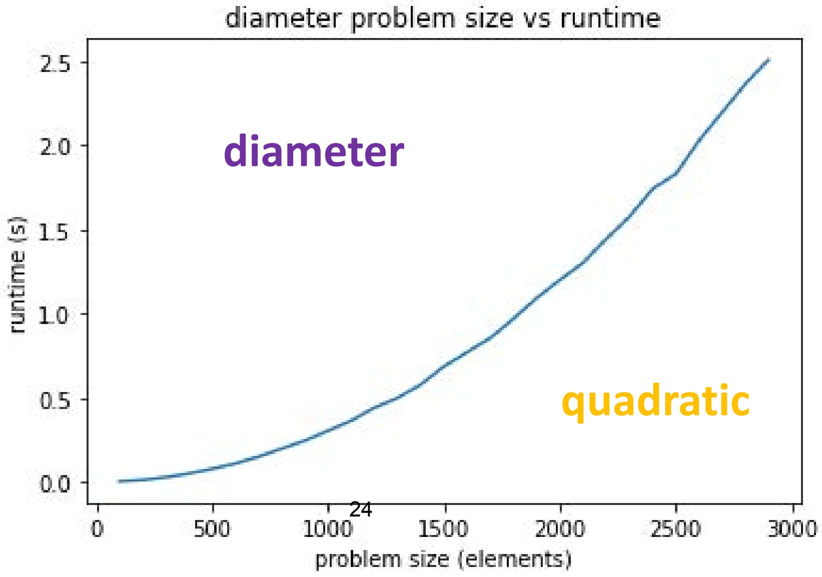
```
def diameter(L):
    farthest_dist = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

- Gets much slower as size of input grows
- *Quadratic: for list of size len(L), does len(L)/2 operations per element on average*
- *len(L) x len(L)/2 operations — worse than linear growth*

PLOT OF INPUT SIZE vs. TIME TO RUN

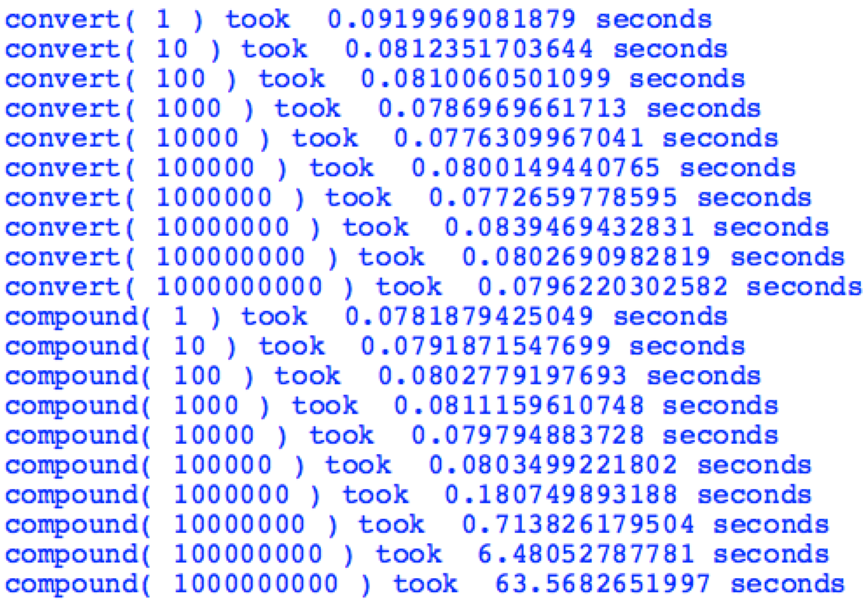

is_in problem size vs runtime


binary search problem size vs runtime


diameter problem size vs runtime

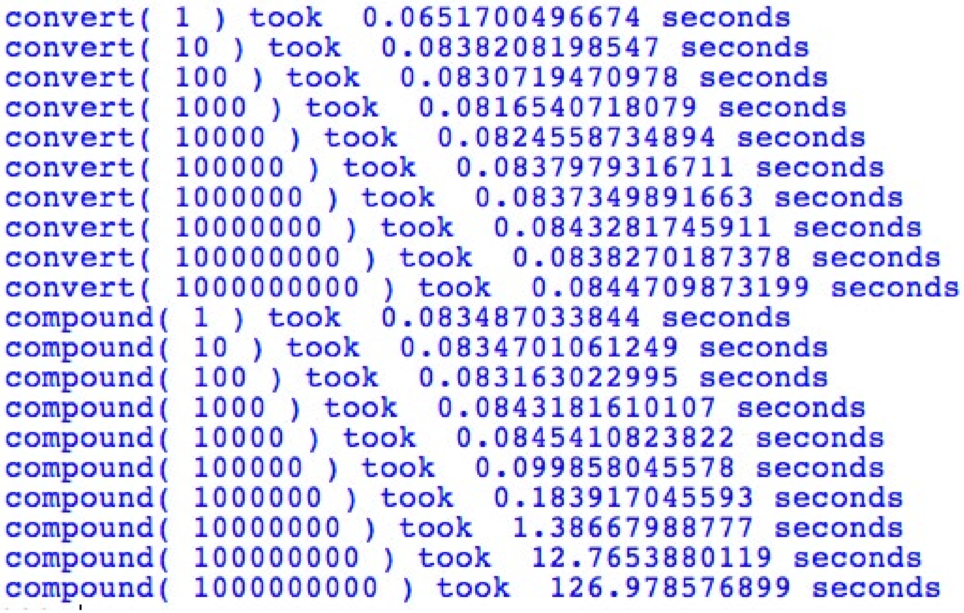
TWO DIFFERENT MACHINES

My old laptop



```
convert( 1 ) took  0.0919969081879 seconds
convert( 10 ) took  0.0812351703644 seconds
convert( 100 ) took  0.0810060501099 seconds
convert( 1000 ) took  0.0786969661713 seconds
convert( 10000 ) took  0.0776309967041 seconds
convert( 100000 ) took  0.0800149440765 seconds
convert( 1000000 ) took  0.0772659778595 seconds
convert( 10000000 ) took  0.0839469432831 seconds
convert( 100000000 ) took  0.0802690982819 seconds
convert( 1000000000 ) took  0.0796220302582 seconds
compound( 1 ) took  0.0781879425049 seconds
compound( 10 ) took  0.0791871547699 seconds
compound( 100 ) took  0.0802779197693 seconds
compound( 1000 ) took  0.0811159610748 seconds
compound( 10000 ) took  0.079794883728 seconds
compound( 100000 ) took  0.0803499221802 seconds
compound( 1000000 ) took  0.180749893188 seconds
compound( 10000000 ) took  0.713826179504 seconds
compound( 100000000 ) took  6.48052787781 seconds
compound( 1000000000 ) took  63.5682651997 seconds
```

My old desktop



```
convert( 1 ) took  0.0651700496674 seconds
convert( 10 ) took  0.0838208198547 seconds
convert( 100 ) took  0.0830719470978 seconds
convert( 1000 ) took  0.0816540718079 seconds
convert( 10000 ) took  0.0824558734894 seconds
convert( 100000 ) took  0.0837979316711 seconds
convert( 1000000 ) took  0.0837349891663 seconds
convert( 10000000 ) took  0.0843281745911 seconds
convert( 100000000 ) took  0.0838270187378 seconds
convert( 1000000000 ) took  0.0844709873199 seconds
compound( 1 ) took  0.083487033844 seconds
compound( 10 ) took  0.0834701061249 seconds
compound( 100 ) took  0.083163022995 seconds
compound( 1000 ) took  0.0843181610107 seconds
compound( 10000 ) took  0.0845410823822 seconds
compound( 100000 ) took  0.099858045578 seconds
compound( 1000000 ) took  0.183917045593 seconds
compound( 10000000 ) took  1.38667988777 seconds
compound( 100000000 ) took  12.7653880119 seconds
compound( 1000000000 ) took  126.978576899 seconds
```

~2x slower for large problems

Observation 1: even for the same code, the actual machine may affect speed.

Observation 2: Looking only at the relative increase in run time from a prev run, if input is n times as big, the run time is approx. n times as long.

DON’T GET ME WRONG!

- Timing is a **critical tool to assess the performance** of programs
    - At the end of the day, it is irreplaceable for real-world assessment

- But we will see a complementary tool (**asymptotic complexity**) that has other advantages
    - A priori evaluation (before writing or running code)
    - **Assesses algorithm** independent of machine and implementation (what is intrinsic efficiency of algorithm?)
    - Provides direct insight into the **design** of efficient algorithms

COUNTING

COUNT OPERATIONS

- Assume these steps take **constant time**:
  - Mathematical operations
  - Comparisons
  - Assignments
  - Accessing objects in memory
- Count number of these operations executed as function of size of input

convert_to_km → 2 ops

```
def convert_to_km(m):
    return m * 1.609
```

2 ops

sum_of → 1+len(L)*3+1 = 3*len(L)+2 ops

```
def sum_of(L):
    total = 0
    for i in L:
        total += i
    return total
```

1 op

1 op

loop len(L) times

2 ops

1 op

COUNT OPERATIONS: is_in

```
def is_in_counter(L, x):

    for elt in L:

        if elt==x:
            return True
    return False
```

COUNT OPERATIONS: is_in

```
def is_in_counter(L, x):
    global count
    count += 1
    for elt in L:
        count += 2
        if elt==x:
            return True
    return False
```

Return value

Set elt as val from L,
Check elt==x

Global lets us reference and change an external variable inside a function – OK for debugging / timing but not good practice in real programs

COUNT OPERATIONS: binary search

```
def binary_search_counter(L, x):
    global count
    lo = 0
    hi = len(L)
    count += 3
    while hi-lo > 1:
        count += 2
        mid = (hi+lo) // 2
        count += 3
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
        count += 3
    count += 3
    return L[lo] == x
```

Set lo, hi, len

While test and the subtraction

Addition, //, and assign mid

Access mid, if test and assign mid

Access lo, == test, return

COUNT OPERATIONS

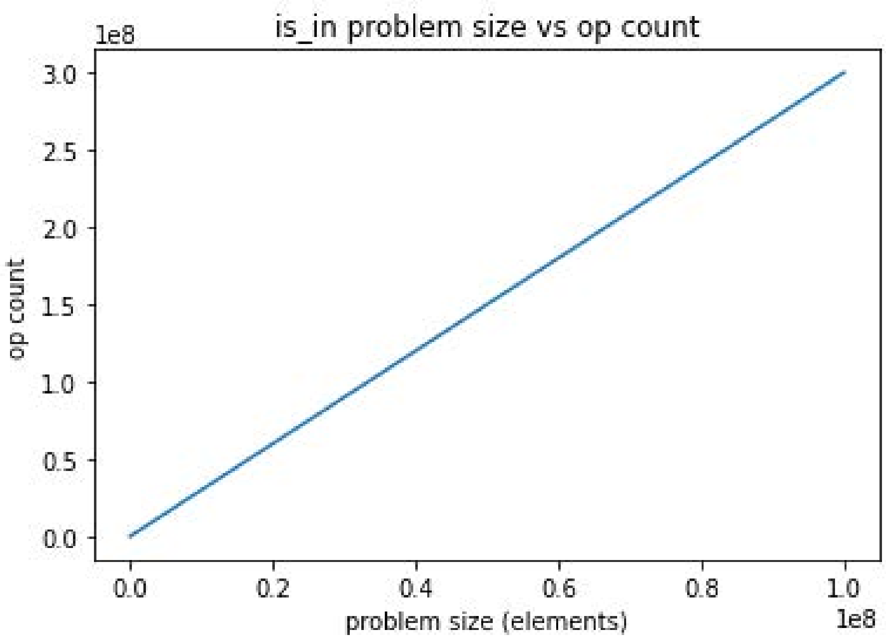
```
is_in testing
for  1 element, is_in used 9 operations
for  10 element, is_in used 37 operations
for  100 element, is_in used 307 operations
for  1000 element, is_in used 3007 operations
for  10000 element, is_in used 30007 operations
for  100000 element, is_in used 300007 operations
for  1000000 element, is_in used 3000007 operations
```

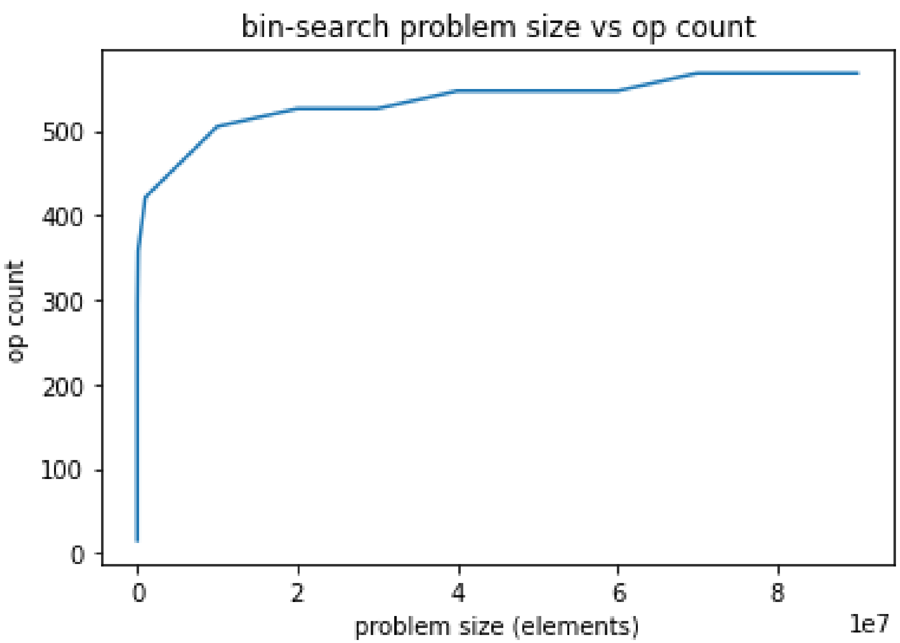
Observation 1: number of operations for is_in increases by 10 as size increases by 10

```
binary_search testing
for  1 element, binary search used 15 operations
for  10 element, binary search used 85 operations
for  100 element, binary search used 148 operations
for  1000 element, binary search used 211 operations
for  10000 element, binary search used 295 operations
for  100000 element, binary search used 358 operations
for  1000000 element, binary search used 421 operations
```

Observation 2: *but* number of operations for binary search grows *much more slowly.* Unclear at what rate.

PLOT OF INPUT SIZE vs. OPERATION COUNT





PROBLEMS WITH TIMING AND COUNTING

- **Timing** the exact running time of the program
  - Depends on **machine**
  - Depends on **implementation**
  - **Small inputs** don’t show growth
- **Counting** the exact number of steps
  - Gets us a **formula!**
  - **Machine independent**, which is good
  - Depends on **implementation**
  - **Multiplicative/additive constants** are irrelevant for large inputs
- Want to:
  - evaluate **algorithm**
  - evaluate **scalability**
  - evaluate **in terms of input size**

EFFICIENCY IN TERMS OF INPUT: BIG-PICTURE
RECALL `mysum` (one loop) and `square` (nested loops)

- `mysum(x)`
  - What happened to the **program efficiency as x increased**?
  - 10 times bigger x meant the program
    - Took approx. 10 times as long to run
    - Did approx. 10 times as many ops
  - Express it in an “order of” way vs. the input variable: **efficiency = Order of x**
- `square(x)`
  - What happened to the **program efficiency as x increased**?
  - 2 times bigger x meant the program
    - Took approx. 4 times as long to run
    - Did approx. 4 times as many ops
  - 10 times bigger x meant the program
    - Took approx. 100 times as long to run
    - Did approx. 100 times as many ops
  - Express it in an “order of” way vs. the input variable: **efficiency = Order of x^2**

ORDER of GROWTH

ORDERS OF GROWTH

- It’s a notation
- Evaluates programs when **input is very big**
- Expresses the **growth of program’s run time**
- Puts an **upper bound** on growth
- Do not need to be precise: **“order of” not “exact”** growth

- Focus on the **largest factors** in run time (which section of the program will take the longest to run?)

A BETTER WAY
A GENERALIZED WAY WITH APPROXIMATIONS

- Use the idea of counting operations in an algorithm, but **not worry about small variations in implementation**
  - When x is big, 3x+4 and 3x and x are pretty much the same!
  - Don’t care about exact value: ops = 1+x(2+1)
  - Express it in an **“order of” way vs. the input**: ops = Order of x
- Focus on how algorithm performs when **size of problem gets arbitrarily large**
- **Relate time** needed to complete a computation **against the size of the input** to the problem
- Need to decide what to measure. What is the input?

WHICH INPUT TO USE TO MEASURE EFFICIENCY

- Want to express efficiency in terms of input, so need to **decide what is your input**

- Could be an **integer**
  -- `convert_to_km(x)`
- Could be **length of list**
  -- `list_sum(L)`
- **You decide** when multiple parameters to a function
  -- `is_in(L, e)`
  - Might be different depending on which input you consider

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- A function that searches for an element in a list

```
def is_in(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- Does the program take longer to run **as e increases**?
  - No

```
is_in([1,2,3], 0)  vs.
is_in([1,2,3], 1000)
```

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- A function that searches for an element in a list

```
def is_in(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

```
is_in([1,2,3], 0) vs.
is_in([1000,2000,3000], 0)
```

- Does the program take longer to run as `L` increases?
  - What if L has a fixed length and **its elements are big numbers**?
    - No
  - What if L has **different lengths**?
    - Yes!

```
is_in([1,2,3], 0) vs.
is_in([1,2,3,4,5,6,7,8,9,10], 0)
```

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- A function that searches for an element in a list

```
def is_in(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- When `e` is **first element** in the list
  → BEST CASE
- When **look through about half** of the elements in list
  → AVERAGE CASE
- When `e` is **not in list**
  → WORST CASE
  - Want to measure this behavior in a general way

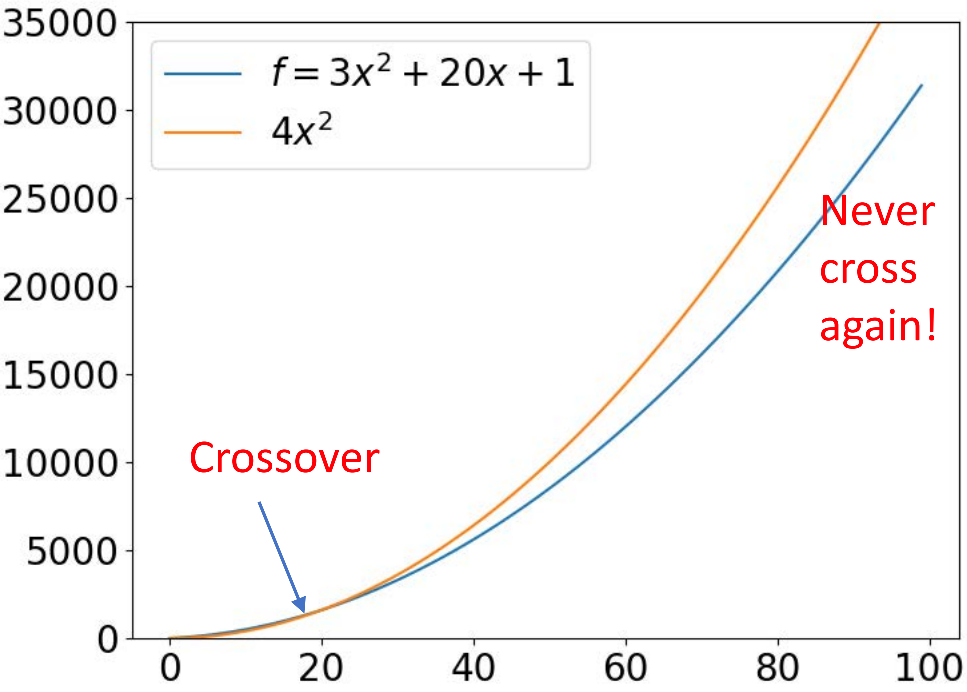
ASYMPTOTIC GROWTH

- Goal: describe how time grows as size of input grows
  - Formula relating input to number of operations
- Given an expression for the number of operations needed to compute an algorithm, want to know **asymptotic behavior as size of problem gets large**
  - Want to put a **bound** on growth
  - Do not need to be precise: **"order of" not "exact"** growth
- Will focus on term that grows most rapidly
  - Ignore additive and multiplicative constants, since want to know how rapidly time required increases as we increase size of input
- This is called ***order of growth***
  - Use mathematical notions of **"big O"** and **"big Θ"**

Big Oh and **Big Theta**

BIG O Definition

$\boxed{3x^2 + 20x + 1 = O(x^2)}$



$\boxed{4x^2} > 3x^2 + 20x + 1 \forall x > \boxed{20.04}$

- Suppose some code runs in $f(x) = 3x^2 + 20x + 1$ steps
  - Think of this as the formula from counting the number of ops.
- Big OH is a way to upper bound the growth of *any* function
- f(x) = O(g(x)) means that g(x) times *some* constant *eventually* always exceeds f(x)
  - *Eventually* means above some threshold value of x

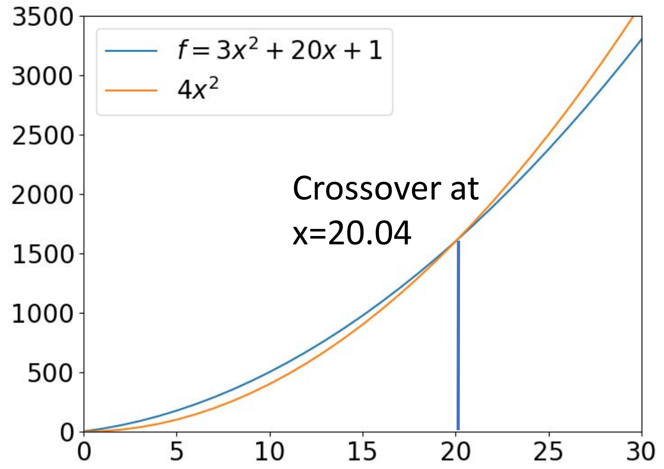
BIG O FORMALLY

- A big Oh bound is an **upper bound** on the growth of some function
- $\boxed{f(x) = O(\mathbf{g(x)})}$ means there exist constants $\mathbf{c_0}, \mathbf{x_0}$ for which $\boxed{\mathbf{c_0}}\boxed{\mathbf{g(x)}} \geq \boxed{\mathbf{f(x)}}$ for all $x > \boxed{\mathbf{x_0}}$

Example: $f(x) = 3x^2 + 20x + 1$

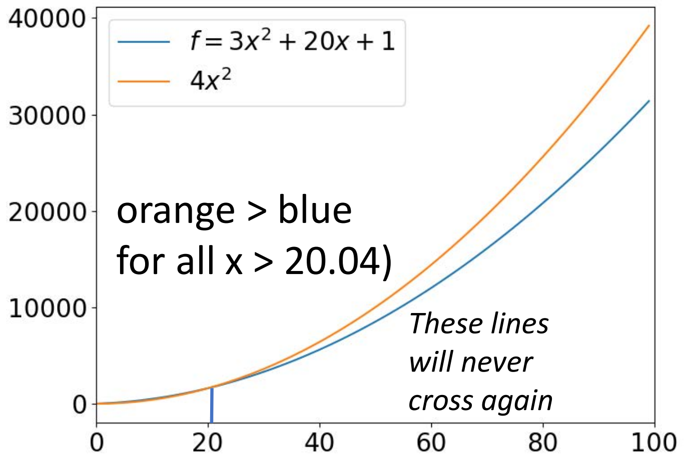
$\boxed{f(x) = O(\mathbf{x^2})}$,because $\boxed{\mathbf{4}}\boxed{\mathbf{x^2}} > \boxed{\mathbf{3x^2 + 20x + 1}} \forall x \geq \boxed{\mathbf{21}}$

$(\mathbf{c_0 = 4}, \mathbf{x_0 = 20.04})$



Crossover at x=20.04

0 <= x <= 30



orange > blue for all x > 20.04)

These lines will never cross again

0 <= x <= 100

BIG Θ Definition

$$\boxed{3x^2 - 20x - 1 = \theta(x^2)}$$

- A **big Θ** bound is **a lower and upper bound** on the growth of some function

Suppose $f(x) = 3x^2 - 20x - 1$

$f(x) = \Theta(g(x))$ means:

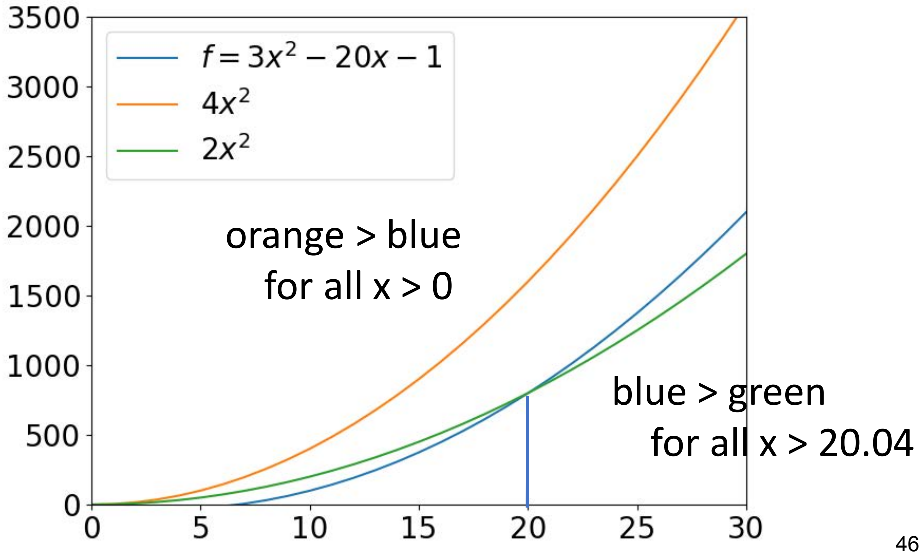
there exist constants c_0, x_0 for which $\boxed{c_0 g(x) \geq f(x) \text{ for all } x > x_0}$

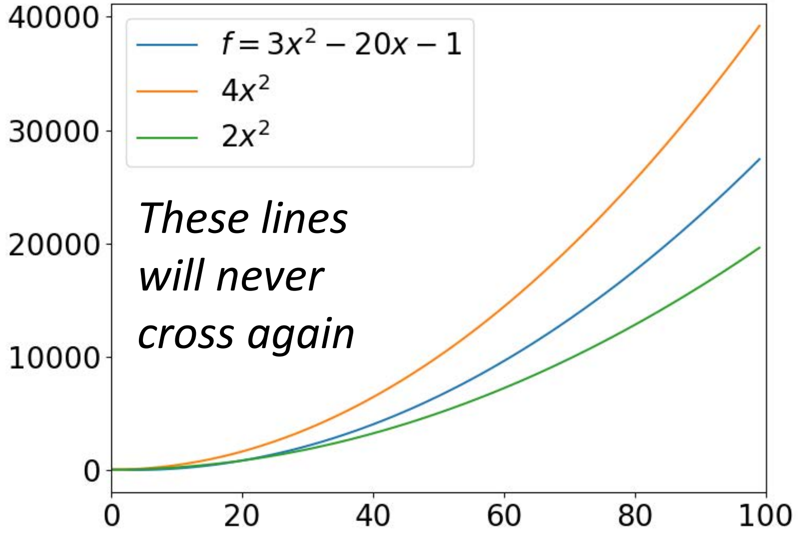
and constants c_1, x_1 for which $c_1 g(x) \leq f(x)$ for all $x > x_1$

Big O definition

- Example, $f(x) = \Theta(x^2)$ because $\boxed{4x^2 > 3x^2 - 20x - 1 \ \forall x \geq 0 \quad (c_0 = 4, x_0 = 0)}$

and $2x^2 < 3x^2 - 20x - 1 \ \forall x \geq 21 \quad (c_1 = 2, x_1 = 20.04)$





0 <= x <= 100

Θ vs O

- In practice, Θ bounds are preferred, because they are "tight"
  For example: $f(x) = 3x^2 - 20x - 1$

- $f(x) = O(x^2) = O(x^3) = O(2^x)$ and anything higher order because they all upper bound it

- $\boldsymbol{f(x) = \Theta(x^2)}$
  $\neq \Theta(x^3) \neq \Theta(2^x)$ and anything higher order because they upper bound but not lower bound it

SIMPLIFICATION EXAMPLES

- Drop constants and multiplicative factors
- Focus on **dominant term**

$\Theta(n^2)$: $\boxed{n^2} + 2n + 2$

$\Theta(x^2)$: $\boxed{3x^2} + 100000x + 3^{1000}$

$\Theta(a)$: $\log(a) + \boxed{a} + 4$

BIG IDEA

Express Theta in terms of the input.

Don’t just use n all the time!

YOU TRY IT!

$\Theta(x)$: `1000*log(x) + x`

$\Theta(n^3)$: $n^2\log(n) + n^3$

$\Theta(y)$: `log(y) + 0.000001y`

$\Theta(2^b)$: $2^b + 1000a^2 + 100*b^2 + 0.0001a^3$

$\Theta(a^3)$

$\Theta(2^b+a^3)$

All could be ok, depends on the input we care about

USING Θ TO EVALUATE YOUR ALGORITHM

```
def fact_iter(n):
    """assumes n an int >= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

5 steps inside loop

1. compare,
2. multiply,
3. assign,
4. subtract,
5. assign

- Number of steps: 5n + 2
- Worst case asymptotic complexity: Θ(n)
  - Ignore additive constants
    - 2 doesn’t matter when n is big
  - Ignore multiplicative constants
    - 5 doesn’t matter if just want to know how increasing n changes time needed

COMBINING COMPLEXITY CLASSES
LOOPS IN SERIES

- Analyze statements inside functions to get order of growth
- Apply some rules, focus on dominant term

- **Law of Addition** for Θ():
  - Used with **sequential** statements
  - $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$
- For example,

```
for i in range(n):          Θ(n)
    print('a')
for j in range(n*n):        Θ(n²)
    print('b')
```

is $\Theta(n) + \Theta(n * n) = \Theta(n + n^2) = \Theta(n^2)$ because of dominant n^2 term

COMBINING COMPLEXITY CLASSES NESTED LOOPS

- Analyze statements inside functions to get order of growth
- Apply some rules, focus on dominant term

- **Law of Multiplication for Θ():**
  - Used with **nested** statements/loops
  - $\Theta(f(n)) * \Theta(g(n)) = \Theta(f(n) * g(n))$
- For example,

Θ(n)

```
for i in range(n):
    for j in range(n//2):
        print('a')
```

Θ(n) for each outer loop iteration

- $\Theta(n) \times \Theta(n) = \Theta(n \times n) = \Theta(n^2)$
  - Outer loop runs n times and the inner loop runs n times for every outer loop iteration.

ANALYZE COMPLEXITY

- What is the Theta complexity of this program?

Always note the input parameter!

```
def f(x):
    answer = 1
    for i in range(x):
        for j in range(i,x):
            answer += 2
    return answer
```

Outer loop is $\Theta(x)$

Inner loop is $\Theta(x)$

Everything else is $\Theta(1)$

- $\boxed{\Theta(1)} + \boxed{\Theta(x) * \Theta(x) * \Theta(1)} + \boxed{\Theta(1)}$
- Overall complexity is **$\Theta(x^2)$** by rules of addition and multiplication

YOU TRY IT!

- What is the Theta complexity of this program? Careful to describe in terms of input
  (hint: what matters with a list, size of elems of length?)

```
def f(L):
    Lnew = []
    for i in L:
        Lnew.append(i**2)
    return Lnew
```

ANSWER:
Loop: Θ(len(L))
f is Θ(len(L))

YOU TRY IT!

- What is the Theta complexity of this program?

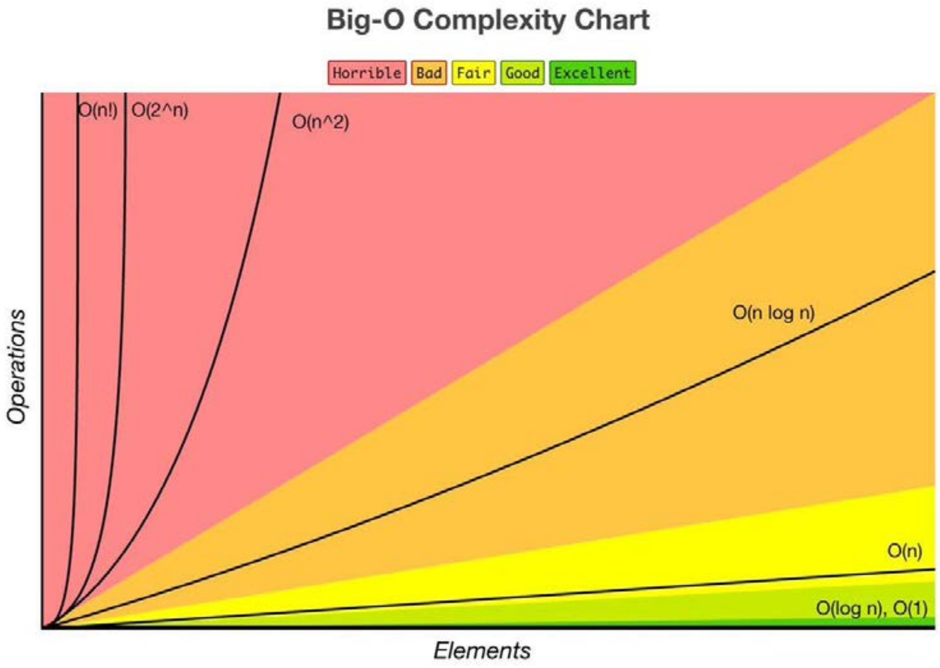
```
def f(L, L1, L2):
    """ L, L1, L2 are the same length """
    inL1 = False
    for i in range(len(L)):
        if L[i] == L1[i]:
            inL1 = True
    inL2 = False
    for i in range(len(L)):
        if L[i] == L2[i]:
            inL2 = True
   return inL1 and inL2
```

ANSWER:

Loop: Θ(len(L)) + Θ(len(L))

f is Θ(len(L)) or Θ(len(L1)) or Θ(len(L2))

COMPLEXITY CLASSES



We want to design algorithms that are as close to top of this hierarchy as possible

- $\Theta(1)$ denotes **constant** running time
- $\Theta(\log n)$ denotes **logarithmic** running time
- $\Theta(n)$ denotes **linear** running time
- $\Theta(n \log n)$ denotes **log-linear** running time
- $\Theta(n^c)$ denotes **polynomial** running time
  (c is a constant)
- $\Theta(c^n)$ denotes **exponential** running time
  (c is a constant raised to a power based on input size)

COMPLEXITY GROWTH

CLASS	N = 10	N = 100	N = 1000	N = 1000000
Constant	1	1	1	1
Logarithmic	1	2	3	6
Linear	10	100	1000	1000000
Log-linear	10	200	3000	6000000
Polynomial	100	10000	1000000	1000000000000
Exponential	1024	1267650600228229401496703205376	10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376	Good Luck!!

SUMMARY

- Timing is machine/implementation/algorithm dependent
- Counting ops is implementation/algorithm dependent
- Order of growth is algorithm dependent

- Compare **efficiency of algorithms**
  - Notation that describes growth
  - **Lower order of growth** is better
  - Independent of machine or specific implementation
- Using Theta
  - Describe asymptotic order of growth
  - **Asymptotic notation**
  - **Upper bound** and a **lower bound**

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022