Complexity of Python Operations

In this lecture we will learn the complexity classes of various operations on Python data types. Then we wil learn how to combine these complexity classes to compute the complexity class of all the code in a function, and therefore the complexity class of the function. This is called "static" analysis, because we do not need to run any code to perform it (contrasted with Dynamic or Emperical Analysis, when we do run code and take measurements of its execution).

Python Complexity Classes

In ICS-46 we will write low-level implementations of all of Python's data types and see/understand WHY these complexity classes apply. For now we just need to try to absorb (not memorize) this information, with some -but minimal- justification.

Binding a value to any name (copying a refernce) is O(1). Simple operators on integers (whose values are small: e.g., under 12 digits) like + or == are also O(1). You should assume small integers in problems unless explicitly told otherwise.

In all these examples, N = len(data-structure). The operations are organized by increasing complexity class

Lists:

Operation	Example	Complexity Class	Notes
Index	l[i]	O(1)	
Store	l[i] = 0	O(1)	
Length	len(l)	O(1)	

Append	l.append(5)	O(1)	mostly: ICS-46 covers details
Pop	l.pop()	O(1)	same as l.pop(-1), popping at end
Clear	l.clear()	O(1)	similar to l = []
Slice	l[a:b]	O(b-a)	l[1:5]:O(l)/l[:]:O(len(l)-0)=O(N)
Extend	l.extend(...)	O(len(...))	depends only on len of extension
Construction	list(...)	O(len(...))	depends on length of ... iterable
check ==, !=	l1 == l2	O(N)	
Insert	l[a:b] = ...	O(N)	
Delete	del l[i]	O(N)	depends on i; O(N) in worst case
Containment	x in/not in l	O(N)	linearly searches list
Copy	l.copy()	O(N)	Same as l[:] which is O(N)
Remove	l.remove(...)	O(N)	
Pop	l.pop(i)	O(N)	O(N-i): l.pop(0):O(N) (see above)
Extreme value	min(l)/max(l)	O(N)	linearly searches list for value
Reverse	l.reverse()	O(N)	
Iteration	for v in l:	O(N)	Worst: no return/break in loop
Sort	l.sort()	O(N Log N)	key/reverse mostly doesn't change
Multiply	k*l	O(k N)	5*l is O(N): len(l)*l is O(N**2)

Tuples support all operations that do not mutate the data structure (and they have the same complexity classes).

Sets:

Operation	Example	Complexity Class	Notes
Length	len(s)	O(1)	
Add	s.add(5)	O(1)	
Containment	x in/not in s	O(1)	compare to list/tuple - O(N)
Remove	s.remove(..)	O(1)	compare to list/tuple - O(N)
Discard	s.discard(..)	O(1)	

Pop	s.pop()	O(1)	popped value "randomly" selected
Clear	s.clear()	O(1)	similar to s = set()
Construction	set(...)	O(len(...))	depends on length of ... iterable
check ==, !=	s != t	O(len(s))	same as len(t); False in O(1) if the lengths are different
<=/<	s <= t	O(len(s))	issubset
>=/>	s >= t	O(len(t))	issuperset s <= t == t >= s
Union	s | t	O(len(s)+len(t))	
Intersection	s & t	O(len(s)+len(t))	
Difference	s - t	O(len(s)+len(t))	
Symmetric Diff	s ^ t	O(len(s)+len(t))	
Iteration	for v in s:	O(N)	Worst: no return/break in loop
Copy	s.copy()	O(N)	

Sets have many more operations that are O(1) compared with lists and tuples. Not needing to keep values in a specific order in a set (while lists/tuples require an order) allows for faster implementations of some set operations.

Frozen sets support all operations that do not mutate the data structure (and they have the same complexity classes).

Dictionaries: dict and defaultdict

Operation	Example	Complexity Class	Notes
Index	d[k]	O(1)	
Store	d[k] = v	O(1)	
Length	len(d)	O(1)	
Delete	del d[k]	O(1)	
get/setdefault	d.get(k)	O(1)	
Pop	d.pop(k)	O(1)	

Pop item	d.popitem()	O(1)	popped item "randomly" selected
Clear	d.clear()	O(1)	similar to s = {} or = dict()
View	d.keys()	O(1)	same for d.values()
Construction	dict(...)	O(len(...))	depends # (key,value) 2-tuples
Iteration	for k in d:	O(N)	all forms: keys, values, items
			Worst: no return/break in loop

So, most dict operations are O(1).

defaultdicts support all operations that dicts support, with the same complexity classes (because it inherits all those operations); this assumes that calling the constructor when a values isn't found in the defaultdict is O(1) - which is true for int(), list(), set(), ... (the things we commonly use)

Note that for i in range(...) is O(len(...)); so for i in range(1,10) is O(1). If len(alist) is N, then

```
  for i in range(len(alist)):
```

is O(N) because it loops N times. Of course even

```
  for i in range (len(alist)//2):
```

is O(N) because it loops N/2 times, and dropping the constant 1/2 makes it O(N): the work doubles when the list length doubles. By this reasoning,

```
  for i in range (len(alist)//1000000):
```

is O(N) because it loops N/1000000 times, and dropping the constant 1000000 makes it O(N): the work doubles when the list length doubles. Remember, we are interested in what happens as N -> Infinity.

Finally, when comparing two lists for equality, the complexity class above shows as O(N), but in reality we would need to multiply this complexity class by

O==(...) where O==(...) is the complexity class for checking whether two values in the list are ==. If they are ints, O==(...) would be O(1); if they are strings, O==(...) in the worst case it would be O(len(string)). This issue applies any time an == check is done. We mostly will assume == checking on values in lists is O(1): e.g., checking ints and small/fixed-length strings.

Composing Complexity Classes: Sequential and Nested Statements

In this section we will learn how to combine complexity class information about simple operations into complexity class information about complex operations (composed from simple operations). The goal is to be able to analyze all the statements in a functon/method to determine the complexity class of executing the function/method. As with computing complexity classes themselves, these rules are simple and easy to apply once you understand how to use them.

Law of Addition for big-O notation

```
 O(f(n)) + O(g(n)) is O( f(n) + g(n) )
```

That is, we when adding complexity classes we bring the two complexity classes inside the O(...). Ultimately, O(f(n) + g(n)) results in the bigger of the two complexity class (because we alwasy drop the lower-complexity added term). So,

```
O(N) + O(Log N)  =  O(N + Log N)  =  O(N)
```

because N is the faster growing term: lim (N->infinity) Log N/N = 0.

This rule helps us understand how to compute the complexity class of doing any SEQUENCE of operations: executing a statement that is O(f(n)) followed by executing a statement that is O(g(n)). Executing both statements SEQUENTIALLY is O(f(n)) + O(g(n)) which is O(f(n) + g(n)) by the rule above.

For example, if some function call f(...) is O(N) and another function call g(...) is O(N Log N), then doing the sequence

```
   f(...)
   g(...)
```

is O(N) + O(N Log N) = O(N + N Log N) = O(N Log N). Of course, executing the sequence (calling f twice)

```
  f(...)
  f(...)
```

is O(N) + O(N) which is O(N + N) which is O(2N) which is O(N) because we discard multiplicative constants in big-O notation.

Note that an if statment sequentially evaluates test AND THEN one of the blocks.

```
  if test:      assume complexity class of computing test is O(T)
     block 1     assume complexity class of executing block 1 is O(B1)
  else:
     block 2     assume complexity class of executing block 2 is O(B2)
```

The complexity class for the if is O(T) + max(O(B1),O(B2)). The test is always evaluated, and one of the blocks is always executed afterward (so, a sequence of evaulating a test followed by executing a block). In the worst case, the if will execute the block with the largest complexity class. So, given

```
  if test:      complexity class is O(N)
     block 1     complexity class is O(N**2)
  else:
     block 2     complexity class is O(N)
```

The complexity class for the if is O(N) + max (O(N**2),O(N))) = O(N) + O(N**2)
= O(N + N**2) = O(N**2).

If the test had complexity class O(N**3), then the complexity class for the if
is O(N**3) + max (O(N**2),O(N))) = O(N**3) + O(N**2) = O(N**3 + N**2) = O(N**3).

In fact, the complexity class for an if can also be written as
O(T) + O(B1) + O(B2): for the if above example O(N) + O(N**2) + O(N) = O(N**2).
Why? Because we always throw away the lower-order terms, whic is like taking
the max of the terms. I prefer writing O(T) + max(O(B1),O(B2)) because it looks
like what is happening: the test is always evaluated, and one of the blocks.

Law of Multiplcation for big-O notation

 O(f(n)) * O(g(n)) is O(f(n) * g(n))

If we repeat an O(f(N)) process O(N) times, the resulting complexity class is
O(N)*O(f(N)) = O(N*f(N)). An example of this is, if some function call f(...)
is O(N**2), then executing that call N times (in the following loop)

```
  for i in range(N):
    f(...)
```

is O(N)*O(N**2) = O(N*N**2) = O(N**3)

This rule helps us understand how to compute the complexity class of doing some
statement INSIDE A BLOCK controlled by a statement that is REPEATING it. We
multiply the complexity class of the number of repetitions by the complexity
class of the statement (sequence; using the summing rule) being repeated.

Compound statements can be analyzed by composing the complexity classes of
their constituent statements. For sequential statements (including if tests and

their block bodies) the complexity classes are added; for statements repeated
in a loop the complexity classes are multiplied.

One Function Specification/3 Implementations and their Analysis

Let's use the data and tools discussed above to analyze (determine the
complexity classes) of three different functions that each compute the same
result: whether or not a list contains only unique values (no duplicates). We
will assume in all three examples that len(alist) is N and that we can compare
the list elements in O(1): e.g., they are small ints or strs.

1) Algorithm 1: A list is unique if each value in the list does not occur in any
later indexes: alist[i+1:] is a list slice containing all values after the one
at index i.

```
def is_unique1 (alist : [int]) -> bool:
    for i in range(len(alist)):         O(N) - for every index; see * below
        if alist[i] in alist[i+1:]:     O(N) - index+add+slice+in: O(1)+O(1)+O(N)+O(N) = O(N)
            return False      O(1) - never executed in worst case; ignore
    return True                    O(1) - always executed in worst case; use
```

*Note that creating a range object requires 3 sequential operations: computing
the arguments, passing the arguments to __init__, and executing the body of
__init__. The latter two are both O(1), and computing len(alist) is also O(1),
so the complexity of range(len(alist)) is O(1)+O(1)+O(1) = O(1).

The complexity class for executing the entire function is O(N) * O(N) + O(1)
= O(N**2). So we know from the previous lecture that if we double the length of
alist, this function takes 4 times as long to execute.

Many students want to write this as O(N) * (O(N) + O(1)) + O(1) because the

if statement's complexity is O(N) + O(1): complexity of test + complexity of block when test is True (there is no block when test is False). But in the WORST CASE, the return is NEVER EXECUTED (the loop keeps executing) so it should not appear in the formula. Although, even if it appears in this formula, the formula still computes the same complexity class (because O(N) + O(1) is still O(N)): O(N**2).

So, in the worst case, we never return False and keep executing the loop, so this O(1) does not appear in the formula. Also, in the worst case the list slice is aliset[1:] which is O(N-1) = O(N), although when i is len(alist) the slice contains 0 values: is empty. The average list slice taken in the if has N/2 values, which is still O(N).

We can also write this function purely using loops (no slicing), but the complexity class is the same.

```
def is_unique1 (alist : [int]) -> bool:
    for i in range(len(alist)):         O(N) - for every index
        for j in range(i+1,len(alist)): O(N) - N-i indexes; O(N) in worst case
            if alist[i] == a[j]:   O(1) - index+index+==: O(1)+O(1)+O(1) = O(1)
                return False       O(1) - never executed in worst case; ignore
    return True                    O(1) - always executed in worst case; use
```

The complexity class for executing the entire function is O(N)*O(N)*O(1) + O(1) = O(N**2). So we know from the previous lecture that if we double the length of alist, this function takes 4 times as long to execute.

------
2) Algorithm 2: A list is unique if when we sort its values, no ADJACENT values are equal. If there were duplicate values, sorting the list would put these duplicate values right next to each other (adjacent). Here we copy the list so as to not mutate (change the order of) the parameter's list by sorting it (functions generally shouldn't mutate their arguments unless that is the purpose

of the function): it turns out that copying the list does not increase the complexity class of the method, because the O(N) used for copying is not the largest added term computing the complexity class of this function's body.

```
def is_unique2 (alist : [int]) -> bool:
    copy = list(alist)             O(N)
    copy.sort()                    O(N Log N) - for fast Python sorting
    for i in range(len(alist)-1):  O(N) - really N-1, but that is O(N); len and - are both O(1)
        if copy[i] == copy[i+1]:   O(1): +, 2 [i], and  == on ints: all O(1)
            return False      O(1) - never executed in worst case
    return True                    O(1) - always executed in worst case
```

The complexity class for executing the entire function is given by the sum O(N) + O(N Log N) + O(N)*O(1) + O(1) = O(N + N Log N + O(N*1) + 1) = O(N + N Log N + N + 1) = O(N Log N + 2N + 1) = O(N Log N). So the complexity class for this algorithm/function is lower than the first algorithm, the is_unique1 function. For large N unique2 will eventually run faster. Because we don't know the constants, we don't know which is faster for small N.

Notice that the complexity class for sorting is dominant in this code: it does most of the work. If we double the length of alist, this function takes a bit more than twice the amount of time. In N Log N: N doubles and Log N gets a tiny bit bigger (i.e., Log 2N = 1 + Log N; e.g., Log 2000 = 1 + Log 1000 = 11, so compared to 1000 Log 1000, doubling N is 2000 Log 2000, which is just 2.2 times bigger, or 10% bigger than just doubling).

Looked at another way if T(N) = c*(N Log N), then T(2N) = c*(2N Log 2N) = c*2N(Log N + 1) = c*2N Log N + c*2N = 2*T(N) + c*2N. Or, computing the doubling signature

```
  T(2N)     c*2N Log N + c*2N     c*2N Log N       c*2N              2
-------- = ------------------- = ------------ + ----------- = 2 + -------
  T(N)         c N Log N            c N Log N     c N Log N         Log N
```

So, the ratio is 2 + a bit (and that bit gets smaller -very slowly- as N increases): for N >= 10**3 it is <= 2.2; for N >= 10**6 it is <= 2.1; for N >= 10**9 it it < 2.07. So, it is a bit worse than doubling each time, but much better than O(N**2) which is quadrupling each time.

In fact, we could also simplify

```
    copy = list(alist)             O(N)
    copy.sort()                    O(N Log N) - for fast Python sorting
```

to just

```
    copy = sorted(alist)                O(N Log N) - for fast Python sorting
```

because sorted will create a list of all the values in its iterable argument, and return it after mutating (sorting) it. So we don't have to explicitly create such a copy in our code.

This change will speed up the code, but it won't change the complexity analysis because O(N + N Log N) = O (N Log N). Speeding up code is always good, but finding an algorithm in a better complexity class (as we did going from is_unique1 to is_unique2) is much btter

Finally, is_unique2 works only if all the values in the list are comparable (using the < relational operator needed for sorting): it would fail if the list contained both integers and strings. Whereas, using either version of is_unique1, requires only comparing values with ==: 3 == 'xyz' is False; it does not raise an exception.

------
3) Algorithm 3: A list is unique if when we turn it into a set, its length is unchanged: if duplicate values were added to the set, its length would be smaller than the length of the list by exactly the number of duplicates in the list added to the set.

```
def is_unique3 (alist : [int]) -> bool:
    aset = set(alist)              O(N): construct set from alist values
    return len(aset) == len(alist) O(1): 2 len (each O(1)) and == ints O(1)
```

The complexity class for executing the entire function is O(N) + O(1) = O(N + 1) = O(N). So the complexity class for this algortihm/function is lower than both the first and second algorithms/functions. If we double the length of alist, this function takes just twice the amount of time. We could write the body of this function more simply as: return len(set(alist)) == len(alist), where evaluating set(alist) takes O(N) and then computing the two len's and comparing them for equality are all O(1): O(N)+O(1)+O(1)+O(1) = O(N).

Unlike is_unique2, it can work for lists containing both integers and strings. But, is_unique3 works only if all the values in the list are hashable/immutable (a requirement for storing values in a set). So, it would not work for a list of lists.

So the bottom line here is that there might be many algorithms/functions to solve some problem. If the function bodies are small, we can analyze them statically (looking at the code, not needing to run it) to determine their complexity classes. For large problem sizes, the algorithm/function with the smallest complexity class will ultimately be best, running in the least amount of time.

But, for small problem sizes, complexity classes don't determine which is best: for small problem we need to take into account the CONSTANTS and lower order terms that we ignored when computing complexity classes). We can run the functions (dynamic analysis, aka empirical analysis) to test which is fastest on small problem sizes.

And finally, sometimes we must put additional constraints on data passed to some implementations: is_unique2 require that its list store values comparable by < (for sorting it); is_unique3 requires that its list store values that hashable/immutable.

Using a Class (implementable 3 ways) Example:

We will now look at the solution of a few problems (combining operations on a priority queue: pq) and how the complexity class of the result is affected by three different classes/implementations of priority queues.

In a priority queue, we can add values to and remove values from the data structure. A correctly working priority queue always removes the maximum value remaining in the priority queue (the one with the highest priority). Think of a line/queue outside of a Hollywood nightclub, such that whenever space opens up inside, the most famous person in line gets to go in (the "highest priority" person), no matter how long less famous people have been standing in line (contrast this with first come/first serve, which is a regular -non priority- queue; in a regaular queue, whoever is first in the line -has been standing in line longest- is admitted next).

For the problems below, all we need to know is the complexity class of the "add" and "remove" operations.

	add	remove
Implementation 1	O(1)	O(N)
Implementation 2	O(N)	O(1)
Implementation 3	O(Log N)	O(Log N)

Implementation 1 adds the new value into the pq by appending the value at the rear of a list or the front of a linked list: both are O(1); it removes the highest priority value by scanning through the list or linked list to find the highest value, which is O(N), and then removing that value, also O(N) in the worst case (removing at the front of a list; at the rear of a linked list).

Implementation 2 adds the new value into the pq by scanning the list or linked list for the right spot to put it and putting it there, which is O(N). Lists store their highest priority at the rear (linked lists at the front); it removes the highest priority value from the rear for lists (or the front for linked lists), which is O(1).

So Implementations 1 and 2 swap the complexity classes in their add/remove method. Implementation 1 doesn't keep the values in order: so easy to add but hard to find/remove the maximum (must scan). Implementation 2 keeps the values in order: so hard to add (need to scan to find where it goes) but easy to find/remove the maximum (at one end; the fast one).

Implementation 3, which is discussed in ICS-46, uses a binary heap tree (not a binary search tree) to implement both operations with "middle" complexity O(Log N): this complexity class greater than O(1) but less than O(N). Because Log N grows so slowly, O(Log N) is actually closer to O(1) than O(N) even though O(1) doesn't grow at all: Log N grows that slowly.

Problem 1: Suppose we wanted to use the priority queue to sort N values: we add N values in the pq and then remove all N values (first the highest, next the second highest, ...). Here is the complexity of these combined operations for each implementation.

```
Implementation 1: N*O(1) + N*O(N)         = O(N)   + O(N**2)    = O(N**2)
Implementation 2: N*O(N) + N*O(1)         = O(N**2) + O(N)      = O(N**2)
Implementation 3: N*O(Log N) + N*O(Log N) = O(NLogN) + O(NLogN) = O(NLogN)
```

Note N*O(...) is the same as O(N)*O(...) which is the same as O(N * ...)

Here, Implementation 3 has the lowest complexity class for the combined operations. Implementations 1 and 2 each do one operation quickly but the other slowly: both are done O(N) times. The slowest operation determines the complexity class, and both are equally slow. The complexity class O(Log N) is between O(1) and O(N); surprisingly, it is actually "closer" to O(1) than O(N), even though it does grow -because it grows so slowly; yes, O(1) doesn't grow at all, but O(Log N) grows very slowly: the known Universe has about 10**90 particles of matter, and Log 10**90 = Log (10**3)**30 = 300, which isn't very big compared to 10**90 (like 86 orders of magnitude less).

Problem 2: Suppose we wanted to use the priority queue to find the 10 biggest (of N) values: we would enqueue N values and then dequeue 10 values. Here is the complexity of these combined operations for each implementation..

```
Implementation 1: N*O(1) + 10*O(N)         = O(N)    + O(N)      = O(N)
Implementation 2: N*O(N) + 10*O(1)         = O(N**2) + O(1)      = O(N**2)
Implementation 3: N*O(Log N) + 10*O(Log N) = O(NLogN) + O(LogN) = O(NLogN)
```

Here, Implementation 1 has the lowest complexity for the combined operations. That makes sense, as the operation done N times (add) is very simple (add to the end of a list/the front of a linked list, where each add is O(1)) and the operation done a constant number of times (10, independent of N) is the expensive operation (remove, which is O(N)). It even beats the complexity of Implementation 3. So, as N gets bigger, implementation 1 will eventually become faster than the other two for the "find the 10 biggest" task.

So, the bottom line here is that sometimes there is NOT a "best all the time" implementation for a data structure. We need to know what problem we are solving (the complexity classes of all the operations in various implementations and HOW OFTEN we must do these operations) to choose the most efficient implementation for solving the problem.
