# Python Intermediate Programming

## Unit 4: Algorithmic Study

CHAPTER 11: THE COMPLEXITY OF PYTHON OPERATIONS

DR. ERIC CHOU IEEE SENIOR MEMBER

# Objectives

- In this lecture we will learn the complexity classes of various operations on Python data types. Then we will 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 Empirical Analysis, when we do run code and take measurements).

# Python Complexity Classes

LECTURE 1

# Python Complexity Classes

- In Data Structure Course, 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 reference) is `O(1)`. Simple operators on integers (whose values are small: e.g., under 12 digits) like `+` or `==` are also `O(1)`. Assume small integers unless explicitly told otherwise.
- In all these examples, `N = len(data-type)`. The operations are organized by increasing complexity

## Lists: Complexity

| Operation | Example | Class | Notes |
|---|---|---|---|
| **Index** | `l[i]` | O(1) | |
| **Store** | `l[i]=0` | O(1) | |
| **Length** | `len(l)` | O(1) | |
| **Append** | `l.append(5)` | O(1) | |
| **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) | |

Note: Tuples support all operations that do not mutate the data structure (and with the same complexity classes).

| Lists: | Complexity | | |
|---|---|---|---|
| Operation | Example | Class | Notes |
| **Containment** | x in/not in l | O(n) | 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) | searches list |
| **Reverse** | l.reverse() | O(n) | |
| **Iteration** | for v in l: | O(n) | |
| **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) |

Note: Tuples support all operations that do not mutate the data structure (and with the same complexity classes).

| Sets: | Complexity | | |
|---|---|---|---|
| **Operation** | **Example** | **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) | |
| **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 |

## Sets: Complexity

| Operation | Example | Class | Notes |
|---|---|---|---|
| **<= / <** | 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) | |
| **Copy** | s.copy() | O(n) | |

# Python Complexity Classes

- 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 (which lists/tuples require) allows for faster set operations.
- Frozen sets support all operations that do not mutate the data structure (and with the same complexity classes).

## Dictionaries: dict and defaultdict Complexity

| Operation | Example | 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/setdefautlt** | d.Method | O(1) | |
| **Pop** | d.pop(k) | O(1) | |
| **Pop Item** | d.popitem() | O(1) | |
| **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 |

So, most dict operations are O(1).

# Python Complexity Classes

- `defaultdicts` support all operations that dicts support, with the same complexity classes (because it inherits all the 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.

# Python Complexity Classes

- 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 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 strings.

# Composing Complexity Classes: Sequential and Nested Statements

LECTURE 2

# Composing Complexity Classes: Sequential and Nested Statements

- In this section we will learn how to combine complexity class information about simple operations into complexity information about complex operations (composed from simple operations). The goal is to be able to analyze all the statements in a function/method to determine the complexity class of executing the function/method.

# Law of Addition for big-O notation

LECTURE 3

# 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 drop the lower added term). So,

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

because N is the faster growing term.

# Law of Addition for big-O notation

- This rule helps us understand how to compute the complexity 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 SEQUENTAILLY 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)`.

# Law of Addition for big-O notation

- Note that for an if statement like

```
if test:          assume complexity of test is O(T)
    block 1       assume complexity of block 1 is O(B1)
else:
    block 2       assume complexity of 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. In the worst case, the if will execute the block with the largest complexity. So, given

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

# Law of Addition for big-O notation

- 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).
```

# Law of Multiplication for big-O notation

LECTURE 4

# Law of Multiplication 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 is `O(N)*O(f(N)) = O( Nf(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)`

# Law of Multiplication for big-O notation

- This rule helps us understand how to compute the complexity 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(s) 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.
- Let's use the data and tools discussed above to analyze (determine their complexity classes) 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.

# Law of Multiplication for big-O notation

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 containing all values after the one at index i.

```
def is_unique1 (alist : [int]) -> bool:          O(N)
    for i in range(len(alist)):                  O(N)
        if alist[i] in alist[i+1:]:              O(N)
            return False                         O(1)
    return True                                  O(1)
```

O(N)
O(N) - index+add+slice+in: O(1)+O(1)+O(N)+O(N) = O(N)
O(1) - never executed in worst case
O(1)

# Law of Multiplication for big-O notation

- 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.
- Note that in the worst case, we never return False and keep executing the loop, so this O(1) does not appear in the answer. Also, in the worst case the list slice is aliset[1:] which is O(N-1) = O(N).

# Law of Multiplication for big-O notation

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: it turns out that copying the list does not increase the complexity class of the method.

```
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 is O(1)
        if copy[i] == copy[i+1]:
                    O(1): +, 2 [i],and  == ints: all O(1)
            return False             O(1) - never executed in worst case
    return True                      O(1)
```

# Law of Multiplication for big-O notation

- 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 unque2 will eventually run faster. Because we don't know the constants, we don't know which is faster for small N.

# Law of Multiplication for big-O notation

- 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, 2000 Log 2000 got 2.2 times bigger, or 10% bigger than just doubling).

# Law of Multiplication for big-O notation

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

$$\frac{T(2N)}{T(N)} = \frac{c^{*}2(N\ Log\ N) + c^{*}2N}{c^{*}(N\ Log\ N)} = 2 + \frac{2}{Log\ N}$$

- 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.

# Law of Multiplication for big-O notation

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)
```

# Law of Multiplication for big-O notation

- 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).

# Law of Multiplication for big-O notation

- 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 running it) to determine their complexity classes. For large problem sizes, the algorithm/function with the smallest complexity class will be best. For small problem sizes, complexity classes don't determine which is best (we need to take into account the CONSTANTS and lower order terms when problem sizes are small), but we could run the functions (dynamic analysis, aka empirical analysis) to test which is fastest on small problem sizes.

# Using a Class (implementable 3 ways) Example

LECTURE 5

# 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.

# Using a Class (implementable 3 ways) Example

- In a priority queue, we can add values and remove values to 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; there, whoever is first in the line -has been standing in line longest- is admitted).

# Using a Class (implementable 3 ways) Example

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

| | add | remove |
|---|---|---|
| Implementation 3 | O(1) | O(n) |
| Implementation 2 | O(n) | O(1) |
| Implementation 1 | O(log n) | O(log n) |

# Using a Class (implementable 3 ways) Example

- 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).

# Using a Class (implementable 3 ways) Example

- 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).

# Using a Class (implementable 3 ways) Example

- 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).