Order of Growth Analysis

1. **Big-O notation**
   a. Gives us a concept of how long an algorithm will take to run, regardless of what machine it’s run on.
   b. When analyzing complexity using Big-O, we don’t care about lower-order terms or constants.
      i. Highest order term trumps the others.
         1. $O(n^2) + O(n) + O(1) \rightarrow O(n^2)$
      ii. Multiplicative or additive constants don’t matter.
         1. $O(10*n) \rightarrow O(n)$
         2. $O(n + 1) \rightarrow O(n)$

2. **Common orders of growth**
   a. $O(1)$: **constant time**, independent of input size
   b. $O(\log(n))$: **logarithmic time**
   c. $O(n)$: **linear time**
   d. $O(n \log(n))$: **log-linear time**
   e. $O(n^2)$, $O(n^3)$, etc.: **polynomial time**
   f. $O(2^n)$: **exponential time** (bad)

Complexity of built-in Python methods

**Constant-time operations:**
   a. Assignment
   b. Basic operations $+$ $-$ $*$ $/$ $>$ $<$

Some built-in methods for data structures in Python are also constant-time, but many are not. It is important to realize that although we don’t look at the underlying machinery of many built-in Python methods, the complexity of their implementations affects the complexity analysis of our own methods.

For example:

1. Dictionaries
   c. look-up: $O(1)$
   d. length: $O(1)$
2. Lists
   a. append: O(1)
   b. length: O(1)
   c. insert: O(n)
   d. delete: O(n)
   e. copy: O(n)
   f. sort: O(n log n)
   g. check if an item is in the list: O(n)

**Strategies for Order of Growth Analysis**

1. Loops
   a. # of iterations in the loop
   b. Amount of work within each loop.

2. Recursive calls
   a. # of recursive calls that are made
   b. Amount of work done for each recursive call
   c. Draw the recursive tree

3. What is “n” in terms of the input?

**Examples**

1. **Check if two lists are equivalent.**

```python
def checkEquivalentLists(L1, L2):
    if len(L1) != len(L2):
        return False
    for i in range(len(L1)):
        if L2[i] != L1[i]:
            return False
    return True
```

**Main points:** len() is O(1), checking equivalence is O(1). n = len(L1) (or len(L2), they are the same). Answer: O(n)
2. Find the intersection of two lists in three ways.

```python
def findIntersection(L1, L2): # doesn’t handle duplicates.
    intersection = []
    for i in L1:
        for j in L2:
            if i == j:
                intersection.append(i)
    return intersection

-------

def findIntersection(L1, L2):
    intersection = []
    for i in L1:
        if i in L2 and i not in intersection:
            intersection.append(i)
    return intersection

-------

def findIntersection(L1, L2): # doesn’t handle duplicates.
    intersection = []
    seen = {}
    for i in L1:
        seen[i] = "yes"
    for j in L2:
        if j in seen:
            intersection.append(j)
    return intersection
```

**Main points:** Nested loops (example 1). Dictionary look-ups are O(1) (example 3), whereas checking if an item is in a list is O(n) (example 2). The first two are O(n^2), the last method is O(n), n = len(L1). Assumes lengths of L1 and L2 are similar.

3. An example with a while loop

```python
def beep(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
```
Main points:
- At every iteration, n is halved. This means there are $\log_2(n)$ loops.
- $\log_2(n) \rightarrow O(\log n)$
- What happens to “sum” basically doesn’t matter in terms of the order of growth analysis, because it doesn’t affect the # of loops and sum += n is a constant-time operation.

4. Bisection Search (example from lecture). bSearch() runs in $O(n \log n)$ whereas bisectSearch() runs in $O(\log n)$.

```python
def bSearch(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)/2
        if L[half] > e:
            return bSearch(L[:half], e)
        else:
            return bSearch(L[half:], e)
```

Main points in bSearch:
1. L[ : half] or L[half : ] -- How does this affect the size of the input in each recursive call? How many recursive calls are there?
2. L[ : half] -- Not a constant time operation.
3. How many recursive calls to bSearch() are there?

```python
def bisectSearch(L, e):
    """Assumes L is a list with elements in ascending order.
    Returns True if e is in L and False otherwise"
    def bisectSearch_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
```
elif L[mid] > e:
    if low == mid: #nothing left to search
        return False
    else:
        return bisectSearch_helper(L, e, low, mid - 1)
else:
    return bisectSearch_helper(L, e, mid + 1, high)
if len(L) == 0:
    return False
else:
    return bisectSearch_helper(L, e, 0, len(L) - 1)