Big O for Python Data Structures

In this lecture we will go over the Big O of built-in data structures in Python: Lists and Dictionaries.

Lists

In Python lists act as dynamic arrays and support a number of common operations through methods called on them. The two most common operations performed on a list are indexing and assigning to an index position. These operations are both designed to be run in constant time, O(1).

Let's imagine you wanted to test different methods to construct a list that is [0,1,2...10000]. Let go ahead and compare various methods, such as appending to the end of a list, concatenating a list, or using tools such as casting and list comprehension.

For example:

In [3]:
```
def method1():
    l = []
    for n in range(10000):
        l = l + [n]

def method2():
    l = []
    for n in range(10000):
        l.append(n)

def method3():
    l = [n for n in range(10000)]

def method4():
    l = list(range(10000))
```

Let's now test these methods using the timeit magic function:

In [4]:
```
%timeit method1()
%timeit method2()
%timeit method3()
%timeit method4()
```

```
184 ms ± 9.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
829 µs ± 125 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
574 µs ± 134 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
194 µs ± 11.8 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

We can clearly see that the most effective method is the built-in range() function in Python!

It is important to keep these factors in mind when writing efficient code. More importantly begin thinking about how we are able to index with O(1). We will discuss this in more detail when we cover arrays general. For now, take a look at the table below for an overview of Big-O efficiencies.

Table of Big-O for common list operations

Please note, in order to see this table, you may need to download this .ipynb file and view it locally, sometimes GitHub or nbveiwer have trouble showing the HTML for it...

Operation	Big-O Efficiency
index []	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n+k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

Dictionaries

Dictionaries in Python are an implementation of a hash table. They operate with keys and values, for example:

```
In [5]: d = {'k1':1,'k2':2}
```

```
In [6]: d['k1']
```

```
Out[6]: 1
```

Something that is pretty amazing is that getting and setting items in a dictionary are O(1)! Hash tables are designed with efficiency in mind, and we will explore them in much more detail later on in the course as one of the most important data structures to undestand. In the meantime, refer to the table below for Big-O efficiencies of common dictionary operations:

Operation	Big-O Efficiency
copy	O(n)
get item	O(1)
set item	O(1)

Operation	Big-O Efficiency
delete item	O(1)
contains (in)	O(1)
iteration	O(n)

Conclusion

By the end of this section you should have an understanding of how Big-O is used in Algorithm analysis and be able to work out the Big-O of an algorithm you've developed. Get ready, there's a quiz up next!