MCS 375 - Fall 2020 Project 1: Find Duplicates
Ben Schwartz

The algorithms I am creating are meant to check if a list in python has any duplicates or repeated elements. For example, [1,1] and [“word”, “word”] are both duplicates. [1,2,3,4,5] does not have any duplicates. I am creating two algorithms. One that runs in O(n^2) time and one that runs in O(n) time. The worst-case run time for each algorithm is a list that has no duplicates as that will have to parse the entire list.

Algorithm 1:

```
O(1)    Function has_duplicates1(list)
O(1)        Initialize contains_duplicates to False

O(n)        For every element in list:
O(n)            For every element in list:
O(1)                if the two elements are equal and not the same index:
O(1)                    set contains_duplicates to True

O(1)        return contains_duplicates
```

The first algorithm checks every element in the list with every other element in the list to see if they are the same. If any element matches another one that means there is a duplicate element in the list.

Defining a function and initializing a variable will always have a constant run time.
The two loops iterate through every element in the list so they will repeat n times.
Inside the loop, we are checking if two specific elements are equal to each other. Since it only has to look at one element of each list it has a constant run time.
Lastly, changing contains_duplicates to true and returning contains_duplicates are both independent of list size so they will have constant run time.

The big O run time for each line is shown left of the pseudocode above.
The total run time is O(1) + O(1) + O(n)(O(n)(O(1) + O(1))) + O(1)
So the nested loop is the highest run time and our algorithm is O(n^2)

Algorithm 2:

```
O(1)    Function has_duplicates2(list)
O(1)        Initialize testSet as an empty set
O(1)        Initialize contains_duplicates to False

O(n)        For every element in list:
O(1)            initialize length1 to the length of testSet
O(1)            Add the element to testSet
O(1)            initialize length2 to the length of testSet
O(1)            If the length1 is equal to length2:
O(1)                set contains_duplicates to True

O(1)        return contains_duplicates
```

The second algorithm takes every element in the list and adds it to a set. Every time it does this it checks the length of the set before and after. Since a set can't contain any duplicates the length of the set will remain unchanged if a duplicate element is added. So if the length of the set is unchanged after adding the element from the list that means the original list must contain a duplicate element.

Defining the function, initializing an empty set, and initializing contains_duplicates do not depend on the length of n so they all have constant run time.

The loop is iterating through every element in the list so it will have a run time of O(n).

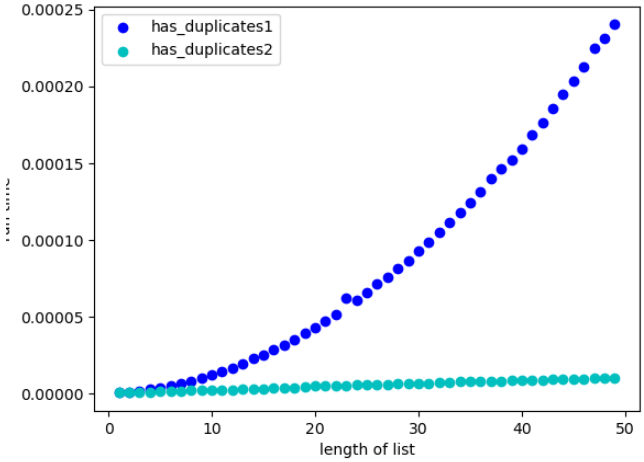
The computer stores the length of a list so to find the length the program simply has to look it up. This gives a constant run time to the first and third lines in the loop that are initializing variables to store the lengths of the set.

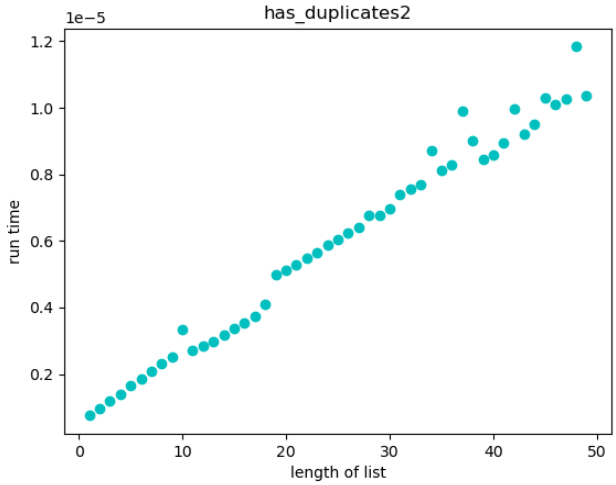
Adding an element to a set also has a constant run time. Sets are stored in hash tables. Adding an element to the set creates a value to store in the hash table. Set's can't store the same value twice so it also has to check if the value is already in the set. To check if the value is in the hash table it computes the hash of that value and checks what entry it is pointing to in the table. The size of the set does not change these operations so we get a constant run time.

Lastly, we are checking if the two length variables are equal, setting contains_duplicates to true, and returning contains_duplicates. None of these operations are dependent on the length of the list so they all have a constant run time.

The big O run time for each line is shown left of the pseudocode above.
The total run time is O(1) + O(1) + O(1) +O(n)(O(1) + O(1) + O(1) + O(1) + O(1)) + O(1)

Our largest run time is coming from the loop repeating n times. Everything within the loop has constant run time so the overall algorithm is O(n)





These plots show the run time for increasing sizes of the list for the two algorithms. The first plot shows the run time for both the first algorithm(has_duplicates1) and the second algorithm(has_duplicates2). The second plot shows a more clear representation of the run times for the second algorithm.

They were created by running each has_duplicates function 100 times for each list of sizes 1 to 50 and averaging the times. Each list was created with all distinct elements to show the worst-case run time of each function.

We can see from the plots that the function has_duplicates1 has an increasing growth rate. As n gets larger the growth rate increases as well as the run time. It appears to have a quadratic growth rate which matches our earlier analysis that the algorithm is O(n^2). The function has_duplicates2 looks fairly linear. As n increases the run time also increases but at a constant rate. A linear plot matches the earlier analysis that the algorithm is O(n).