PSEUDOCODE And Theoretical Analysis for O(1):

- We seek to produce a full row when given a particular SKU.
- Method: Dictionary lookup

```
FUNCTION dict_lookup(dataframe):
    # if it exists, print out the information for that SKU
    IF SKU in dict: # condition check +1
        RETURN dict[SKU] #  calls dict[hashed value] +1
    END IF
END FUNCTION
```

```
Total Time = 1 + 1 = 2 = O(1)
```

PSEUDOCODE And Theoretical Analysis for O(n):

- Goal: create a set, use set to check for any duplicates in data

```
FUNCTION Duplicate_search_set(data)
    #create a set
    s={}  # +1
    #create a counter for number of duplicates
    duplicates = 0  # + 1
    # create a list for list of duplicates
    duplicate_list = [] # + 1
    #iterate through data and check against set
    FOR i in data # loops n times
        # Check if already in set
        IF i in set # + 1
            duplicates++ # +1
            duplicate_list.append(i) # +1
        END IF
        # Add element to set
        s.add(i) # +1
    END FOR
    # return counter, duplicate list
    RETURN duplicates, duplicate_list + 2
END FUNCTION
```

```
Total Time = 1 + 1 + 1 + n*(1+ 1 + 1) + 1 + 2 = 3*n + 6 = O(n)
```

PSEUDOCODE And Theoretical Analysis for O(nlogn) Part 1

- Goal: create a dictionary of category: sorted heap
- Create a dictionary grouping by category

```
Category_dict = {category: Dataframe} # +n+klogk operations since groupby scans
    # n rows then sorts in klogk where k is number of groups
Heap_dictionary = {} # + 1 create a empty dictionary mapping category to sorted heap
FOR category in Category_dict: # k loops where k is number of categories.
    # Create a list to store the heap and temp variable that stores dataframe
    heap = [] # +1
    dataframe = category_dict[category] # +1
        # store each row of dataframe as tuple with Sellability coming first, add to heap
```

```
        FOR row in dataframe: # around n/k loops where k is number of categories
            tuple = (row[Sellability], row) # + 1
            heap.append(tuple) + 1
        END FOR

    #ADD the heap to heap_dict
    heap_dict[category] = heap + 1
END FOR

Total Time =  klogk + 1 + k*(1 + 1 + n/k*(1 + 1) + 1) = klogk + 1 + k(3 + 2n/k) = klogk + 1 + 3k
Therefore in worst case, if every row is a different category, O(n) = nlogn
```

PSEUDOCODE And Theoretical Analysis O(nlogn) Part 2

- GOAL: turn each list in heap dicionary into a minheap

```
FUNCTION(dictionary of lists)
FOR category in dictionary of lists: # k iterations where k is number of categories
    heapq.heapify(dictionary[category]) # Floyd algorithm takes O(n) time to heapify
    # = starting from the 2nd to last level, there are n/4 nodes which each have to do a maximum
    # = sum as h goes from 0 to logn (n*h/2^(h+1))
    # = converges to O(n)
END FOR
END FUNCTION

Time complexity: O(n)
```

PSEUDOCODE And Theoretical Analysis O(nlogn) Part 3

- GOAL: sort heaps in the heap dictionary

```
FUNCTION sort_heap(heap): # to run k times (once for each category)
    sorted_heap = [] # create empty sorted heap +1
    WHILE heap is not empty: # condition check +1; runs n/k times where k is # of categories
        Pop off the heap # 1 + log(n/k) to percolate down heap
        Append to sorted_heap # +1
    END WHILE
    RETURN sorted_heap # + 1
END FUNCTION

ADD each sorted_heap to dictionary # + k operations (k is number of categories and minheaps)

Total time = k * (1 + n/k*(1+1+log(n/k))) + k = k + 2n + n*log(n/k) + k
 = 2k + 2n + nlog(n)-nlog(k) = O(nlogn)

Time Complexity across all 3 parts: nlogn + n + nlogn = O(nlogn)
```

PSEUDOCODE And Theoretical Analysis for O(n^2)

```
FUNCTION duplicate_search(Data):
    # create an empty list to store duplicates
    Duplicate_list = [] # +1
    # create a counter of duplicates
    Duplicates = 0 # +1
    FOR i in range(Data Length): # n times
```

```
        FOR j in range(i+1, Data Length): # loops n-1 then n-2 , ..., then 1 time
            IF Data[i] == Data[j]: # +1
                Duplicate_list.append(Data[i]) # +1
                Duplicates ++ # +1
            END IF
        END FOR
    END FOR
    RETURN
END FUNCTION
```

Total nested loops: 1 + 2 + (n-2) + (n-1) = (n-1)((n-1) + 1)/2 = n(n-1)/2
Total time = n(n-1)/2 * 3 + 1 + 1 = 2 + 3n(n-1)/2 = O(n^2)