A List as a Subset of Another List

```
Def is_subset(l1,l2):
	s=set(l1)
	For elem in l2:
		If elem not in s:
			Return false
	Return true
```

For a lookup list with **m** elements and a subset list with **n** elements, the time complexity is *O(m+n)*.

Check if Lists are Disjoint

```
Def is_disjoint(l1,l2):
	s=set(l1)
	For elem in l2:
		If elem in s:
			Return false
	Return true
```

Same time complexity of O(m+n)

Find Symmetric Pairs in a List

```
Def sym_pairs(l):
	pair_set=set()
	result=[]
	For pair in l:
		pair_tup=tuple(pair)
		pair.reverse()
		reverse_tup=tuple(pair)
		If reverse_tup in pair_set:
			result.append(pair_tup)
			result.append(reverse_tup)
		Else:
```

```
                pair_set.add(pair_tup)
        Return result
```

The hash table lookups work in constant time. Hence, our traversal of the input list makes the algorithm run in *O(n)* where **n** is the list size.

Trace the Complete Path of a Journey

```
Def path(dict):
        reverse_dict={}
        result=[]
        keys=dict.keys()

        For key in keys:
                reverse_dict[dict.get(key)]=key
        from=None
        rev_keys=reverse_dict.keys()

        For key in keys:
                If key not in reverse_dict:
                        from=key
                        Break

        to=dict.get(from)
        While to:
                result.append([from,to])
                from=to
                to=dict.get(to)
        Return result
```

Although a hash table is created and traversed, both take the same amount of time. The complexity for this algorithm is *O(n)* where **n** is the number of source-destination pairs.

Find Two Pairs Such That a+b = c+d

```
Def pairs(list):
    result=[]
    d=dict()
    For i in range(len(list)):
        For j in range(i+1,len(list)):
            added= list[i]+list[j]

            If added not in d:
                d[added]=[ list[i],list[j] ]
            Else:
                prev_pair=d.get(added)
                curr_pair= [ list[i],list[j] ]
                result.append(prev_pair)
                result.append(curr_pair)
                Return result
    Return result
```

The time complexity of this algorithm is $O(n_2)$ for two for loops.

A Sublist with a Sum of 0

```
Def find_sum_0(list):
    d=dict()
    total_sum=0
    For elem in list:
        total_sum+=elem
        If elem is 0 or total_sum is 0 or d.get[total_sum]:
            Return True
        d[total_sum]=elem
    Return false
```

A linear iteration over **n** elements means that the algorithm's time complexity is *O(n)*.

Word Formation Using a Hash Table

```
Def is_formation(word_list, the_word):
        If len(the_word) <2 or len(word_list)<2:
            Return false
        hashtable=HashTable()
        For elem in word_list:
            hashtable.insert(elem, true)
        For i in range(1, len(the_word)):
            first=the_word[0:i]
            second=the_word[i:len(word)]
            check1=False
            check2=False
            If hashtable.search(first):
                check1=True
            If hashtable.search(second):
                Check2 =true
            If check1 and check2:
                Return true
        Return false
```

Note: The solution only works for two words and not more.

We perform the insert operation **m** times for a list of size **m**. After that, we linearly traverse the `word` of size **n** once. Furthermore, we slice strings of size **n** in each iteration. Hence the total time complexity is

O(m + n)2

Find Two Numbers that Add up to "k"

```
Def two_sum(list, k):
    s=set()
    For elem in list:
        If k-elem in d:
            Return [elem, k-elem]
        s.add(elem)
```

```
        Return false
```

The time complexity of the solution above is
O(n)

First Non-Repeating Integer in a List

```
Def first_unique(list):
        counts=collections.OrderedDict() or dict()
        counts=counts.fromkeys(list,0)
        For elem in list:
                counts[elem]+=1
        For elem in counts:
                If counts[elem]==1:
                        Return elem
        Return none
```

Since the list is only iterated over only once, therefore the time complexity of this solution is linear, i.e.,
O(n).

Detect Loop in a Linked List

```
Def detect_loop(ll):
        Visited nodes=set()
        curr=ll.head
        While curr:
                If curr not in visited_nodes:
                        visited_nodes.add(curr.data)
                        curr=curr.next
                else:
                        Return true
        Return false
```

We iterate the list once. On average, lookup in a set takes $O(1)$ time. Hence, the average runtime of this algorithm is $O(n)$. However, in the worst case, lookup can increase up to $O(n)$, which would cause the algorithm to work in $O(n^2)$.