Leet Code #3: longest substring w/o repeating Characters.

• Brute force approach:

```
         { for i in string
O(N²)    {     for j in string
                    substring = string.substring(i, j+1)
         { if has unique Chars(substring)
O(N)     {      result = max(result, len(substring))
```

Time complexity = $O(N^3)$

More optimized solution:
  Sliding Window Algorithm

• Window (or section) formed over parts of data (in this case parts of the string) we move window over increments of the data.

P $\boxed{W\ W}$ K E W ← window

• Sliding window is just looking at data incrementally.

ex1.)

```
          0   1   2   3   4   5
          P   W   W   K   E   W
          :
pointers < i
          j
```

Max = 0
set = [ ]

• max keeps track of length of substring w/ no repeating Char.

• set keeps track of unique char in substring window.

length of window is $i-j+1$

• move i pointer first
• check if letters are in set.
• j moves if you find duplicate.

• When i is greater than string length, return Max leet code #3: length of longest substring continued.

- return an int.
- first: check if our input is valid
  - if its null or empty, we know there's no substring; return ∅.
  - Initialize values i, j, max, set.
  - Set = new hash set.
- next, we know we need to move pointers, starting w/ i.
  - extract char at i pointer position.
  - this where sliding window comes in.
    - we need to check if if CharAt(i) is already in set.
    - if set contains char "c", remove the char that j is pointing to from the set.
- when we come out of loop we know that char c is not in our set.
  - we can do set.add(c)

Time complexity is linear...In the worst case i & j have to touch every single char. Algo is techincally O(2N), but 2 can be dropped to be O(N).

Space complexity: O(N), where N is the size of our string

Worst case is ~~that~~ ~~all~~ values are unique and set contains whole string.