Question 3

Basic Algorithm: The basic approach for this algorithm is to consider every elements as the starting point and compare with the desired substring. Below is a persu do code for the algorithm:

```
for i from 0 to len(input(String)){
    Check if start from position i, the following characters matches the substring required
    Return the index if found. (else, if not found return false)
}
```

However this algorithm is not efficient enough to solve the given cases since it will do character by character check every time.

To improve this algorithm, we can use hash functions. The basic idea is as follows:
We choose a good hash function. The general idea is as follows:

```
hash_value1 = hash( input_substring)
for i from 0 to n-m{
     hash_value2 = hash[input_string[i ….i+m]
     check if hash_value1 == hash_value2{
      If yes,  do the string comparison.
         {if yes, return the index, otherwise continue}
}
Return false
}
```

This algorithm hashes the input_substring and checks it with the other string. The running time for this algorithm is O(n*(length of substring). This algorithm will be not efficient if the length of the substring becomes very large.

To optimize this algorithm:
A good hash function generate the hash values based on previous values so that the cost of generate hash value for next substring check will only be O(1) instead of O(m) where m is the length of input_substring.
Below is my hash function with examples:
input _String = [A ,G ,G ,C ,T ,G]

S1 = [A, G, G, C]
S2 = [G, G, C, T]
S3 = [G, C, T, G]

This is 3 substring of length 4 with the input String.
We use ASCII values of the input string A = 65 G = 71 C = 67 T = 84. We choose a relative large prime: 163.

The hash function for S1 is as follows:
S1 = [A, G, G, C]

h(s1) = 65* 163^3 + 71* 163^2 + 71* 163^1 + 67*163^0 = 283396594

S2 is the next examined sub_string: S2 = [G, G, C, T] and should be hashed based on S1.

h(s2) = [163 *(283396594 - (163 * 163^3)] + 97*163^0

In general, the next hash function is:
h(next_hash) = [163* (h(old_hash) - value of old character) + value of new character)]
This method reduces the time to generate the next hash to O(1)

As a result, in the above code, we can generate the new hash for the input string based on the old hash function using this hash method. We still have to do the character by character check if the hash value matches since there might be possible collisions.

The overall running time for this algorithm is O(n) instead of O(n*m) where m is the length of the substring. The reason is that the above hash function generate new hash function based on the hold hash value and a new character.