```python
# Following program is the python implementation of
# Rabin Karp Algorithm given in CLRS book

# pat  -> pattern
# txt  -> text
# q    -> A prime number


def search(pat, txt, q):
    m = len(pat)
    n = len(txt)
    j = 0
    p = 0  # hash value for pattern
    t = 0  # hash value for txt
    h = 1

    # d is the number of characters in input alphabet
    d = 256

    # The value of h would be "pow(d, m-1)%q"
    for i in range(m - 1):
        h = (h * d) % q

    # Calculate the hash value of pattern and first window
    # of text
    for i in range(m):
        p = (d * p + ord(pat[i])) % q
        t = (d * t + ord(txt[i])) % q

    # Slide the pattern over text one by one
    for i in range(n - m + 1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters one by one
        if p == t:
            # Check for characters one by one
            for j in range(m):
                if txt[i + j] != pat[j]:
                    break

            j += 1
            # if p == t and pat[0...m-1] = txt[i, i+1, ...i+m-1]
            if j == m:
                print("Pattern found at index " + str(i))

        # Calculate hash value for next window of text: Remove
        # leading digit, add trailing digit
        if i < n - m:
            t = (d * (t - ord(txt[i]) * h) + ord(txt[i + m])) % q

            # We might get negative values of t, converting it to
            # positive
            if t < 0:
                t = t + q


# Driver program to test the above function
txt = "a hello there to you all! hello all!"
pat = "hello"
q = 101  # A prime number
search(pat, txt, q)
```

```python
def find_boyer_moore(T, P):
    """Return the lowest index of T at which substring P begins (or else -1)."""
    n, m = len(T), len(P)  # introduce convenient notations
    if m == 0: return 0  # trivial search for empty string
    last = {}  # build 'last' dictionary
    for k in range(m):
        last[P[k]] = k  # later occurrence overwrites
    # align end of pattern at index m-1 of text
    i = m - 1  # an index into T
    k = m - 1  # an index into P
    while i < n:
        if T[i] == P[k]:  # a matching character
            if k == 0:
                return i  # pattern begins at index i of text
            else:
                i -= 1  # examine previous character
                k -= 1  # of both T and P
        else:
            j = last.get(T[i], -1)  # last(T[i]) is -1 if not found
            i += m - min(k, j + 1)  # case analysis for jump step
            k = m - 1  # restart at end of pattern
    return -1
```

```
import time
from collections import defaultdict, Counter


def get_suffix_array(str):
    return sorted(range(len(str)), key=lambda i: str[i:])


def sort_bucket(str, bucket, order):
    d = defaultdict(list)
    for i in bucket:
        key = str[i:i + order]
        d[key].append(i)
    result = []
    for k, v in sorted(d.items()):
        if len(v) > 1:
            result += sort_bucket(str, v, order * 2)
        else:
            result.append(v[0])

    # print(d)

    return result


def suffix_array_ManberMyers(str):
    return sort_bucket(str, (i for i in range(len(str))), 1)


if __name__ == "__main__":
    str = 'Four score and seven years ago our forefathers brought forth to this 
continent a new nation'

    start_time = time.time()
    x = get_suffix_array(str)
    end_time = time.time()
    print("Time for python sort was %g seconds" % (end_time - start_time))

    start_time = time.time()
    y = suffix_array_ManberMyers(str)
    end_time = time.time()
    print("Time for Manber Myers was %g seconds\n" % (end_time - start_time))

    # for i in y:
    #     print("{}\t{}".format(i, str[i:]))
```

```python
def find_kmp(T, P):
    """Return the lowest index of T at which substring P begins (or else -1)."""
    n, m = len(T), len(P)  # introduce convenient notations
    if m == 0: return 0  # trivial search for empty string
    fail = compute_kmp_fail(P)  # rely on utility to precompute
    j = 0  # index into text
    k = 0  # index into pattern
    while j < n:
        if T[j] == P[k]:  # P[0:1+k] matched thus far
            if k == m - 1:  # match is complete
                return j - m + 1
            j += 1  # try to extend match
            k += 1
        elif k > 0:
            k = fail[k - 1]  # reuse suffix of P[0:k]
        else:
            j += 1
    return -1  # reached end without match


def compute_kmp_fail(P):
    """Utility that computes and returns KMP 'fail' list."""
    m = len(P)
    fail = [0] * m  # by default, presume overlap of 0 everywhere
    j = 1
    k = 0
    while j < m:  # compute f(j) during this pass, if nonzero
        if P[j] == P[k]:  # k + 1 characters match thus far
            fail[j] = k + 1
            j += 1
            k += 1
        elif k > 0:  # k follows a matching prefix
            k = fail[k - 1]
        else:  # no match found starting at j
            j += 1
    return fail

txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
find_kmp(txt, pat)
```

```python
from bisect import bisect_left, bisect_right

strls = ['a', 'awkward', 'awl', 'awls', 'axe', 'axes', 'bee']

# t = 'This is the world.$'
t = 'banana$'

suffixes = sorted([t[i:] for i in range(len(t))])

print(len(suffixes))
print(suffixes)

# Get range of elements with ‘aw’ as a prefix
st, en = bisect_left(strls, 'aw'), bisect_left(strls, 'ax')
print(st, en)  # output: (1, 4)
```

```python
"""
An implementation of Boyer-Moore-Horspool string searching.
"""
from collections import defaultdict


def boyer_moore_horspool(pattern, text):
    m = len(pattern)
    n = len(text)

    if m > n:
        return -1

    skip = defaultdict(lambda: m)
    found_indexes = []

    for k in range(m - 1):
        skip[ord(pattern[k])] = m - k - 1

    k = m - 1

    while k < n:
        j = m - 1
        i = k
        while j >= 0 and text[i] == pattern[j]:
            j -= 1
            i -= 1
        if j == -1:
            found_indexes.append(i + 1)

        k += skip[ord(text[k])]

    return found_indexes


if __name__ == '__main__':

    tests = [
        [[8, 25], 'the', 'this is the string to do the search in'],
        [[0, 2, 10], 'co', 'cocochanelco'],
        [[2, 6], 'co', 'mycocacola'],
        [[2, 4, 6, 9], 'co', 'mycococoacola'],
        [[2, 4], 'coco', 'mycococoacola'],
        [[10], 'co', 'lalalalalaco'],
        [[0], 'co', 'colalalalala'],
        [[], 'a', 'zzzzzzzzzzz'],
        [[0], 'a', 'a'],
        [[], 'z', 'a'],
        [[], 'z', 'aa'],
        [[1], 'z', 'az'],
        [[0], 'z', 'za'],
        [[r for r in range(11)], 'z', 'zzzzzzzzzzz'],
        [[5, 6], 'z', 'aaaaazzaaaaa'],
    ]

    for test in tests:
        assert boyer_moore_horspool(test[1], test[2]) == test[0]
```

```python
def KMPSearch(pat, txt):
    m = len(pat)
    n = len(txt)

    # create lps[] that will hold the longest prefix suffix
    # values for pattern
    lps = [0] * m
    j = 0  # index for pat[]

    # Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, m, lps)

    i = 0  # index for txt[]
    while i < n:
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == m:
            print("Found pattern at index " + str(i - j))
            j = lps[j - 1]

        # mismatch after j matches
        elif i < n and pat[j] != txt[i]:
            # Do not match lps[0..lps[j-1]] characters,
            # they will match anyway
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1


def computeLPSArray(pat, m, lps):
    length = 0  # length of the previous longest prefix suffix

    lps[0] = 0  # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < m:
        if pat[i] == pat[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                # This is tricky. Consier the example AAACAAAA
                # and i = 7
                length = lps[length - 1]

                # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i += 1


txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
KMPSearch(pat, txt)
```