Project 3

Mengxiang Jiang
CSEN 5336 Analysis of Algorithms

October 26, 2023

Problem 1. Five DNA sequences of different lengths are given (DNA1.txt, DNA2.txt, etc).

Find the occurences of the pattern ACT, GATTACA, and CATCATCAT in each using the naive algorithm. Submit your code to solve the problem. How much time (in seconds or milliseconds) is required by your computer to run the algorithm?

```
def naive(text, pattern):
    matches = []
    for i in range(0, len(text) - len(pattern)):
        found = True
        for j in range(len(pattern)):
            if text[i+j] != pattern[j]:
                found = False
                break
        if found == True:
            matches.append(i)
    return matches
```

Listing 1: naive algorithm

```
import time
if __name__ == '__main__':
    text = ''
    patterns = ['ACT', 'GATTACA', 'CATCATCATCAT']
    for fname in ['DNA1.txt', 'DNA2.txt', 'DNA3.txt','DNA4.txt', 'DNA5.txt
    ']:
        with open(fname, 'r') as f:
            text = f.read().rstrip()
        for pattern in patterns:
            start_time = time.perf_counter()
            matches = naive(text, pattern)
            end_time = time.perf_counter()
            print(f'For {fname} and pattern {pattern}, the naive algorithm
    took {end_time-start_time} seconds with {len(matches)} matches')
```

Listing 2: main function code

Note: the main function code is similar for running all the algorithms.

```
$ python naive.py
    For DNA1.txt and pattern ACT, the naive algorithm took
   0.3632765369984554 seconds with 65495 matches
    For DNA1.txt and pattern GATTACA, the naive algorithm took
   0.3577618750023248 seconds with 242 matches
    For DNA1.txt and pattern CATCATCATCAT, the naive algorithm took
   0.3621113970002625 seconds with 0 matches
    For DNA2.txt and pattern ACT, the naive algorithm took
   0.727408620998176 seconds with 131034 matches
    For DNA2.txt and pattern GATTACA, the naive algorithm took
   0.7238480179985345 seconds with 491 matches
    For DNA2.txt and pattern CATCATCATCAT, the naive algorithm took
   0.7162508259971219 seconds with 1 matches
    For DNA3.txt and pattern ACT, the naive algorithm took
   1.4487447140018048 seconds with 262192 matches
    For DNA3.txt and pattern GATTACA, the naive algorithm took
   1.4347486420010682 seconds with 1002 matches
    For DNA3.txt and pattern CATCATCATCAT, the naive algorithm took
   1.4541748980009288 seconds with 1 matches
    For DNA4.txt and pattern ACT, the naive algorithm took
   2.882448870997905 seconds with 524454 matches
    For DNA4.txt and pattern GATTACA, the naive algorithm took
   2.870233528999961 seconds with 1976 matches
    For DNA4.txt and pattern CATCATCATCAT, the naive algorithm took
   2.910066188000201 seconds with 5 matches
    For DNA5.txt and pattern ACT, the naive algorithm took
   5.771988509000948 seconds with 1050689 matches
    For DNA5.txt and pattern GATTACA, the naive algorithm took
   5.708121202998882 seconds with 4161 matches
    For DNA5.txt and pattern CATCATCATCAT, the naive algorithm took
   5.878649592999864 seconds with 3 matches
```

Listing 3: Terminal output running code

Even though the time complexity of the naive algorithm is $O((n-m+1)m)$ in the worst case, in practice the probability the algorithm will break out early makes it perform significantly faster than expected where the times seem to indicate something closer to $O(n)$.

Problem 2. Repeat the previous problem with the Rabin-Karp algorithm.

```
SigMap = {'A': 0, 'T' : 1, 'G' : 2, 'C' : 3}
PriMod = 13

def rabin_karp(text, pattern):
    matches = []
    plen = len(pattern)
    phash = rabin_hash(pattern)
    for i in range(0, len(text) - len(pattern)):
        if i == 0:
            thash = rabin_hash(text[0:len(pattern)])
        else:
            thash = roll(plen, thash,
                         text[i - 1], text[i + plen - 1])
        if thash == phash:
            if text[i:i+plen] == pattern:
                matches.append(i)
    return matches

def rabin_hash(text):
    rhash = 0
    for c in text:
        rhash = (rhash * len(SigMap) % PriMod + SigMap[c]) % PriMod
    return rhash

def roll(plen, oldhash, oldchar, newchar):
    newhash = ((oldhash + PriMod
                - SigMap[oldchar] * (len(SigMap) ** (plen - 1)) % PriMod)
                * len(SigMap) + SigMap[newchar]) % PriMod
    return newhash
```

Listing 4: Rabin-Karp algorithm

```
$ python rabin_karp.py
    For DNA1.txt and pattern ACT, the Rabin-Karp algorithm took
   0.639081138993788 seconds with 65495 matches
    For DNA1.txt and pattern GATTACA, the Rabin-Karp algorithm took
   0.6668751739998697 seconds with 242 matches
    For DNA1.txt and pattern CATCATCATCAT, the Rabin-Karp algorithm took
   0.7319347989978269 seconds with 0 matches
    For DNA2.txt and pattern ACT, the Rabin-Karp algorithm took
   1.2784735669993097 seconds with 131034 matches
    For DNA2.txt and pattern GATTACA, the Rabin-Karp algorithm took
   1.3436157860051026 seconds with 491 matches
    For DNA2.txt and pattern CATCATCATCAT, the Rabin-Karp algorithm took
   1.4682704229999217 seconds with 1 matches
    For DNA3.txt and pattern ACT, the Rabin-Karp algorithm took
   2.5847115970027517 seconds with 262192 matches
    For DNA3.txt and pattern GATTACA, the Rabin-Karp algorithm took
   2.6529827709964593 seconds with 1002 matches
    For DNA3.txt and pattern CATCATCATCAT, the Rabin-Karp algorithm took
   2.936001311005384 seconds with 1 matches
    For DNA4.txt and pattern ACT, the Rabin-Karp algorithm took
   5.080261541996151 seconds with 524454 matches
    For DNA4.txt and pattern GATTACA, the Rabin-Karp algorithm took
   5.316721924995363 seconds with 1976 matches
    For DNA4.txt and pattern CATCATCATCAT, the Rabin-Karp algorithm took
   5.90457762600272 seconds with 5 matches
    For DNA5.txt and pattern ACT, the Rabin-Karp algorithm took
   10.213705198999378 seconds with 1050689 matches
    For DNA5.txt and pattern GATTACA, the Rabin-Karp algorithm took
   10.634073597997485 seconds with 4161 matches
    For DNA5.txt and pattern CATCATCATCAT, the Rabin-Karp algorithm took
   11.496143873002438 seconds with 3 matches
```

Listing 5: Terminal output running code

Despite having better expected time complexity of $O(n)$ compared to the naive algorithm, Rabin-Karp appears to be about twice as slow for these examples.

Problem 3. Repeat the previous problem with the finite automata algorithm.

```
SigMap = {'A': 0, 'T' : 1, 'G' : 2, 'C' : 3}

def finite_automata(text, pattern):
    matches = []
    m = len(pattern)
    tf = computeTF(pattern)

    state = 0
    for i in range(len(text)):
        state = tf[state][SigMap[text[i]]]
        if state == m:
            matches.append(i-m+1)

    return matches

def computeTF(pattern):
    m = len(pattern)
    tf = [[0 for i in range(len(SigMap))] for j in range(m + 1)]
    for state in range(m + 1):
        for c in SigMap:
            nextState = getNextState(pattern, state, c)
            tf[state][SigMap[c]] = nextState
    return tf

def getNextState(pattern, state, c):
    # c matches next character in pattern
    if state < len(pattern) and c == pattern[state]:
        return state + 1

    # find longest prefix which is also the suffix
    # of pattern[0:nextState]
    i = 0
    for nextState in range(state, 0, -1):
        if pattern[nextState-1] == c:
            while (i < nextState - 1):
                pfxidx = state - nextState + 1 + i
                if pattern[i] != pattern[pfxidx]:
                    break
                i += 1
            if i == nextState - 1:
                return nextState
    return 0
```

Listing 6: finite automata algorithm

```
$ python finite_automata.py
    For DNA1.txt and pattern ACT, the finite automata algorithm took
   0.15551975400012452 seconds with 65495 matches
    For DNA1.txt and pattern GATTACA, the finite automata algorithm took
   0.14494252900476567 seconds with 242 matches
    For DNA1.txt and pattern CATCATCATCAT, the finite automata algorithm
   took 0.14850315799412783 seconds with 0 matches
    For DNA2.txt and pattern ACT, the finite automata algorithm took
   0.30578777501068544 seconds with 131034 matches
    For DNA2.txt and pattern GATTACA, the finite automata algorithm took
   0.29742643699864857 seconds with 491 matches
    For DNA2.txt and pattern CATCATCATCAT, the finite automata algorithm
   took 0.2898057919956045 seconds with 1 matches
    For DNA3.txt and pattern ACT, the finite automata algorithm took
   0.6287656860076822 seconds with 262192 matches
    For DNA3.txt and pattern GATTACA, the finite automata algorithm took
   0.59431980199588 seconds with 1002 matches
    For DNA3.txt and pattern CATCATCATCAT, the finite automata algorithm
   took 0.5852815399994142 seconds with 1 matches
    For DNA4.txt and pattern ACT, the finite automata algorithm took
   1.2557424819970038 seconds with 524454 matches
    For DNA4.txt and pattern GATTACA, the finite automata algorithm took
   1.202983769995626 seconds with 1976 matches
    For DNA4.txt and pattern CATCATCATCAT, the finite automata algorithm
   took 1.1666112150123809 seconds with 5 matches
    For DNA5.txt and pattern ACT, the finite automata algorithm took
   2.4493759730103193 seconds with 1050689 matches
    For DNA5.txt and pattern GATTACA, the finite automata algorithm took
   2.3914340260089375 seconds with 4161 matches
    For DNA5.txt and pattern CATCATCATCAT, the finite automata algorithm
   took 2.339489956997568 seconds with 3 matches
```

Listing 7: Terminal output running code

The finite automata algorithm performs faster than both the naive and Rabin-Karp algorithms, and also seems truly independent of the length of the pattern, peforming about the same or even quicker in the longer patterns.

Problem 4. Repeat the previous problem with the Knuth-Morris-Pratt(KMP) algorithm.

```
def kmp(text, pattern):
    matches = []
    m = len(pattern)
    n = len(text)
    pfxfun = getPrefixFunction(pattern)
    i = 0
    j = 0

    while i < n:
        if text[i] == pattern[j]:
            i += 1
            j += 1
            if j == m:
                matches.append(i-j)
                j = pfxfun[j-1]
        else:
            if j > 0:
                j = pfxfun[j-1]
            else:
                i += 1

    return matches

def getPrefixFunction(pattern):
    pfxfun = [0 for i in range(len(pattern))]
    i = 1
    j = 0
    while i < len(pattern):
        if pattern[i] == pattern[j]:
            j += 1
            pfxfun[i] = j
            i += 1
        else:
            if j != 0:
                j = pfxfun[j - 1]
            else:
                pfxfun[i] = 0
                i += 1
    return pfxfun
```

Listing 8: Knuth-Morris-Pratt algorithm

```
$ python kmp.py
For DNA1.txt and pattern ACT, the Knuth-Morris-Pratt algorithm took 0.2486665429896675 seconds with 65495 matches
For DNA1.txt and pattern GATTACA, the Knuth-Morris-Pratt algorithm took 0.2490224240027601 seconds with 242 matches
For DNA1.txt and pattern CATCATCATCAT, the Knuth-Morris-Pratt algorithm took 0.24674882799445186 seconds with 0 matches
For DNA2.txt and pattern ACT, the Knuth-Morris-Pratt algorithm took 0.4929636379965814 seconds with 131034 matches
For DNA2.txt and pattern GATTACA, the Knuth-Morris-Pratt algorithm took 0.4923475959949428 seconds with 491 matches
For DNA2.txt and pattern CATCATCATCAT, the Knuth-Morris-Pratt algorithm took 0.49851692000811454 seconds with 1 matches
For DNA3.txt and pattern ACT, the Knuth-Morris-Pratt algorithm took 0.9728445020009531 seconds with 262192 matches
For DNA3.txt and pattern GATTACA, the Knuth-Morris-Pratt algorithm took 0.7028228560084244 seconds with 1002 matches
For DNA3.txt and pattern CATCATCATCAT, the Knuth-Morris-Pratt algorithm took 0.704158250009641 seconds with 1 matches
For DNA4.txt and pattern ACT, the Knuth-Morris-Pratt algorithm took 1.40684947000409 seconds with 524454 matches
For DNA4.txt and pattern GATTACA, the Knuth-Morris-Pratt algorithm took 1.424388044004445 seconds with 1976 matches
For DNA4.txt and pattern CATCATCATCAT, the Knuth-Morris-Pratt algorithm took 1.4213517800089903 seconds with 5 matches
For DNA5.txt and pattern ACT, the Knuth-Morris-Pratt algorithm took 2.793876569994609 seconds with 1050689 matches
For DNA5.txt and pattern GATTACA, the Knuth-Morris-Pratt algorithm took 2.8016387920069974 seconds with 4161 matches
For DNA5.txt and pattern CATCATCATCAT, the Knuth-Morris-Pratt algorithm took 2.790721905999817 seconds with 3 matches
```

Listing 9: Terminal output running code

The Knuth-Morris-Pratt algorithm performs faster than both the naive and Rabin-Karp algorithms but slower than the finite automata. It also seems independent of the length of the pattern like finite automata. It might be slower than finite automata because it does more comparisons per character of the text.