Project 3

Mengxiang Jiang
CSEN 5336 Analysis of Algorithms

October 24, 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?

```python
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

```python
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

```
$ 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.

```
alpha = {'A': 0, 'T' : 1, 'G' : 2, 'C' : 3}
pmod = 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:
            texthash = rabin_hash(text[0:len(pattern)])
        else:
            texthash = roll(plen, texthash,
                            text[i - 1], text[i + plen - 1])
        if texthash == 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(alpha) % pmod + alpha[c]) % pmod
    return rhash

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

Listing 4: Rabin-Karp algorithm

```
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 = rabin_karp(text, pattern)
            end_time = time.perf_counter()
            print(f'For {fname} and pattern {pattern}, the Rabin-Karp
    algorithm took {end_time-start_time} seconds with {len(matches)}
    matches')
```

Listing 5: main function code

```
$ python rabinkarp.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 6: 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.

```
alpha = {'A': 0, 'T' : 1, 'G' : 2, 'C' : 3}
pmod = 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:
            texthash = rabin_hash(text[0:len(pattern)])
        else:
            texthash = roll(plen, texthash,
                            text[i - 1], text[i + plen - 1])
        if texthash == 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(alpha) % pmod + alpha[c]) % pmod
    return rhash

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

Listing 7: Rabin-Karp algorithm

```
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 = rabin_karp(text, pattern)
            end_time = time.perf_counter()
            print(f'For {fname} and pattern {pattern}, the Rabin-Karp
    algorithm took {end_time-start_time} seconds with {len(matches)}
    matches')
```

Listing 8: main function code

```
$ python rabinkarp.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 9: 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.