**Build a class of Boyer-Moore algorithm to index the pattern**

In [1]:

*#!/usr/bin/env python*

*"""bm\_preproc.py: Boyer-Moore preprocessing."""*

\_\_author\_\_ = "Ben Langmead"

**import** **unittest**

**def** z\_array(s):

*""" Use Z algorithm (Gusfield theorem 1.4.1) to preprocess s """*

**assert** len(s) > 1

z = [len(s)] + [0] \* (len(s)-1)

*# Initial comparison of s[1:] with prefix*

**for** i **in** range(1, len(s)):

**if** s[i] == s[i-1]:

z[1] += 1

**else**:

**break**

r, l = 0, 0

**if** z[1] > 0:

r, l = z[1], 1

**for** k **in** range(2, len(s)):

**assert** z[k] == 0

**if** k > r:

*# Case 1*

**for** i **in** range(k, len(s)):

**if** s[i] == s[i-k]:

z[k] += 1

**else**:

**break**

r, l = k + z[k] - 1, k

**else**:

*# Case 2*

*# Calculate length of beta*

nbeta = r - k + 1

zkp = z[k - l]

**if** nbeta > zkp:

*# Case 2a: zkp wins*

z[k] = zkp

**else**:

*# Case 2b: Compare characters just past r*

nmatch = 0

**for** i **in** range(r+1, len(s)):

**if** s[i] == s[i - k]:

nmatch += 1

**else**:

**break**

l, r = k, r + nmatch

z[k] = r - k + 1

**return** z

**def** n\_array(s):

*""" Compile the N array (Gusfield theorem 2.2.2) from the Z array """*

**return** z\_array(s[::-1])[::-1]

**def** big\_l\_prime\_array(p, n):

*""" Compile L' array (Gusfield theorem 2.2.2) using p and N array.*

*L'[i] = largest index j less than n such that N[j] = |P[i:]| """*

lp = [0] \* len(p)

**for** j **in** range(len(p)-1):

i = len(p) - n[j]

**if** i < len(p):

lp[i] = j + 1

**return** lp

**def** big\_l\_array(p, lp):

*""" Compile L array (Gusfield theorem 2.2.2) using p and L' array.*

*L[i] = largest index j less than n such that N[j] >= |P[i:]| """*

l = [0] \* len(p)

l[1] = lp[1]

**for** i **in** range(2, len(p)):

l[i] = max(l[i-1], lp[i])

**return** l

**def** small\_l\_prime\_array(n):

*""" Compile lp' array (Gusfield theorem 2.2.4) using N array. """*

small\_lp = [0] \* len(n)

**for** i **in** range(len(n)):

**if** n[i] == i+1: *# prefix matching a suffix*

small\_lp[len(n)-i-1] = i+1

**for** i **in** range(len(n)-2, -1, -1): *# "smear" them out to the left*

**if** small\_lp[i] == 0:

small\_lp[i] = small\_lp[i+1]

**return** small\_lp

**def** good\_suffix\_table(p):

*""" Return tables needed to apply good suffix rule. """*

n = n\_array(p)

lp = big\_l\_prime\_array(p, n)

**return** lp, big\_l\_array(p, lp), small\_l\_prime\_array(n)

**def** good\_suffix\_mismatch(i, big\_l\_prime, small\_l\_prime):

*""" Given a mismatch at offset i, and given L/L' and l' arrays,*

*return amount to shift as determined by good suffix rule. """*

length = len(big\_l\_prime)

**assert** i < length

**if** i == length - 1:

**return** 0

i += 1 *# i points to leftmost matching position of P*

**if** big\_l\_prime[i] > 0:

**return** length - big\_l\_prime[i]

**return** length - small\_l\_prime[i]

**def** good\_suffix\_match(small\_l\_prime):

*""" Given a full match of P to T, return amount to shift as*

*determined by good suffix rule. """*

**return** len(small\_l\_prime) - small\_l\_prime[1]

**def** dense\_bad\_char\_tab(p, amap):

*""" Given pattern string and list with ordered alphabet characters, create*

*and return a dense bad character table. Table is indexed by offset*

*then by character. """*

tab = []

nxt = [0] \* len(amap)

**for** i **in** range(0, len(p)):

c = p[i]

**assert** c **in** amap

tab.append(nxt[:])

nxt[amap[c]] = i+1

**return** tab

**class** **BoyerMoore**(object):

*""" Encapsulates pattern and associated Boyer-Moore preprocessing. """*

**def** \_\_init\_\_(self, p, alphabet='ACGT'):

*# Create map from alphabet characters to integers*

self.amap = {alphabet[i]: i **for** i **in** range(len(alphabet))}

*# Make bad character rule table*

self.bad\_char = dense\_bad\_char\_tab(p, self.amap)

*# Create good suffix rule table*

\_, self.big\_l, self.small\_l\_prime = good\_suffix\_table(p)

**def** bad\_character\_rule(self, i, c):

*""" Return # skips given by bad character rule at offset i """*

**assert** c **in** self.amap

**assert** i < len(self.bad\_char)

ci = self.amap[c]

**return** i - (self.bad\_char[i][ci]-1)

**def** good\_suffix\_rule(self, i):

*""" Given a mismatch at offset i, return amount to shift*

*as determined by (weak) good suffix rule. """*

length = len(self.big\_l)

**assert** i < length

**if** i == length - 1:

**return** 0

i += 1 *# i points to leftmost matching position of P*

**if** self.big\_l[i] > 0:

**return** length - self.big\_l[i]

**return** length - self.small\_l\_prime[i]

**def** match\_skip(self):

*""" Return amount to shift in case where P matches T """*

**return** len(self.small\_l\_prime) - self.small\_l\_prime[1]

**class** **TestBoyerMoorePreproc**(unittest.TestCase):

**def** test\_z\_1(self):

s = 'abb'

*# -00*

z = z\_array(s)

self.assertEqual([3, 0, 0], z)

**def** test\_z\_2(self):

s = 'abababab'

*# 00604020*

z = z\_array(s)

self.assertEqual([8, 0, 6, 0, 4, 0, 2, 0], z)

**def** test\_z\_3(self):

s = 'abababab'

*# 00604020*

z = z\_array(s)

self.assertEqual([8, 0, 6, 0, 4, 0, 2, 0], z)

**def** test\_n\_1(self):

s = 'abb'

*# 01-*

n = n\_array(s)

self.assertEqual([0, 1, 3], n)

**def** test\_n\_2(self):

s = 'abracadabra'

*# 1004010100-*

n = n\_array(s)

self.assertEqual([1, 0, 0, 4, 0, 1, 0, 1, 0, 0, 11], n)

**def** test\_n\_3(self):

s = 'abababab'

*# 0204060-*

n = n\_array(s)

self.assertEqual([0, 2, 0, 4, 0, 6, 0, 8], n)

**def** test\_big\_l\_prime\_1(self):

s = 'abb'

*# 001*

big\_l\_prime = big\_l\_prime\_array(s, n\_array(s))

self.assertEqual([0, 0, 2], big\_l\_prime)

**def** test\_big\_l\_prime\_2(self):

s = 'abracadabra'

*# 01234567890*

*# L' 00000003007*

*# L 00000003337*

big\_l\_prime = big\_l\_prime\_array(s, n\_array(s))

self.assertEqual([0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 8], big\_l\_prime)

**def** test\_small\_l\_prime\_1(self):

s = 'abracadabra'

*# N 1004010100-*

*# l' 1*

*# l' 4*

*# l' 44444444111*

small\_l\_prime = small\_l\_prime\_array(n\_array(s))

self.assertEqual([11, 4, 4, 4, 4, 4, 4, 4, 1, 1, 1], small\_l\_prime)

**def** test\_good\_suffix\_match\_mismatch\_1(self):

p = 'GGTAGGT'

big\_l\_prime, big\_l, small\_l\_prime = good\_suffix\_table(p)

self.assertEqual([0, 0, 0, 0, 3, 0, 0], big\_l\_prime)

self.assertEqual([0, 0, 0, 0, 3, 3, 3], big\_l)

self.assertEqual([7, 3, 3, 3, 3, 0, 0], small\_l\_prime)

self.assertEqual(0, good\_suffix\_mismatch(6, big\_l\_prime, small\_l\_prime))

self.assertEqual(0, good\_suffix\_mismatch(6, big\_l, small\_l\_prime))

*# t: xT*

*# p: GGTAGGT*

*# L': -000300*

*# L: -000333*

self.assertEqual(7, good\_suffix\_mismatch(5, big\_l\_prime, small\_l\_prime))

self.assertEqual(4, good\_suffix\_mismatch(5, big\_l, small\_l\_prime))

*# t: xGT*

*# p: GGTAGGT*

*# L': -000300*

*# L: -000333*

self.assertEqual(7, good\_suffix\_mismatch(4, big\_l\_prime, small\_l\_prime))

self.assertEqual(4, good\_suffix\_mismatch(4, big\_l, small\_l\_prime))

*# t: xGGT*

*# p: GGTAGGT*

*# L': -000300*

*# L: -000333*

self.assertEqual(4, good\_suffix\_mismatch(3, big\_l\_prime, small\_l\_prime))

self.assertEqual(4, good\_suffix\_mismatch(3, big\_l, small\_l\_prime))

*# t: xAGGT*

*# p: GGTAGGT*

*# L': -000300*

*# L: -000333*

self.assertEqual(4, good\_suffix\_mismatch(2, big\_l\_prime, small\_l\_prime))

self.assertEqual(4, good\_suffix\_mismatch(2, big\_l, small\_l\_prime))

*# t: xTAGGT*

*# p: GGTAGGT*

*# L': -000300*

*# L: -000333*

self.assertEqual(4, good\_suffix\_mismatch(1, big\_l\_prime, small\_l\_prime))

self.assertEqual(4, good\_suffix\_mismatch(1, big\_l, small\_l\_prime))

*# t: xGTAGGT*

*# p: GGTAGGT*

*# L': -000300*

*# L: -000333*

self.assertEqual(4, good\_suffix\_mismatch(0, big\_l\_prime, small\_l\_prime))

self.assertEqual(4, good\_suffix\_mismatch(0, big\_l, small\_l\_prime))

**def** test\_good\_suffix\_table\_1(self):

s = 'abb'

*# 001*

big\_l\_prime, big\_l, small\_l\_prime = good\_suffix\_table(s)

self.assertEqual([0, 0, 2], big\_l\_prime)

self.assertEqual([0, 0, 2], big\_l)

self.assertEqual([3, 0, 0], small\_l\_prime)

**def** test\_good\_suffix\_table\_2(self):

s = 'abracadabra'

*# 01234567890*

*# L' 00000003007*

*# L 00000003337*

*# l' -4444444111*

big\_l\_prime, big\_l, small\_l\_prime = good\_suffix\_table(s)

self.assertEqual([0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 8], big\_l\_prime)

self.assertEqual([0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 8], big\_l)

self.assertEqual([11, 4, 4, 4, 4, 4, 4, 4, 1, 1, 1], small\_l\_prime)

**Boyer-Moore Function**

In [2]:

**def** boyer\_moore(p, p\_bm, t):

*""" Do Boyer-Moore matching. p=pattern, t=text,*

*p\_bm=BoyerMoore object for p """*

i = 0

occurrences = []

**while** i < len(t) - len(p) + 1:

shift = 1

mismatched = **False**

**for** j **in** range(len(p)-1, -1, -1):

**if** p[j] != t[i+j]:

skip\_bc = p\_bm.bad\_character\_rule(j, t[i+j])

skip\_gs = p\_bm.good\_suffix\_rule(j)

shift = max(shift, skip\_bc, skip\_gs)

mismatched = **True**

**break**

**if** **not** mismatched:

occurrences.append(i)

skip\_gs = p\_bm.match\_skip()

shift = max(shift, skip\_gs)

i += shift

**return** occurrences

**Function of FASTA File Loading**

In [3]:

**def** loadFasta(filename):

f = open(filename)

name = ''

dna = ''

**for** line **in** f:

**if** line[0] == '>':

name = line[1:].rstrip()

**else**:

dna += line.rstrip()

**return** name, dna

**Load the FASTA File**

In [4]:

file = 'chr1.GRCh38.excerpt.fasta'

chromosome\_name, chromosome\_dna = loadFasta(file)

**Question 1**

How many alignments does the naive exact matching algorithm try when matching the string**GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG** (derived from human Alu sequences) to the excerpt of human chromosome 1? (Don't consider reverse complements.) Answer for Question 1 You entered:

In [5]:

**def** naive\_alignment(p, t):

appearance = []

alignment = 0

**for** i **in** range(len(t) - len(p) + 1):

alignment += 1

**for** j **in** range(len(p)):

match = **True**

**if** t[i + j] != p[j]:

match = **False**

**break**

**if** match == **True**:

appearance.append(i)

**return** appearance, alignment

In [6]:

alu = 'GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG'

In [7]:

naive\_alignment(alu, chromosome\_dna)

Out[7]:

([56922], 799954)

**Question 2**

How many character comparisons does the naive exact matching algorithm try when matching the string GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG (derived from human Alu sequences) to the excerpt of human chromosome 1? (Don't consider reverse complements.)

In [8]:

**def** naive\_comparison(p, t):

appearance = []

comparison = 0

**for** i **in** range(len(t) - len(p) + 1):

**for** j **in** range(len(p)):

comparison += 1

match = **True**

**if** t[i + j] != p[j]:

match = **False**

**break**

**if** match == **True**:

appearance.append(i)

**return** appearance, comparison

In [9]:

naive\_comparison(alu, chromosome\_dna)

Out[9]:

([56922], 984143)

**Question 3**

How many alignments does Boyer-Moore try when matching the string**GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG** (derived from human Alu sequences) to the excerpt of human chromosome 1? (Don't consider reverse complements.)

In [11]:

**def** boyer\_moore(p, p\_bm, t):

*""" Do Boyer-Moore matching. p=pattern, t=text,*

*p\_bm=BoyerMoore object for p """*

i = 0

occurrences = []

alignment = 0

**while** i < len(t) - len(p) + 1:

alignment +=1

shift = 1

mismatched = **False**

**for** j **in** range(len(p)-1, -1, -1):

**if** p[j] != t[i+j]:

skip\_bc = p\_bm.bad\_character\_rule(j, t[i+j])

skip\_gs = p\_bm.good\_suffix\_rule(j)

shift = max(shift, skip\_bc, skip\_gs)

mismatched = **True**

**break**

**if** **not** mismatched:

occurrences.append(i)

skip\_gs = p\_bm.match\_skip()

shift = max(shift, skip\_gs)

i += shift

**return** occurrences, alignment

In [12]:

alu\_bm = BoyerMoore(alu)

boyer\_moore(alu, alu\_bm, chromosome\_dna)

Out[12]:

([56922], 127974)

**Answer:**

127974

**Question 4**

Index-assisted approximate matching. In practicals, we built a Python class called Index implementing an ordered-list version of the k-mer index. The Index class is copied below.

In [13]:

\_\_author\_\_ = "Ben Langmead"

**import** **bisect**

**class** **Index**(object):

*""" Holds a substring index for a text T """*

**def** \_\_init\_\_(self, t, k):

*""" Create index from all substrings of t of length k """*

self.k = k *# k-mer length (k)*

self.index = []

**for** i **in** range(len(t) - k + 1): *# for each k-mer*

self.index.append((t[i:i+k], i)) *# add (k-mer, offset) pair*

self.index.sort() *# alphabetize by k-mer !!!!!! Attention, remember this sorting step!!!!*

**def** query(self, p):

*""" Return index hits for first k-mer of p """*

kmer = p[:self.k] *# query with first k-mer*

i = bisect.bisect\_left(self.index, (kmer, -1)) *# binary search*

hits = []

**while** i < len(self.index): *# collect matching index entries*

**if** self.index[i][0] != kmer:

**break**

hits.append(self.index[i][1])

i += 1

**return** hits

In [14]:

**def** index\_approximate\_match(p, t, t\_index, n):

segment\_length = int(len(p) / (n + 1))

hits = []

all\_match =set() *# We use set to unique the repetitive items.*

**for** c **in** range(n + 1):

start = c \* segment\_length

end = (c + 1) \* segment\_length

segment = p[start:end]

i = bisect.bisect\_left(t\_index.index, (segment, -1))

**while** i < len(t\_index.index):

**if** segment != t\_index.index[i][0]:

**break**

**else**:

t\_offset = t\_index.index[i][1]

hits.append(t\_offset)

i += 1

**for** h **in** hits:

**if** start > h **or** h-start+len(p) > len(t):

**continue**

mismatch = 0

**for** x **in** range(0, start):

**if** p[x] != t[h-start+x]:

mismatch += 1

**if** mismatch > n:

**break**

**for** y **in** range(end, len(p)):

**if** p[y] != t[h-start+y]:

mismatch += 1

**if** mismatch > n:

**break**

**if** mismatch <=n:

all\_match.add(h-start) *# We use set to unique the repetitive items.*

**return** hits, list(all\_match)

In [15]:

chromosome\_dna\_index = Index(chromosome\_dna, 8)

We also implemented the pigeonhole principle using Boyer-Moore as our exact matching algorithm.

Implement the pigeonhole principle using Index to find exact matches for the partitions. Assume P always has length 24, and that we are looking for approximate matches with up to 2 mismatches (substitutions). We will use an 8-mer index.

Download the Python module for building a k-mer index.

<https://d28rh4a8wq0iu5.cloudfront.net/ads1/code/kmer_index.py>

Write a function that, given a length-24 pattern P and given an Index object built on 8-mers, finds all approximate occurrences of P within T with up to 2 mismatches. Insertions and deletions are not allowed. Don't consider any reverse complements.

How many times does the string GGCGCGGTGGCTCACGCCTGTAAT, which is derived from a human Alu sequence, occur with up to 2 substitutions in the excerpt of human chromosome 1? (Don't consider reverse complements here.)

Hint 1: Multiple index hits might direct you to the same match multiple times, but be careful not to count a match more than once.

Hint 2: You can check your work by comparing the output of your new function to that of the naive\_2mm function implemented in the previous module.

In [16]:

p = 'GGCGCGGTGGCTCACGCCTGTAAT'

hits, matches = index\_approximate\_match(p, chromosome\_dna, chromosome\_dna\_index, 2)

In [17]:

len(matches)

Out[17]:

19

**Answer:**

19

**Question 5**

Using the instructions given in Question 4, how many total index hits are there when searching for occurrences of**GGCGCGGTGGCTCACGCCTGTAAT** with up to 2 substitutions in the excerpt of human chromosome 1?

(Don't consider reverse complements.) Hint: You should be able to use the boyer\_moore function (or the slower naive function) to double-check your answer.

In [18]:

len(hits)

Out[18]:

90

**Answer:**

90

**Question 6**

Let's examine whether there is a benefit to using an index built using subsequences of T rather than substrings, as we discussed in the "Variations on k-mer indexes" video. We'll consider subsequences involving every N characters. For example, if we split ATATAT into two substring partitions, we would get partitions ATA (the first half) and TAT (second half). But if we split ATATAT into two subsequences by taking every other character, we would get AAA (first, third and fifth characters) and TTT (second, fourth and sixth).

Another way to visualize this is using numbers to show how each character of P is allocated to a partition. Splitting a length-6 pattern into two substrings could be represented as 111222, and splitting into two subsequences of every other character could be represented as 121212

The following class SubseqIndex is a more general implementation of Index that additionally handles subsequences. It only considers subsequences that take every Nth character:

In [19]:

**class** **SubseqIndex**(object):

*""" Holds a subsequence index for a text T """*

**def** \_\_init\_\_(self, t, k, ival):

*""" Create index from all subsequences consisting of k characters*

*spaced ival positions apart. E.g., SubseqIndex("ATAT", 2, 2)*

*extracts ("AA", 0) and ("TT", 1). """*

self.k = k *# num characters per subsequence extracted*

self.ival = ival *# space between them; 1=adjacent, 2=every other, etc*

self.index = []

self.span = 1 + ival \* (k - 1)

**for** i **in** range(len(t) - self.span + 1): *# for each subseq*

self.index.append((t[i:i+self.span:ival], i)) *# add (subseq, offset)*

self.index.sort() *# alphabetize by subseq*

**def** query(self, p):

*""" Return index hits for first subseq of p """*

subseq = p[:self.span:self.ival] *# query with first subseq*

i = bisect.bisect\_left(self.index, (subseq, -1)) *# binary search*

hits = []

**while** i < len(self.index): *# collect matching index entries*

**if** self.index[i][0] != subseq:

**break**

hits.append(self.index[i][1])

i += 1

**return** hits

Write a function that, given a length-24 pattern P and given a SubseqIndex object built with k = 8 and ival = 3, finds all approximate occurrences of P within T with up to 2 mismatches.

When using this function, how many total index hits are there when searching for GGCGCGGTGGCTCACGCCTGTAAT with up to 2 substitutions in the excerpt of human chromosome 1? (Again, don't consider reverse complements.)

Hint: See this notebook for a few examples you can use to test your function.

In [20]:

t\_subseq\_index = SubseqIndex(chromosome\_dna, 8, 3)

len(t\_subseq\_index.query(p))

Out[20]:

35

In [27]:

**def** index\_approximate\_subseq\_match(p, t, t\_subseq\_index, n):

hits = []

**for** position **in** range(3):

subseq = p[position:][:22:3]

i = bisect.bisect\_left(t\_subseq\_index.index, (subseq, -1)) *# binary search*

**while** i < len(t\_subseq\_index.index): *# collect matching index entries*

**if** t\_subseq\_index.index[i][0] != subseq:

**break**

hits.append(t\_subseq\_index.index[i][1])

i += 1

**return** hits

In [28]:

hits = index\_approximate\_subseq\_match(p, chromosome\_dna, t\_subseq\_index, 2)

GGGGCGTA

GCTCACGA

CGGTCCTT

In [29]:

len(hits)

Out[29]:

79