Congratulations! You passed!

TO PASS 70% or higher

Keep Learning

Retake the assignment in **7h 51m**

GRADE
100%

Programming Homework 2

LATEST SUBMISSION GRADE

100%

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.)

1 / 1 point

Correct

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.)

1 / 1 point

Correct

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.)

1 / 1 point

Correct

4. **Index-assisted approximate matching.** In practicals, we built a Python class called `Index`

1 / 1 point

implementing an ordered-list version of the k-mer index. The `Index` class is copied below.

```
class Index(object):
    def __init__(self, t, k):
        ''' Create index from all substrings of size 'length' '''
        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

    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
```

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.

Correct

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?

1 / 1 point

(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.

✓ **Correct**

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).

1 / 1 point

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:

```
import bisect

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

```

For example, if we do:

```
ind = SubseqIndex('ATATAT', 3, 2)
print(ind.index)

```

we see:

```
[('AAA', 0), ('TTT', 1)]

```

And if we query this index:

```
p = 'TTATAT'
print(ind.query(p[0:]))

```

we see:

```
[]

```

because the subsequence `TAA` is not in the index. But if we query with the second subsequence:

```
print(ind.query(p[1:]))

```

we see:

```
[1]
```

because the second subsequence `TTT` is in the index.

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.

✓ **Correct**