EN 601.447/647: Computational Genomics

Homework 4

Fall 2021, Ben Langmead, TA: Zitong He, CA: Robert Ye

The homework is out of 38 points (20 programming, 18 written). Your submission must be your work alone. You may reuse code from the lecture notes and associated Jupyter notebooks. You should submit your assignment as two separate submissions: Programming and Written. Submit written answers as a PDF, starting each new response on a new page. Use the stated file-name conventions. The homework is due by 10/28/2021, 11:59PM but Gradescope will allow late submissions until 10/31/2021, 11:59PM. See syllabus for late-day and other grading rules.

Please note that passing the visible Gradescope tests does not indicate you will receive full credit for a programming solution. The visible tests are small and simple, useful mainly to ensure you are not far off. It is up to you to ensure you've implemented the solution correctly as requested, and we encourage you to create and use more test cases than the ones provided.

Programming questions – submit to "HW4 Programming" in Gradescope

1. (8 pts) Write a program with name prefix `hw4q1`. The program will take three command-line arguments:

   (a) The name of a FASTQ file containing sequencing reads

   (b) A integer K, corresponding to the minimum suffix/prefix match length

   (c) The name of a output file containing information described below

   The program will then find **best unambiguous match to the right** relationships for the reads in the FASTQ file. To find these relationships, we consider each read. For a read A:

   - Look for a different read B such that a suffix of A matches a prefix of B and:
     (a) The length of the suffix/prefix match is the longest among all choices for B, and
     (b) The length of the match is at least K, and
     (c) There is **no tie**; i.e. no other read C has as long a matching prefix as B.
   - If such a read B is found, your program should print a single, space-separated line with the ID of A ,the length of the overlap, and ID of B, like this:

     ```
     0255/2 88 2065/1
     ```

     This indicates that a 88 nt suffix of the read with ID 0255/2 is also a prefix of the read with ID 2065/1, and that no other read besides 2065/1 has a prefix of 88 nt (or longer) that is also a suffix of read 0255/2.

   You may use the suffix/prefix matching idea we discussed in class, implemented in this Python function. It takes two strings, str1 and str2. It looks for the longest prefix of str2 of length at least `min_overlap` that is also a suffix of str1 and returns the length of the suffix/prefix match. If no such match is found, it returns 0.

```
def suffix_prefix_match(str1, str2, min_overlap):
    if len(str2) < min_overlap:
        return 0
    str2_prefix = str2[:min_overlap]
    str1_pos = -1
    while True:
        str1_pos = str1.find(str2_prefix, str1_pos + 1)
        if str1_pos == -1:
            return 0
        str1_suffix = str1[str1_pos:]
        if str2.startswith(str1_suffix):
            return len(str1_suffix)
```

Note:

- The results should be output in the same order as the input reads. That is, if read ID1 comes before read ID2 in the input, then any record having ID1 in the first column should come before any record having ID2 in the first column.
- You may assume all reads in a FASTQ are the same length. In other words, if the first read is N bases long, then all the reads in the file will be N bases long.
- Because of the "no tie" requirement, a read ID should appear in the leftmost column at most once in your output. That means if a read's longest match has more than one read, then discard them all.

For $K = 3$ and this input:

```
@r1
AAACCCGGG
+
IIIIIIIII
@r2
CCCGGGTTT
+
IIIIIIIII
@r3
CCCGGGAAA
+
IIIIIIIII
```

The output should be:

```
r3 3 r1
```

For full credit, your solution must be able to run on the Autograder without timing out or running out of memory. An example we will use when testing your code is available here:

`http://www.cs.jhu.edu/~langmea/resources/hw4_reads.fastq`

Hint: Running your program on these reads with $K = 85$ should yield 4,998 output records.

Note on efficiency: If you simply call the `suffixPrefixMatch` function on every ordered pair of reads in this `hw4_reads.fastq` dataset, you will probably find yourself waiting for many seconds and perhaps even minutes for an answer. This could cause the Gradescope timeout to activate, causing you to fail Gradescope tests. Besides that, it's inconvenient to have to wait that long to check your solution.

We recommend that you use a simple filtration strategy to reduce the number of read pairs to check for suffix/prefix matches. The idea is that if two reads do not share a substring of length K, they can't possibly have a relevant suffix/prefix match and you don't need to check them. Here's a function that groups reads according to whether they share a length-K substring. **You do not have to use this code or this exact idea**, but you may if you like:

```
def make_kmer_table(seqs, k):
    """ Given read dictionary and integer k, return a dictionary that
        maps each k-mer to the set of names of reads containing the k-mer. """
    table = {}
    for name, seq in seqs.items():
        for i in range(0, len(seq) - k + 1):
            kmer = seq[i:i+k]
            if kmer not in table:
                table[kmer] = set()
            table[kmer].add(name)
    return table
```

2. (8 pts) Write a program with name prefix `hw4q2`. Your program will take two command-line arguments:

   (a) The name of a file containing records of the kind your `hw4q1` program output

   (b) The name of a output file containing information described below

   Your program will use this "best unambiguous match to the right" information to build "unitigs," **uni**quely assemble-able con**tigs**. To construct unitigs, you will need to know the best "best unambiguous match to the right" and the "best unambiguous match to the left" for each read, if they exist. Since the output from your solution to `hw4q1` only gives the best match to the right, you will have to *infer* each read's best match to the left. Consider the following output:

   ```
   A 60 B
   E 40 A
   C 70 B
   D 40 C
   ```

   A's best match to the right (BMR) is B. But B's best match to the left (BML) is C, not A! So A and B are not mutual best unambiguous matches. Your program should form unitigs by joining together two reads X and Y if they are mutual best unambiguous matches. X and Y are mutual best matches if X's BMR is Y and Y's BML is X (or the same, but with X and Y switched). In this example, we would join D, C, and B into a single unitig (and in that order), and would join reads E and A into a single unitig (also in that order).

Your program should output the unitigs found. To report a unitig, first print the ID of the leftmost read in the unitig on its own line. For each subsequent read in the unitig, moving left-to-right, print a line consisting of the length of the prefix of the read that matched a suffix of the previous one, then a space, then the read's ID.

```
D
40 C
70 B
E
40 A
```

The first three lines encode a unitig of 3 reads. The first (leftmost) read is D. The second read is C, which has a 40-base prefix that is a suffix of D. The third (and final) read is B, which has a 70-base prefix that is a suffix of C.

Unitigs should be reported in **alphabetical order** according to the ID of the unitig's leftmost read; in this example, the unitig with D as the leftmost read comes before the unitig with E as the leftmost read.

For full credit, your solution must be able to run on the Autograder without timing out or running out of memory. An example we will use when testing your code will be the output from our solution to `hw4q1` when we run it on the `hw4_reads.fastq` input with $K = 85$.

Hint: Running your program on the output from `hw4q1` when run on the `hw4_reads.fastq` reads file with $K = 85$ should yield 4 unitigs.

3. (4 pts) Write a program with name prefix `hw4q3`. The program will take three command-line arguments:

   (a) The name of a FASTQ file containing sequencing reads

   (b) The name of a file containing the unitigs output by your solution to `hw4q2`

   (c) The name of a output file containing information described below

   Your program should use the unitigs and the sequences of the reads in the FASTQ file to construct the actual full sequence of each unitig. Your program should output the unitigs in FASTA format (one FASTA record per unitig) with line breaks placed so that no line has more than 60 characters. On the name line of the FASTA record (the line starting with `>`) you should print the ID of the leftmost read in the unitig, then a space, then the number of reads in this unitig. So if the unitig consists of reads with IDs D, C, and B in that order, the name line should be `>D 3`.

   You can use the following Python code to help write the FASTA with line breaks:

```
def write_solution(unitigID, unitigSequence, n, out_fh, per_line=60):
    offset = 0
    out_fh.write(f">{unitigID} {n}\n")
    while offset < len(unitigSequence):
        line = unitigSequence[offset:offset + per_line]
        offset += per_line
        out_fh.write(line + "\n")
```

For $K = 30$ and this FASTQ input:

```
@unitig1_first
AAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@unitig1_second
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCGAAAAAAAAAAAAAAAAAAAAAAAAAAAG
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@unitig2_first
GGGGGGGGGGGGGGGGGGGGGGGGGGGGTTTTTTTTTTTTTTTTTTTTTTTTTT
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@unitig2_second
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTAGGGGGGGGGGGGGGGGGGGGGGGGGGGA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
```

And this unitig input (as written by your `hw4q2` solution):

```
unitig1_first
30 unitig1_second
unitig2_first
30 unitig2_second
```

The output should be:

```
>unitig1_first 2
AAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
GAAAAAAAAAAAAAAAAAAAAAAAAAAG
>unitig2_first 2
GGGGGGGGGGGGGGGGGGGGGGGGGGGGTTTTTTTTTTTTTTTTTTTTTTTTTT
AGGGGGGGGGGGGGGGGGGGGGGGGGGA
```

Hint: Running your program on the output from `hw4q1` and `hw4q2` when starting with the `hw4_reads.fastq` file and $K = 85$ should yield 4 unitigs, the longest of which has 3,127 bases and the shortest of which has 252 bases.

(If you can complete these questions, you built a simple genome assembler using the "best buddy" principle and unitigs!)

Written questions – submit to "HW4 Written " in Gradescope

4. As described in Gusfield section 11.6.5, we can use dynamic programming alignment to find approximate occurrences of a pattern P ($|P| = n$) in a text T ($|T| = m$). Gusfield calls this the "Approximate occurrences of P in T" problem, which is distinct from both the global and local alignment problems. While you could use either an edit-distance like penalty scheme (all edits cost 1) or a more general global-alignment penalty scheme, the penalties are not relevant to the question

   (a) (1 pt) Exactly how many dynamic programming matrix elements do we have to fill in to solve this problem? Don't forget to account for the empty prefixes.

   (b) (2 pt) How do we initialize the first row and column?

   (c) (3 pts) Say we were interested only in the best approximate occurrence (the one with the fewest mismatches and gaps) of P in T. Describe briefly how we would find it given the filled-in matrix, and how we would find where the mismatches and gaps in the alignment are? Assume there is no tie.

5. (a) (3 pts) Fill in this local alignment matrix. Rows are labeled with characters from X and columns are labeled with characters from Y. Use a scoring function where matches get a bonus of +1, and mismatches, insertions and deletions get a penalty of -1.

	ϵ	G	A	T	C	G
ϵ						
G						
A						
T						
A						
C						
C						

   (b) (3 pts) Report the pair(s) of substrings of X of Y with maximum global alignment value, along with their global alignment value. If there is a tie, report all pairs with maximal global alignment value.

6. (3 pts) Another way of thinking about the assembly problem is as it relates to finding a Hamiltonian Path, which is NP complete. Say we have an overlap graph that has a Hamiltonian path: a path that visits *each node exactly once*. Does the path necessarily visit each *edge* exactly once? If so, argue why. If not, show a small example where it doesn't.

7. (3 pts) Below is a filled-in global alignment matrix. The penalty function has one penalty for transitions (when A mismatches G or vice versa, or when C mismatches T or vice versa), a different penalty for transversions (all substitutions that aren't transitions), and a linear gap penalty. What were the transition, transversion, and gap penalties used?

	ϵ	T	T	C	G	A	A
ϵ	0	4	8	12	16	20	24
G	4	2	6	10	12	16	20
G	8	6	4	8	10	14	18
G	12	10	8	6	8	12	16
A	16	14	12	10	9	8	12
C	20	18	16	12	12	11	10
T	24	20	18	16	14	14	13