UC Berkeley, CS 176: Algorithms for Computational Biology (Fall 2017)

RNA-Sequencing Read-Alignment Project

Instructor: Nir Yosef

1 Introduction

In this project you will first implement some basic bowtie functions that we covered in class, and then implement a simplified version of an aligner for RNA sequencing reads. This assignment is due on **Monday 11/13 11:59pm PST**.

2 Rules and Tips

- **Your submission must keep all required (skeleton) functions / classes / methods in** *project.py*. Put all the python files in *project.zip*; you can create and import other files for helper functions. Do not put your util files in subdirectories, just put them in the same directory structure as *project.py*.
- You may only use Python 3. You can only use libraries and modules that are provided by the default installation of Python, as well as numpy.
- Do not change the provided function signatures and return types. We will feed you the inputs and evaluate your outputs following the defined signature and return types.
- We suggest using *jupyter notebook* for loading in the data and developing / evaluating your alignment algorithm; it's nice for keeping your variables saved if your code crashes somewhere. Once you have a working solution then port it back to the Python file.

3 BWT Functions

Implement all the functions according to lectures 4 to 8.

3.1 Rules and Tips

- We expect 0-indexed solutions, and any intervals are [inclusive, exclusive).
- You will solely be graded on correctness on small or medium sized examples (probably length < 10000). Your function should not time out unless it *reallllly* takes too long to run.
- Assume an alphabet consisting of ['$', A', 'C', 'T', 'G'], where "$" will always only be the terminator.
- Assume the *s* parameter in all the function is already terminated by "$".
- For constructing the suffix array, we do not suggest implementing the KS algorithm unless you want to for fun and implement it so it have good enough performance. I implemented the KS last year and it was actually slow as hell (probably because of the python overhead). You can use naive sorting on prefixes instead of radix sort if you want. **Constructing the suffix array should be efficient** if you want to use it for the aligner part of the project (see performance requirement details about the aligner). Also, when you use your function for the aligner, it should not use a radix of over 100 for sorting, as we will be limiting your memory usage to something reasonable, not to mention that longer radixes take longer to generate as well.
- **Do not delete docstring for** *exact_suffix_matches* if you want a sanity check of your code (see Testing below).

3.2 Testing

We have provided a simple python doctest as sanity check for your BWT functions. You can run this by running "python -m doctest project.py" on the command line.

4 RNA Read Alignment

In this part of the project you will implement some method to align RNA reads to genome. You are given a reduced size genome of roughly 11 million bases with the location of genes, isoforms, and exons. You may use an alignment technique discussed in lecture 13 or come up with your own technique, as long as the performance isn't too bad.

4.1 Reads

Your reads are generated from the **transcriptome** of the given genome. The given genome sequence corresponds to the forward strand; assume the genes / isoforms / exons are all on the forward strand, and all reads match to the forward strand. **There is no insertions or deletions** (but there could be mismatches) in the reads (i.e. each position in the read corresponds to some position in the genome). In addition, the genes / isoforms / exons that some reads are generated from have been **hidden**, i.e. these "unknown" genes will not be passed to your *Aligner* class in *Aligner.__init__*. In addition, some reads could be randomly generated. You will not be penalized for not being able to align reads to the transcriptome if we are also not able to align these to the transcriptome. See Evaluation / Scoring for more details on what we wil evaluate you on. In any read generated from the genome (visible or unknown), there are at most **6** mismatches, so any alignment with more than 6 mismatches is invalid.

4.1.1 Tips

- **Only align to the forward strand**, i.e. match the read sequence to the genome sequence, there should be no need to reverse complement anything.
- Remember that there is no insertion or deletion, so do not output alignments with insertion or deletion.
- Ideally, you should first align the reads to the transcriptome then the genome.
- If you implement STAR, you can assume minimum and maximum intron sizes to be 20 and 10000. I'm using window sizes of 64000 bases from the anchor for my implementation, but you can use whatever works best.

4.2 *Aligner* Class

Implement the *Aligner* class in *project.py*. We will initialize your *Aligner* with genome information, call *Aligner.align* on a number of sequences to get your outputs, then evaluate the outputs.

4.2.1 Initialization / Constructor

In *Aligner.__init__*, you are given the genome sequence as well as *genes*, which is a list of *Gene* container objects that we have defined in *shared.py*. Each *Gene* contains a list of *Isoform* objects that correspond to the gene, and each *Isoform* contains a list of *Exon* objects. You should look at how these classes are defined in the file, but you cannot modify these classes.

4.2.2 Format of an Alignment

Since we specified that there is no insertion or deletion, an alignment of a read to the genome can be thought as k (usually k will be 1 or 2) separate ungapped alignments between the read and the genome (with some start index in the read and start index in the genome). Thus we expect you to specify your alignment as a python list of k tuples of ($\langle$read start index$\rangle$, $\langle$genome start index$\rangle$, $\langle$length$\rangle$). For example, an alignment of [(0, 2, 3), (3, 6, 10)] specifies that the 0th position of the read aligns to the 2nd position of the genome for 3 bases, and then the 3rd position of the read aligns to the 6th positon of the genome for 10 bases. If you can't find an alignment for a read, you may return an empty list [].

Warning: if the ranges of two consecutive alignment pieces overlap in the read, i.e. if $\langle$read start 1$\rangle + \langle$length 1$\rangle > \langle$read start 2$\rangle$, we will discard the second piece of your alignment and score you accordingly. We check for this case with the provided functions in *evaluation.py* (see Evaluation / Scoring).

4.3 Test Files

You are given files containing examples of what kind of genome and read sequences we might test you on. You don't technically have to do anything with this, but we **highly** recommend that you parse the files and try your algorithm on these examples since they will be representative of our evaluation set.

- *genome.fa* is a FASTA file with the genome sequence.

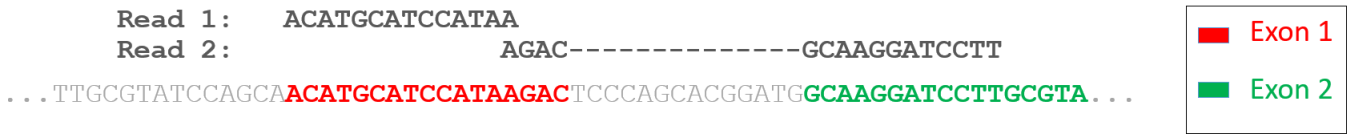
- *reads.fa* is a FASTA files with read sequences. Note that the file does not include base quality (PHRED) scores, as we have seen in the Bowtie1 algorithm. If you would like to implement Bowtie1, you can assume that the PHRED score is fixed for all bases.
- *genes.tab* is a tab-separated file containing three types of rows: a gene row begins with "gene" and specifies the *gene_id* then a semicolon-separated list of *isoform_id*; an isoform row begins with "isoform" and specifies the *isoform_id* then a sorted semicolon-separated list of *exon_id*; and an exon row begins with "exon" and specifies the *exon_id*, *start*, and *end* of the exon. Moreover, the genomic elements that are "unknown" (hidden when we test your *Aligner*) are prefixed with "unknown_". If a gene is unknown, its corresponding isoforms and exons will also be marked as unknown. **You should parse this file to construct your python set of genes that you feed into your** *Aligner.__init__*. Make sure to construct each *Gene* with all of its corresponding *Isoform* objects, and etc.

4.4 Evaluation / Scoring

4.4.1 Alignment Prioritization

Your should **always** prioritize aligning reads to known isoforms (in *genes.tab*) with 6 or less mismatches. Only if you can't align with 6 or less mismatches, you should try to align reads to other parts of the genome with 6 or less mismatches (as these regions may represent unknown genes). You should also minimize the number of mismatches, but do not choose an alignment to an unknown isoform with less mismatches over an alignment to a known isoform with more mismatches (but still 6 or less).

Priority 1: align read (with up to XXX mismatches) to known exons or known exon junctions



Priority 2: align read (with up to XXX mismatches) to unknown exons or unknown exon junctions

If you don't align some indices of a read, those indices will be counted as mismatches (so it's always in your best interest to align all indices of the read). Since there is no insertion or deletion, we will be scoring your alignment based on one-to-one and **consecutive** correspondence to the transcriptome. i.e. if you match a read to an isoform, make sure that the read aligns **consecutively** to the transcript generated by concatenating the exons within the isoform, there should be no gaps when aligning to the transcriptome.

4.5 *evaluation.py*

To make sure you are able to evaluate your aligner's alignments, we have provided the function that we will use to evaluate your alignments in *evaluation.py*. You first need to construct a python set of known *Isoform* and a python set of unknown *Isoform* objects. You can then build an index by calling the *index_isoform_locations* function, and use this index to run *evaluate_alignment* on the alignment that you generate with Aligner.align.

4.5.1 Runtime Performance Requirement

For a 11 million base genome, your *Aligner.__init__* method must take less than 500 seconds to run. Your *Aligner.align* method must take on average less than 0.5 seconds per read. In addition, you will be penalized if your *Aligner.align* is more than 2 times slower than the mean runtime of everyone in the class.

5 Submission

This assignment is due on **Monday 11/13 11:59pm PST**. Please create an Instructional account for CS 176 and submit your project there. Only one person per pair needs to submit, but you could both submit as well just to be safe. Indicate the name of your partner when prompted by Glookup. As stated above, **your submission must keep all required (skeleton) functions / classes / methods in** *project.py*. Put all the python files in *project.zip*; you can create and import other files for helper functions. Do not put your util files in subdirectories, just put them in the same directory structure as *project.py*. You may, depending on the reason, have another chance to submit with a penalty if your first submission crashes.