02-510/710: Computational Genomics, Spring 2022

HW1: Sequence alignment

Version: 1
Due: 23:59 EST, Feb 16, 2022 on Gradescope

Topics in this assignment:

1. Sequence alignment
2. Burrows Wheeler Transform
3. Multiple alignments
4. Suffix tries

What to hand in.

- One write-up (**in pdf format**) addressing each of following questions.
- All source code. If the skeleton is provided, you just need to complete the script and send it back. Your code is tested by the Gradescope autograder, please be careful with your main script name and output format. You may submit the code as many times as you want until the deadline.

It is highly recommended that you typeset your write-up. Illegible handwriting will not be graded. The LaTeXtemplate is provided for your convenience.

Gradescope submission:
Submit the following files containing the completed code to the Gradescope 'HW1 Programming' assignment. *Do not submit a zip file or a folder containing these files. Gradescope expects these files submitted as-is.*

NW.py
BWT.py

1. [**25 points**] **Global Sequence Alignment**

   You should implement the dynamic programming from scratch in Python 3.

   Complete the function `needleman_wunsch` in the python script `NW.py` to implement the Needleman-Wunsch algorithm with the scoring function below:

Match:	1
Mismatch:	-2
Gap:	-1

Note: Your code is graded by an autograder. You may use the `numpy` Python package if desired. The script should be able to run with command line:

```
python NW.py example.fasta
```

The `needleman_wunsch` function should return 3 values: the alignment score, the alignment in the first sequence, and the alignment in the second sequence. The alignment score should be an integer, and the two alignments should be strings with no whitespace. Use a dash '-' to indicate gaps.

See the following snippet of code for an example:

```
> > score, align1, align2 = needleman_wunsch('AGCTA', 'AGGTCA')
> > print(score, '\n', align1, '\n', align2)
1
AGCT-A
AGGTCA
```

Hint: You may have to specifically test for edge cases (e.g., allowing for the output to begin with an indel (gap), handling multiple indels in a row, handling inputs that differ drastically in length).

2. [**25 points**] **Burrows-Wheeler Transform**

In this problem, you will complete several functions in `BWT.py`, related to the Burrows-Wheeler Transform (BWT).

(a) The mouse genome has about 2,800 million basepairs. How many gigabytes would it take to store this information (Hint: first determine how many bits you need to encode a nucleotide)? Fruit fly has the genome size of 122,653,977 base pairs; how many gigabytes would you need to store it?

Solution

Add your solution here.

Genomes are quite large! Compressing the genome to be as small as possible, without losing any information, is vital for efficient storage and computation. Here we will explore a simple, reversible compression technique called Run Length Encoding (RLE). If a character is repeated more than once, we replace it with two instances of that character followed by the length of the run. If it only occurs once, we leave it alone. For example:

$$RLE(\text{“WAAAAHAAAHAAAA!”}) = \text{“WAA4HAA3HAA4!”}$$

(b) Complete the `rle` function in `BWT.py` to return the run-length-encoded version of an input string. **This must be your own code, you may not take code from elsewhere**. Use your function to encode the string:
`"steelerssteelerssteelerssteelers"`
How long is the encoded string?

Solution

Add your solution here.

BWT is a technique to transform a sequence into one that has more repeated runs of characters. The transform is lossless, and easily reversible. As we discussed in class, this helps when trying to search for a substring. It can also help when performing lossless compression. See the Wikipedia article for more details. **Feel free to use their python implementation for your code below.**

(c) Complete the `bwt_encode` function in `BWT.py` to return the transformed version of an input string (You must as a first step insert “`{`” and “`}`” at the beginning and end of the input string, respectively. You may assume that these two characters will not appear in any raw input strings).

(d) Complete the `bwt_decode` function in `BWT.py` to return the original string given the BWT version as input (The input BWT string will contain the “`{`” and “`}`” delimiters, but your decoded output should not).

For questions 2e and 2f below, assume you are using the most straightforward, basic implementation of BWT, and that anytime you need to sort, you use Merge Sort. Note that the complexity of sorting K integers might be different than the complexity of sorting K strings of length N. Typically, the complexity of a sorting algorithm is written as a function of the number of items to be sorted, and is given in terms of the number of comparisons, but assumes that the length of each item is a constant (i.e. $K >> N$). This assumption means that each comparison is a constant-time operation. Is this true for situations when BWT needs to sort? You should see that here the aforementioned assumption does not hold (i.e. $K=N$).

(e) For a string of length N, what is the worst case runtime complexity of `bwt_encode`?

Solution

Add your solution here.

(f) For a string of length N, what is the worst case runtime complexity of `bwt_decode`?

Solution

Add your solution here.

(g) Now compute RLE(BWT(<same string from 2b>)). How long is the encoded string now?

Solution

Add your solution here.

(h) Explain in your own words how BWT makes it so that the transformed strings have long runs of identical characters.

Solution

Add your solution here.

(i) Finally, explore using the BWT followed by RLE for various strings (each of your examples should be at most 100 characters long, and you can use the full alphabet, save for "{" and "}", and **must not be trivial copies of the two given strings in the file**):

i. Give an example string where the final string is longer than the original.

Solution

Add your solution here.

ii. List two example strings where the final strings are shorter.

Solution

Add your solution here.

iii. What type of strings seem to compress better with `rle(bwt(s))`? What are these called in DNA?

Solution

Add your solution here.

In practice, even more compression is applied after RLE(BWT(s)), and there are clever ways to search for matches against a string while it is compressed. BowTie (19,695 citations as of 01/20/2022) and BowTie2 (30,446 citations as of 01/20/2022) are famous DNA alignment tools that make use of these techniques, making it possible to align millions of DNA reads to the genome within hours on a laptop.

3. **[25 points] Multiple Sequence Alignment (MSA)**

(a) We discussed in class (and in problem 1) a method to align two strings using dynamic programming. At each step, the algorithm needs to look at three neighbors to fill in a dynamic programming table cell. Consider now the problem of multiple sequence alignment. Assume we are attempting to align three strings simultaneously using a dynamic programming cube in a similar fashion. How many neighbors will the algorithm need to check to fill in each entry in this matrix? Explain.

Solution

Add your solution here.

(b) Extend this further to $K > 3$ strings, each of length roughly N. What is the runtime complexity of the alignment algorithm in terms of K and N (Explain how you arrive at this answer)? Would you consider this an efficient algorithm?

Solution

Add your solution here.

(c) Does this method result in the optimal multiple sequence alignment? (Yes or No is fine).

Solution

Add your solution here.

(d) Given the run time discussed above, multiple sequence alignment methods rely on heuristics. Consider the following method:
We are performing a progressive multiple sequence alignment. The scoring is (match $m = 2$, mismatch $n = -1$, gap $g = -1$, and gaps that align with each other do not contribute to the score). We first align Seq1 and Seq2. Now consider adding the sequence Seq3=`ACT` to the pairwise alignment. The total score for the MSA (which we are maximizing) is the sum of pairwise alignments between the sequences.

$Seq1$ `A_GT`

$Seq2$ `ACGT`

- What is the optimal alignment and its score for Seq3 under the sum-of-pairs MSA scoring method? Make sure to correctly score induced alignments. Show your work.

  Solution

  Add your solution here.

- What should the mismatch score be so that the two possible alignments (assigning the C of Seq3 to position 2 or position 3) have equal scores?

  Solution

  Add your solution here.

- As we see in part (b), global alignment for multiple sequences is very time consuming, especially when there are a large number of sequences need to be aligned together. Therefore, heuristic algorithms are needed for the multiple sequence alignment problem. Please come up with an efficient heuristic algorithm, explain how it works and give the runtime complexity.
  (Hint: Possible heuristic methods can be based on finding a reasonable ordering of sequences for alignment, e.g. by clustering the sequences or constructing a hierarchical tree according to some similarity metric. If you have other good ideas on how to solve the multiple sequence alignment problem, you can definitely ignore this hint.)

  Solution

  Add your solution here.

4. **[25 points] Suffix Tries**

(a) Draw the **suffix <u>trie</u>** for the sequence "GACGA". Include a picture of your trie in your write-up.

Solution

Add your solution here.

(b) Why is the termination character '\$' necessary? (*Hint:* try removing it from the leaves of your completed suffix trie, and explain any issues that arise). You should make a general statement about why \$ is necessary for Suffix Tries, for any string, not just this example. The example given here might be a special case where it might not be necessary but that is coincidental. It may help to look at another short string, like "banana".

Solution

Add your solution here.

(c) Suppose that we are interested in finding the **longest repeated substring** in the reference string of length N. For example, if we used the above sequence as the reference string, the longest repeated substring would be "GA". Explain how we can do this using suffix tries. What is the runtime complexity?

Solution

Add your solution here.

(d) Suppose that we modify the above problem to finding the longest repeated substring with **at least** K **occurrences**. How should we modify our approach? Additionally, does this change the runtime? Explain why or why not.

Solution

Add your solution here.