02-510/710: Computational Genomics, Spring 2025

HW1: Sequence alignment

Version: 1
Due: 23:59 EST, Feb 9, 2025 on Gradescope

Topics in this assignment:

1. Sequence alignment
2. Burrows Wheeler Transform
3. Multiple alignments
4. Suffix arrays
5. MinHash

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. Note that Q1 and Q5 are graded by autograder, but you must write your own code for Q2 and Q4 as well. *Do not submit a zip file or a folder containing these files. Gradescope expects these files submitted as-is.*

SW.py
BWT.py
suffix_arrays.py
minhash.py

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

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

Complete the function `smith_waterman` in the python script `SW.py` to implement the Smith-Waterman algorithm with the scoring function below:

Match:	2
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 SW.py example.fasta
```

The `smith_waterman` 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 = smith_waterman('GGTTGACTA',
'TGTTACGG')
> > print(score, '\n', align1, '\n', align2)
9
GTTGAC
GTT-AC
```

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 Domestic cat genome has about 2,500 million base pairs. How many gigabytes would it take to store this information (Hint: first determine how many bits you need to encode a nucleotide; 1 gigabyte = 1073741824 bytes)? A Fruit fly has a genome size of 122,653,977 base pairs; how many gigabytes would you need to store it?

Solution

Cat: about 0.582 GB, Fly: about 0.029 GB

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:
`"mississippiicecreammississippi"`
How long is the encoded string?

Solution

length = 38

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

$O(N^2 \log N)$

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

Solution

$O(N^2 \log N)$

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

Solution

33

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

Solution

For a given string, the BWT sorting organizes the cyclic rotations with similar prefixes next to each other. This makes it so similar characters are grouped together in long runs.

(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. Provide an example in which the final string is longer than the original string and include the lengths of both for comparison.

Solution

original (len 21) "abcdefghijk1234567890"; final (len 23) "9k12345678{abcdefghij}0"

ii. Provide an example in which the final string is shorter than the original string and include the lengths of both for comparison.

Solution

original (len 10) "meooooooow"; final (len 8) "m{eoo7}w"

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

Solution

strings with stretches of repeating characters or sequences; tandem repeats

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 (22784 citations as of 01/26/2024) and BowTie2 (44091 citations as of 01/26/2024) 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)**

- We discussed in class a method to align two strings using dynamic programming. Consider now the problem of multiple sequence alignment, where more than 2 sequences are simultaneously aligned to each other. The MSA problem is NP-Hard, as is typically approximated using heuristics. One example is the Star MSA method method:
  To align a given set of strings S:
  - Build all optimal pairwise alignments
  - Select the string S_c such that S_c is the most similar to all of the other strings. S_c should minimize $\sum_{i \neq c} a(S_i, S_c)$, where $a(S_i, S_c)$ is the optimal pairwise cost
  - Align the remaining strings to S_c

  For the following set of strings S, use the Star MSA method to select S_c and write out the optimal alignment of all strings. The scoring parameters are (match $m = 2$, mismatch $n = -1$, gap $g = -1$, and gaps that align with each other do not contribute to the score).

  $Seq1$ `TCGAATC`
  $Seq2$ `CCCCGAA`
  $Seq3$ `TTCGAAC`

  Solution

  S_c: TTCGAAC (Seq3); Optimal alignment: T–CGAATC TT-CGAA-C CCCCGAA–

- The selection of S_c will vary based on the scoring parameters. Give new parameters for the mismatch and gap cost such that a new S_c is chosen.

  Solution

  match $m = 2$, mismatch $n = -3$, gap $g = -1$; new S_c = TCGAATC (Seq1)

- A common application of MSA is to find consensus sequences among a set of strings. Provide an additional biological application where multiple sequence alignment would be helpful.

  Solution

  Phylogenetics: determining evolutionary relationships by comparing sequence similarity and divergence

- Report the longest consensus sequence in the set of strings in msa.fa. You can use code to help you.

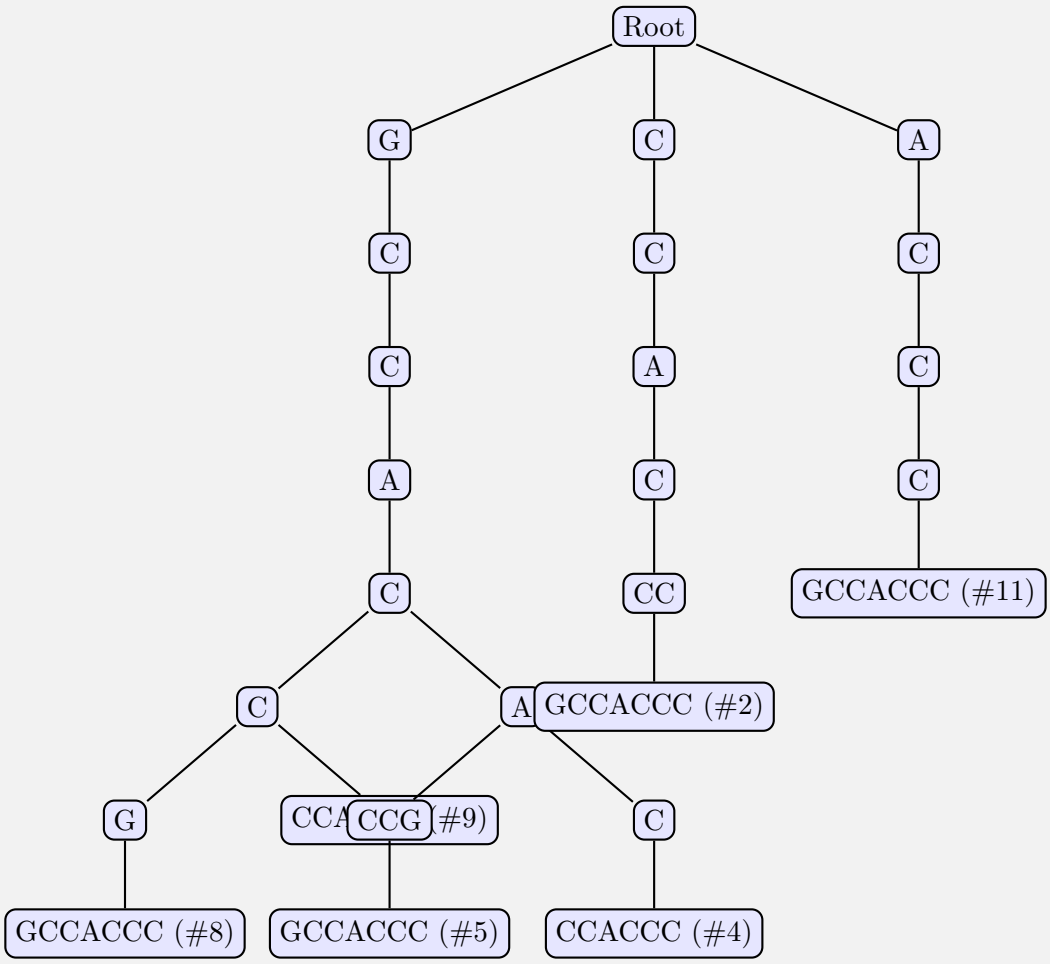
  Solution

  GACGACTTGTAGTAGTCCAATACCT

4. **[25 points] Pattern Searches with Suffix Trees**

- Draw the suffix tree for the following string. Include at least 3 of the suffix links.
  S = GCCACCGGCCACCC

Solution



- Describe an O(n) time algorithm that will find the longest substring that is both a prefix and suffix of S.

Solution

(a) Construct the suffix tree for the string S and its reversed version S' in $O(n)$ time.

(b) Perform a depth-first search of the tree to find the deepest internal node common to both S and S'.

(c) The longest common path to this node corresponds to the longest substring that is both a prefix and a suffix.

(d) Extract the substring from S in $O(1)$, for an overall time complexity of $O(n)$.

- Describe an O(n) time algorithm that will return the number of times a pattern P occurs in S. Hint: Tree traversal methods such as DFS and BFS run in O(n + m) time, where n is the number of nodes and m is the number of edges.

  Solution

  (a) Construct the suffix tree for S in $O(n)$ time.

  (b) Traverse the tree following the characters of P. If traversal completes, the last reached node represents the subtree where P occurs.

  (c) Perform a DFS or BFS from this node to count the number of leaf nodes, which corresponds to the occurrences of P in S.

- Which data structure is more memory-efficient: suffix trees or suffix arrays? Briefly explain why.

  Solution

  Suffix arrays are more memory-efficient because they store only indices, not full nodes with pointers like in a tree.

5. **[25 points] MinHash**

The MinHash sketch measures the Jaccard index (resemblance) of two sets by storing the n minimum hash values of each set; the proportion of the n minimum hash values of the union that are shared by both sets is an estimate of the Jaccard index. Obviously, to get a reasonable error, you will want n to be large (you get roughly $O(1/\sqrt{n})$ additive error).

MinHash has been applied to biological sequences to measure similarity by measuring the Jaccard index of the set of 'k-mers' (length-k substrings) of a sequence. For example, the string `AACCGGTT` has 4-mers `AACC`, `ACCG`, `CCGG`, `CGGT`, `GGTT`.

Complete the function `minhash` in the python script minhash.py to implement the Minhash algorithm for comparing two genomes. The function take as input a fasta file with two genomes and an argument specifying the length k of the k-mers. It should compute the Jaccard index of their k-mer sets to an expected error of ± 0.1. You will need to determine how many hash values to take to achieve this goal.

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

```
python minhash.py example.fasta k
```

The `minhash` function should return 1 value: the Jaccard index.

See the following snippet of code for an example:

```
> > score = minhash('GGTTGACTA', 'TGTTACGG' ,2)
> > print(score)
0.66
```

Similarly, using the coronavirus example fasta provided with the code package, running

```
python3 minhash.py two_coronaviruses.fa 8
```

should return something like

```
Score:  0.31
```

keeping in mind that the exact score may vary a little bit depending on the random hash function you choose.