Bioinformatics Example Sheet 3

Pattern matching

Exercise 1 [Single pattern matching]

(a) Define the *single pattern matching* problem.

(b) Give an algorithm for solving this problem. What is the time complexity for your approach?

Further Reading 1 [Efficient string matching] There exist several algorithms that solve the pattern matching problem in $\mathcal{O}(n+m)$ time. The most famous algorithm is that by Knuth-Morris-Pratt (e.g. see here or here), but a slightly simpler algorithm is the one known as Z-algorithm (e.g. see here).

Exercise 2 [Multiple pattern matching]

(a) Define the *multiple pattern matching* problem.

(b) Give an algorithm for solving this problem. What is the time complexity of your approach?

Exercise 3 [Trie]

(a) Define the *trie* data structure for a set S of strings.

(b) Draw the trie data structure for the strings *cambridge*, *cambden*, *donation*, *donate*, *donor*, *donors*, *donator*.

(c) How would you represent a trie for the English language in a programming language like `C++` or `Java`?

(d) Explain how the *find-string* operation works in a trie with two examples in the trie you constructed in (b). What is its time complexity?

(e) Explain how the *add-string* operation works in a trie. What is its time complexity?

(f) (optional) Implement the *find-string* and *add-string* operations. (This may help if you get stuck.)

(g) Modify the trie data structure to support counting the number of strings in S that start with a given prefix.

Exercise 4 [An interview question (optional)] How would you implement the autocomplete feature for an English keyboard.

Further Reading 2 [An alternative way for efficient pattern matching] There is a way to perform more efficient multiple pattern matching by extending the ideas of the KMP algorithm. This is known as the Aho-Corasick algorithm (you can read more about it here and here).

Further Reading 3 [Game on a trie (optional)] Consider a game where players A and B alternate between appending letters to an initially empty string, in each step forming the prefix of a valid word (e.g. a word that appears in a dictionary). The one who appends the last valid letter wins. Assuming that both players play optimally and A plays first, design an algorithm to determine which of the two players has a winning strategy.

Exercise 5 [Suffix trie]

(a) Define the *suffix trie* data structure for a single string s.

(b) Draw the suffix trie data structure for the string *mississippi*.

(c) What is the time complexity to construct the suffix trie?

(d) How would you use a suffix trie to answer the multiple pattern matching problem?

Exercise 6 [Compressed suffix trie] You may find this visualisation helpful.

(a) What is the worst-case number of nodes used for a suffix trie of a string of length n?

(b) How can you compress the suffix trie? Draw the compressed suffix trie for the string *mississippi*.

(c) How would you modify the algorithm for multiple pattern matching to work on the compressed suffix trie?

Further Reading 4 [Faster construction of suffix trees] The compressed trie is also known as the suffix tree. There is an algorithm by E. Ukkonen ("On-line construction of suffix trees") which constructs the suffix tree in $\mathcal{O}(n)$ time, where n is the length of the string. However, the algorithm is quite involved. There is a slightly simpler approach by constructing the suffix automaton (e.g. see here) and then converting it to a suffix tree.

Exercise 7 [Other applications of suffix tree (optional)] These are not needed for the course, but may give you more insights into suffix trees:

(a) Compute the number of different substrings of a string s.

(b) Find the longest common (continuous) substring of strings s_1 and s_2. [*Hint:* Construct the suffix tree for the string s_1 + "$" + s_2].

(c) Find the longest palindromic substring in s. [*Hint:* Try to reduce it to the longest common substring problem].

(d) Find the shortest lexicographical suffix of a string s. Use this to efficiently check if one string is a cyclic rotation of another.

Genome compression

Exercise 8 [Run-length encoding]

(a) Describe the *run-length encoding* using an example.

(b) What is the time complexity for encoding and decoding?

Exercise 9 [Burrows-Wheeler encoding] You may find this visualisation helpful.

(a) Explain how the encoding part of the *Burrows-Wheeler transform* works using "mississippi" as an example.

(b) Why might this encoding produce strings where same characters are next to one another? When is this more profound?

(c) Using normal sorting what is the time and memory complexity of BWT encoding? How does this improve if you use a suffix tree?

Exercise 10 [Burrows-Wheeler decoding]

(a) Explain how the decoding part of the *Burrows-Wheeler transform* works by decoding the example of Exercise 9.

(b) Using sorting what is the time and memory complexity of BWT decoding?

(c) Describe the *first-last* property and argue why it holds.

(d) Show how this property improves time and memory complexity of the BWT decoding.

Exercise 11 [Suffix array] You may find this visualisation helpful.

(a) Describe the *suffix array* data structure using “mississippi” as an example.

(b) How can you obtain the suffix array from the suffix tree?

(c) How can you use the suffix array for pattern matching?

Exercise 12 [Approximate matching]

(a) Define the *approximate pattern matching* problem.

(b) Describe the *seeding* approach.

(c) Describe how to do approximate matching using BWT.

(d) (+) Describe how to do approximate matching using suffix trees. [*Hint:* Modify the suffix tree to do approximate matching with 1 mismatch. Then, with 2 mismatches and so on.]

Algorithms to identify subsequences

Exercise 13 [Probability reminder] Recall the following formulas from the Part IA Probability course:

- Baye’s rule;
- $\Pr(X = x) = \sum_y \Pr(Y = y, X = x) = \sum_y \Pr(X = x \mid Y = y) \cdot \Pr(Y = y)$;
- $\Pr(X_1 = x_1, X_2 = x_2, \ldots, X_N = x_N) = \Pr(X_N = x_N \mid X_{N-1} = x_{N-1}, \ldots, X_1 = x_1) \cdot \Pr(X_{N-1} = x_{N-1}, \ldots, X_1 = x_1)$.

Exercise 14 [Hidden Markov Models] You may find the following notes helpful.

(a) Define the HMM model including the *transition probabilities* a_{ij}, the *start probabilities* and the *emission probabilities*. State the independence assumptions made by the model.

(b) Given a sequence x and a parse π, derive the joint likelihood $\Pr(\pi, x)$, assuming the parameters of the HMM are known. What is the time complexity of your algorithm?

(c) Given a sequence x, find the parse π^* that maximises the likelihood $\Pr(\pi^* \mid x)$, assuming the parameters of the HMM are known. What is the time complexity of your algorithm?

(d) Given a sequence x, show how to compute the parameters that maximise $\Pr(x)$.

(e) Give a few examples of problems where HMMs can be applied in Bioinformatics.

Exercise 15 [Evaluation] Define *false positives*, *false negatives*, *true positives*, *true negatives* and the *F_1 score.*

Exercise 16 [DNA storage] Attempt **[2020P9Q2 (e)]**.

Exercise 17 [Adleman’s approach] Attempt **[2019P8Q2 (d)]**.

Exercise 18 [Doob-Gillepsie algorithms]

(a) Describe the *Doob-Gillepsie* algorithms.

(b) Attempt **[2010P9Q3 (d)]**.

(c) Attempt **[2009P9Q3 (d)]**.