Programming Lab II
Handout 7

Dr. Martin Vogt
`martin.vogt@bit.uni-bonn.de`

Jun 9, 2020

Deadline: Jun 28, 2020 **Next Handout: Jun 23, 2020**

Read mapping and exact string matching

Next generation sequencing typically produces a large set of short reads that need to be mapped to a genome. This process is known as *read mapping* and, seen from a computer science perspective, this corresponds to identifying a large number of (short) substrings in a large body of text. This seems like a very simple problem and Python provides some string methods like `find`, `index`, and `count` that can be used to find occurences of a pattern in a text. (Alternatively, one can also consider the regular expression matching capabilities of the `re` module.) This seems especially straightforward, if we do not allow mismatches between the pattern and the text, which we will also won't allow for this exercise.

1. *(5+3 pts) Naive string matching*

   (a) *(5 pts) Mapping reads*
   Write a function `map_reads_naive(reads,sequence)` that takes a collection (e.g., a list) of (short) strings, the `reads`, and identifies all occurring positions in `sequence`. The function should return a dictionary, where the keys are those reads that occur in the sequence and the value of a key is a list of positions (i.e., the indices) where the key occurs.

   E.g. Let `'abracadabra'` be the sequence and `['a','bra','cada','arba']` be the list of short reads. The function should return a dictionary:

   ```
   {'a': [0,3,5,7,10], 'bra': [1,8], 'cada': [4]}
   ```

   (The order of the items may vary.)

   (b) *(Optional 3 pts) English words in the genome*
   The function can now, for instance, be used to solve the vital problem of identifying English words in the proteome of organisms. Use the dictionary file `words.txt` and select all the English words with at least four letters. Find the occurrences of English words of at least four letters in the proteomes of *Escherichia coli* (file `ecoli-proteome.faa`) and *Drosophila melangoster* (files `chrX.faa`, `chr2L.faa`, `chr2R.faa`, `chr3L.faa`, `chr3R.faa`). Answer the following questions:

      i. For *Escherichia coli* and *Drosophila melangoster* find the longest English words and the proteins in which they occur.

      ii. Find the proteins of *Escherichia coli* and *Drosophila melangoster* containing the most English words of at least four letters.

   Because the standard one-letter amino acid code lacks some important English letters, you can (optionally) choose to not distinguish between the letters `I` and `J`, the letters `O` and `Q`, and the letters `U` and `V` when identifying English words in a proteome.

2. *(6 pts) Speed of matching*
   Finding English words in a proteome is not very close to the practical problem of mapping reads of nucleotide sequences to a genome. A slighlty more realistic scenario is to use randomly generated data. Note, that for simulating realistic data using simple random sequences is still not very good, as, of course, genomes are not random. For realistic evaluation one would like to either use real experimental data or a more sophisticated randomized model. However, for the assessment of mapping efficiency, using simple random data is good enough for now.

   (a) *(3 pts) Generate random data*
   Write a function `get_random_data(genome_size,nr_of_reads,read_length,random_seed)` which generates a random string over the alphabet $\{a, c, g, t\}$ of length `genome_size` representing the genome. From this genome, randomly select `nr_of_reads` substrings of length `read_length` comprising the set of reads. Use the parameter `random_seed` to set up the random number generator, so the identical set of random sequences can be reproduced. The function should return a tuple containing the sequence and the set of reads.

   (b) *(3 pts) Test performance of read mapping*
   Use random data to determine the running time of `map_reads_naive` of exercise 1. ()a) for

      i. A genome of size 10^6 and 10^4 reads of length 10.
      ii. A genome of size 10^6 and 10^5 reads of length 11.
      iii. A genome of size 10^7 and 10^4 reads of length 12.

   Compare the running times to those of ex. 3 (b) and 4 (d).
   Warning: ii. and iii. might take quite some time (around 5 min. – 60 min.) so you might want to wait testing on these data sets until you finished everything.

As you will see, using the naive method for read mapping tends to be quite time consuming. When looking for a single substring in a single string, for instance using `index` or `find`, we can safely assume that Python already implements an efficient algorithm. (Even for this seemingly simple problem efficient solutions can be surprisingly complex, see, for instance, the *Boyer–Moore string-search algorithm* or the *Knuth–Morris–Pratt algorithm*). However, here we deal with the problem that we want to match a lot of different short patterns, the reads, to one large string, the genome.

The basic idea for speeding up solutions to this problem is by either preprocessing the reads, so that we can more or less match them simultaneously to the sequence or by preprocessing the sequence, making it easier to look up the positions of the substrings.

3. *(9 pts) Preprocessing the "genome"*
   One idea for preprocessing is to keep a complete dictionary of all short reads of a fixed length occurring in the genome. For instance, for the "genome" `acgtatcgtatcc` we would want our dictionary of four-letter word to look like this:

```
{'acgt': [0],
 'atcc': [9],
 'atcg': [4],
 'cgta': [1, 6],
 'gtat': [2, 7],
 'tatc': [3, 8],
 'tcgt': [5]}
```

   We can generate such a dictionary by scanning the genome sequence once, from left to right, looking at the four letter sequences `acgt`, `cgta`, `gtat`,... and adding the positions to the appropriate dictionary entry. Now, if we want to look for a read (e.g., `gtatcc`) we look up the positions of the prefix of length 4 (e.g., `gtat`) and we only need to check if the read occurs at one of the positions indicated by the occurrences of the prefix (e.g., 2 and 7, where the subsequence starting at 7 matches the read).

(a) *(6 pts)* Implement the read mapping function `map_reads_lookup(reads,sequence,lookup_size)` using the outlined strategy. Here, `lookup_size` is the size of the substrings to be used in the lookup table.

To make this function efficient it should preprocess the sequence *once* at the beginning of the function and then use the generated lookup dictionary for finding all the reads. It should return a dictionary as described in ex. 1 (a).

(b) *(3 pts)* Use a lookup size of 4 and 8 and determine the running times for the sample data generated in ex. 2 (b).

Note, that generating lookup-tables requires quite a bit of memory. For one, the total number of indices that need to be stored corresponds to the size of the sequence, requiring about 4 times more storage than storing the sequence itself. Also, as the keys consist of subsequences, using a large lookup size requires storing a large number of subsequences, which can increase memory requirements by a considerable factor making a tradeoff between lookup size (which translates to increased performance) and memory consumption necessary.

Popular read mapping tools nowadys frequently use different techniques like *suffix trees* or *suffix arrays* and the related *Burrow-Wheeler transform.* Suffix arrays rely on the idea of considering all suffixes of a genome, i.e., the substrings `sequence[i:]` and sort all these strings lexicographically. A read can then simply be found by performing a binary search on these substrings. It would be infeasible to actually store all suffixes and sort them as the memory requirements would be quadratic with respect to the genome size; instead, one only needs to store the indices where each suffix starts and then sort the indices according to the lexicographic order of the suffices.

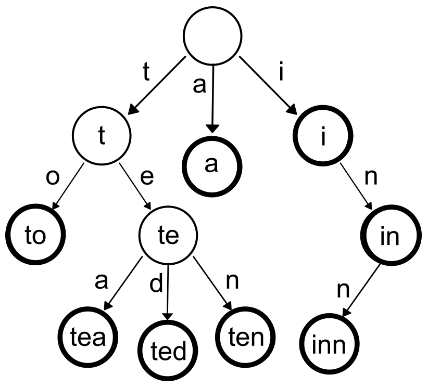
Suffix trees organize the suffices in a tree like structure, which also allows for an easy lookup of reads.

In the following, however, we will use a tree-structure to preprocess the reads instead and then use the preprocessed reads to look them up in sequences. This should be helpful for speeding up the identification of English words in the proteome of organisms, as in this case, the set of reads, i.e., the words of the English language, are fixed and the search is to be applied to a large number of protein sequences.

4. *(10 + 3 pts) Preprocessing the reads*

(The method outlined here is the basis of the Aho-Corasick method matching sets of patterns.)

The idea is to organize all the words from a set of patterns (i.e., the reads) in a tree, which is known as a *keyword tree* (and is similar to a *trie*). Each word should be represented in the tree as a path starting at the root and ending at a node. The following example indicates a tree for the words a, i, in, inn, to, tea, ted, ten and to. Note, that a complete word does not have to end at a leaf node, it can also be an inner node if a word is a prefix of another (e.g., in and inn).



(a) *(4 pts)* Write a Python class `KeywordTree` for representing such a keyword tree. The class will need to contain at least one method `add_word` for adding new words to a tree. How this might be done:

- One way might be to represent the children of a node using a dictionary; the keys could represent the edge labels and the values are the child nodes. In this way, the nodes do not need to have an explicit label, however, for each node you should at least indicate whether the path from the root to the node is a valid word from the set od reads.
- The method for adding words would first look if it already had an edge corresponding to first letter of the word; if not, the dictionary should be extended with a new node as value for the key corresponding to the first letter. Then, or if the node already exists, the add-word method should be called for the child node using the word with the first letter removed. This way, a word is added by proceeding down the tree until the add-word method is called with an empty word. In this case, the node should be marked as representing a complete word (corresponding to the unique path from the root to the node).

A function for generating a keyword tree from a list of reads might then look like this:

```
def make_keyword_tree(reads):
    tree = KeywordTree()
    for word in reads:
        tree.add_word(word)
    return tree
```

(If you feel like it, consider adding an (optional) argument to the constructor so that the call `KeywordTree(reads)` returns a keyword tree where all words in `reads` have already been added.)

(b) *(3 pts)* With such a keyword tree, it is easy to check whether a sequence starting at index `i` contains a word from the keyword tree: Check if there is an edge from the root of the tree corresponding to the first letter `sequence[i]`, if so, proceed to `i+1` and to the corresponding child; repeat. If a child is marked as a word, the subsequence `sequence[i:i+k+1]` is a valid word. If at one point, no appropriate edge exist, stop. The process is repeated for all `i` in the range `0` until `len(sequence)`.

Write a read mapping function `map_reads_tree(reads_tree,sequence)`, which uses keyword trees for read mapping. Here, te argument `reads_tree` should be the keyword tree generated from the reads. The function, again, should return a dictionary as described in ex. 1 (a).

(c) *(Optional 3 pts)* Repeat the exercise 1 (b) of finding the longest English words in the proteomes of *Escherichia coli* and *Drosophila melangoster* and compare the running times.

(d) *(3 pts)* Determine the running times for the sample data generated in ex. 2 (b).