COMP3506/7505 Assignment 2

Version 2010-09-30 (fixed typos from original version).

This assignment is worth **15%** of the total course marks.

Administration

Due 5pm Friday 15 October 2010 (Week 11)

Assignments will be returned during tutorials. Refer to the course profile for policy regarding late submission of assignments.

School Policy on Student Misconduct

You are required to read and understand the School statement on student misconduct, available at:

http://www.itee.uq.edu.au/about_ITEE/policies/student-misconduct.html

In particular, you must familiarise yourself with UQ's definition of plagiarism, and understand the School's policies regarding student plagiarism.

Submission

The assignment must be submitted through the ITEE online assignment submission service:

http://submit.itee.uq.edu.au

All parts of the assignment must be in a single zip archive, created by exporting the modified project from Eclipse. If you are responding to any of the bonus questions (compulsory for COMP7505), also create a directory in your project named "`doc`". Place a PDF file named "`report.pdf`" in this directory documenting your written responses. To then create the archive:

- Select the File → Export… menu item
- Select General → Archive File (as the export destination)
- Enter the destination file name, which must be `Student<your_student_number>.zip` with `<your_student_number>` replaced by your full eight-digit student number.

Export the project only **after** you have completed all parts of the assignment. Please ensure that your exported archive file contains all of your work before submitting it.

Coding Style

Your code must be formatted using the built-in Eclipse coding style. Open the Eclipse "Preferences" window, go to Java → Code Style → Formatter and under "Active Profile" make sure that "Eclipse [built-in]" is selected. You should reformat your code to follow this coding style before submission, by selecting Source → Format or Source → Format Element.

Resources and restrictions

You **may not** use any classes from the Java Class Library except those in the *java.lang* package. You **may not** use any code that you did not author, except for code provided to you by us.

To complete this assignment you must first download an Eclipse archive file from:

http://www.itee.uq.edu.au/~comp3506/Resources

Import the downloaded archive into a new Eclipse workspace using the "Existing Projects into Workspace" option:

1. Select the File → Import menu item
2. For the Import Source, select General → Existing Projects into Workspace and click "Next"
3. Select "Select archive file", "Browse", and select the archive file and click "OK"
4. Select the available project and click "Finish"

Introduction

Assignment 2 is divided into two parts both worth 15 marks; these 30 marks comprise 15% of the overall course mark. There are 3 bonus marks included in the questions; the tasks associated with these marks are optional for COMP3506 students, and compulsory for COMP7505 students.

This document details a domain-specific analysis problem, and shows how it can be solved more efficiently through clever application of algorithms and data structures. Importantly, this document describes the specific problems that your work should solve. Marks will be given as indicated, and only for accurate code that passes a range of tests (partially based on novel data sets). You will be provided with several complete Java classes, and "stubs" for some other classes—all of which will be provided in the form of an Eclipse project file. The problems in this document will mostly request you to complete the "stubs". In the Eclipse project (under "data") you will have access to several data files on which you can test the accuracy and performance of your implementations. When you are done, your code should be saved as an Eclipse project and submitted as such (see above).

Background context

This assignment involves writing code for analysing **DNA sequence** data. DNA sequences can be thought of as long strings composed only of **the letters A, C, G and T**, corresponding to 4 different nucleic acids (see Figure 1). You will develop Java code for:

- (Part A) finding "**a median *k*-mer**"—a representation of the most frequently occurring "words" (a *k*-mer is a word composed of *k* letters) in specified DNA; and
- (Part B) creating and querying a tree representing all *k*-mers occurring in a DNA sequence (also known as a "**trie**").

There are serious computational issues with naïve approaches to the problems in both Part A and B. Significant challenges abound if we wish to analyse large sequence sets. You can overcome some of these by using your skills in algorithms and data structures.

DNA binding

In molecular biology, the development of methods like those above is (amongst other things) imperative to the discovery of DNA binding "motifs"—sites at which special "regulatory" proteins bind to modulate the activation of genes. You will be given real DNA sequences from bacterial genomes that are believed to harbour actual but as yet unknown binding sites (see Figure 1 for a schematic illustration of the concepts). Typically, DNA binding sites are 6-20 positions long. As you may know DNA molecules are "double-stranded", but we will stick to analysing just one—the

opposite strand is “complementary”. Lastly, any analysis will have to account for naturally occurring “mutations”—the letters mentioned above vary between individuals. Variation is “accepted” to some degree, though at some point, mutations will break biological activity.

Figure 1: A DNA sequence with two genes identified (solid green; arrow indicates the direction of a gene). We search sequences located “upstream” from each gene (dotted red). Regulatory proteins (gray ellipses) can be seen bound to the DNA—it is the make-up of their “binding sites” that we are after.

Sequence data and the FASTA file format

The data you are requested to work with are provided in a simple text file format. Consider an example DNA sequence file containing 100 nucleic acids “upstream” from two known genes (*ydeP* and *yfdX*) in *E. coli*.

```
>ydeP Escherichia_coli_K12_substr__MG1655:NC_000913.2:1584511-1584610:R
CCTACTCTTATATATCCATGTTGGCGATAATCCCTTTGTATGTACTGTGCATCATCGCTA
TTACAAATCCTAATAATTCATTTCCACACAGGATAAGTAG
>yfdX Escherichia_coli_K12_substr__MG1655:NC_000913.2:2492425-2492524:R
GATAACCCTTTCAAAATGACCGTTGCTCTCTGATTTCTCATTTCATGCTCACCCAATATG
ATGGCGGCGTTTTCTAAAACTGTTAAAGAATGAGGTAAGT
```

The first line of each sequence entry (starting with the ‘>’ sign) contains information about the sequence, e.g., name, species, location on chromosome, etc. The following lines (up until the end of file or the next sequence entry; line breaks are ignored) contain the nucleic acid sequence (a string composed of A, C, G and T). You have access to a complete Java class definition `DNASequence` to hold all data you are interested in, and to read FASTA files.

Part A: A method for computing “median” k-mers (15 marks total)

In part A of the assignment, we consider two basic algorithms, `getDistance` and `findMedianKMer`, and in Problems 1 and 2 you are requested to implement variations of each in Java.

`getDistance` (see below for pseudo code) takes a single DNA sequence (e.g. the instance of *ydeP*) and a specific word (e.g. the 7-mer “GATAACC”). The algorithm determines the number of mismatched letters when the specified word is aligned optimally to the sequence—the minimum Hamming distance. The algorithm scans all possible positions for the alignment to find the one that results in the smallest mismatch. (For *ydeP* above, it should return 1; see highlighted section in the sequence above for the aligned position.)

```
Algorithm getDistance(DNA, kmer)
      Input: DNA = a DNA sequence, kmer = a k-mer
      Output: distance = the minimum number of mismatches of k-mer when
      optimally aligned to DNA sequence

      N = length of DNA
      K = length of kmer
      mismatches = K { start with the worst possible distance }

      For each position i in DNA – the number of positions in kmer + 1
            count ← 0
            For each position j in kmer
{1}               If (symbol i in DNA <> symbol j in kmer) then
                        count ← count + 1
{2}         mismatches ← MIN(count, mismatches)
      Return mismatches
```

Problem 1: Develop methods for computing the minimum Hamming distance between a sequence and a k-mer (5 marks)

You have access to a class `DNASequence`, a class `Alphabet` and a class `KMer` . You are also provided with a stub for `MedianKMer` that is intended to encapsulate both the aforementioned algorithm and the code in Problem 2 (see separate code and Javadocs). Read the algorithm above carefully—you are requested to implement a variation of this algorithm. A number of tests are provided in `TestMedianKMer`. Your code must pass the first four (that are relevant to getDistance implementations). You will need to study both Alphabet and KMer to understand how sequences are encoded, e.g. that elements are represented numerically: 'A'=0, 'C'=1, 'G'=2, and 'T'=3.

`public static int getDistance(DNASequence seq, KMer word):`
The iteration over symbols in *k*-mers in `getDistance` above can be ended early if we notice that the number of mismatches is already at or beyond the current best [see line marked with {1}]. Implement this more efficient method in `MedianKMer`. Follow the definition in the provided Java file. Do not change it as this may cause automatic tests to fail, resulting in zero marks.

`public static int[] getDistances(DNASequence seq, KMer word):`
There are situations when it is useful to know the full list of distances. Implement another variation of `getDistance` that returns the distance of the word to each position in a DNA sequences, i.e. an array with a value for each valid start position, keeping in mind that a word can not extend beyond the end of the sequence. Again, follow the definition in the Java file.

`findMedianKMer` is more exciting. In its current form (see algorithm pseudo code below) it is a "brute force search" for the *k*-mer that has the minimum (Hamming) distance to a set of sequences. We call this the median *k*-mer. (Biologically speaking, for the sequences in the example we suspect that the same type of protein binds to, and activates each of them, and we wish to establish what is the most probable binding "motif".)

```
Algorithm findMedianKMer(DNAs, string, k)
      Input: DNAs = set of DNA sequences, string = prefix list of searched
      symbols (empty initially), K = k-mer length
      Output: distance = the number of non-matching symbols in k-mer when
      optimally aligned to each DNA sequence, kmer = the k-mer that was used to
      determine the distance

      If (string has K symbols) then
            For each DNA sequence in DNAs
                  distance ← distance + getDistance(sequence, string)
            Return (distance, string)
      Else then
            { we are not at a leaf of the tree of all possible strings }
{1}         { so unless we think of something... we recurse down the tree }
            best_dist ← K * number of sequences in DNAs + 1
            best_kmer ← nil
{2}         For each symbol in alphabet
                  ext_string ← concatenate string and symbol
{3}               (dist, kmer) ← findMedianKMer(DNAs, ext_string, K)
                  If (dist < best_dist) then
                        best_dist = dist
                        best_kmer = kmer
      Return (best_dist, best_kmer)
```

We make a number of remarks and then specify the problem you face. First, a brute force search like this is computationally very costly. In fact, its running time is exponential in the length of the *k*-mer. However, we can use a "branch-and-bound" design, and "heuristics" as explained below.

Consider the implicit search tree of median *k*-mers (see Figure 2 for a schematic illustration of such a tree for *k*=3). The leaves are complete *k*-mers and we can easily determine and sum their distances to the sequences using the method in Problem 1 (and the results—a distance and the *k*-mer with that distance—are passed "upwards" via recursion in the algorithm). However, the intermediate nodes (the strings that do not yet have *k* letters) are not evaluated. Can they be evaluated? If so, what can this evaluation be used for? As it turns out, this evaluation can be used to make an early decision about whether it is worth our while to branch out—and thereby remove whole sub-trees from our search.

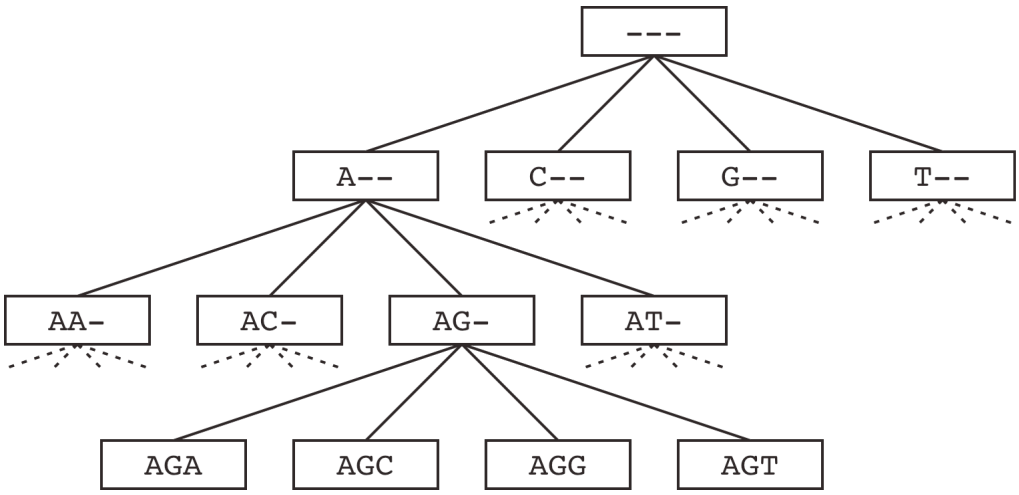

Figure 2: A search tree of k*-mers, where* k*=3. Intermediate nodes represent partial* k*-mers (strings of length <* k*).*

For any *k*-mer $(s_1, s_2, \ldots, s_k)$, the distance between a sequence and $(s_1, s_2, \ldots, s_k)$ is always greater than or equal to the distance for $(s_1, s_2, \ldots, s_{k-1})$. Hence, we can view the distance for $(s_1, s_2, \ldots, s_{k-1})$ as a lower bound on the distance for $(s_1, s_2, \ldots, s_k)$. This value is not a replacement for the distance we compute at the leaves of the tree—but can inform the continued search as discussed next.

Assume now that we have computed a distance *d* for a complete *k*-mer (that is, we have visited a leaf node). This operation gives us a guarantee: we will not do worse than *d*. We keep a record of the *k*-mer and *d* for later. When we continue traversing the tree, we can measure the distance *d'* of a partial *k*-mer (that is, an intermediate node). If we find $d' \geq d$, there is no point in traversing the sub-tree: a *k*-mer under this point will add another symbol and thus at best have a distance of *d'*—it will never decrease as we follow a path.

For example, we search for a median 7-mer and establish that "ATAACCA" has a distance of 1. Not before long, we visit the intermediate node for the partial *k*-mer "CCCC". The distance for "CCCC" is 1, and even if we were to optimistically assume that some *k*-mer with "CCCC" as a prefix (e.g. "CCCCAGT", and "CCCCTTT") added no additional mismatches, we would still not do better than our guaranteed "bound". We thus skip the whole sub-tree under "CCCC", removing $4^1+4^2+4^3 = 84$ node visits that would never lead to an improvement in distance.

Problem 2: Develop a "branch-and-bound" variation to find the median k-mer (10 marks)

Again refer to the class `MedianKMer`*, and add methods as described below. The Java file for* `MedianKMer` *includes a class definition* `Distance` *that simply holds a distance and the* `KMer` *instance that this distance was based on (see return statement in the* `findMedianKMer` *algorithm). Also note that MedianKMer can be run using command line arguments (specifying the file from which to read sequences, etc). A number of tests are provided in* `TestMedianKMer`*. Your code must pass all of them, except* `testFindMedianKMer5`*.*

`public Distance findMedianKMer(int k)`: Develop this method in accordance with the recursive algorithm above, evaluating each *k*-mer on all sequences supplied to the constructor of `MedianKMer`, but use an efficient "branch-and-bound" design as discussed. Set "bounds" on basis of what we have seen earlier while traversing the tree, make assessments at intermediate nodes before branching and terminate recursion if applicable. In

implementing the recursion, you will need to introduce another method `public Distance findMedianKMer(…)` that accepts additional parameters for book keeping. (For example, you need to pass the partial *k*-mer like we do in the pseudo code.) Use `Distance` to hold information that is relevant to your bound.

Bonus mark (compulsory for COMP7505): In regard to your implementation describe worst-case running time of `findMedianKMer`, and evaluate practical running times for 2-25 sequences when *k* ranges between 5 and 12. Try different sequence files in the data directory and different values of *k* and tabulate the running times and numbers of nodes/*k*-mers in the search tree. Add your description to a PDF document that is submitted as a separate document with your code (see submission instructions).

The "branching-order" trick: In practice, the "branch-and-bound" strategy will lead to major improvements in terms of running time compared to the original, exhaustive algorithm. The original and improved algorithms both iterate through the alphabet in the same order at every branching point [see line marked with *{2}*]. We would like to search the most promising *k*-mers first so that a strong bound can be established early. The order in which we "branch" out can have a major impact on the effectiveness of the search. We should thus consider the order in which new *k*-mers, extending the *k*-mer of the current node, are evaluated. Do we have any information on which to "sort" symbols? We shall return to this trick after completing Part B (and Problem 4).

The "re-use distance calculations" trick: Using the "branch-and-bound" strategy, as we descend the tree towards the leaves, we are computing distances for a word one symbol longer than that in the previous step [see line marked with *{3}*]. In fact, we have already computed the distance of the current *k*-mer up until position $k - 1$. It should thus be possible to re-use those distance calculations and simply add the match/mismatch (distance 0/1) of the *k*th symbol for each position in the sequence. We shall revisit this idea in Problem 4 after considering the main algorithm for Part B.

Part B: A tree for matching k*-mers with frequency (15 marks total)*

Even long DNA sequences do not contain all possible combinations of *k* letters when *k* is large. We can use tree structures to efficiently store and access *k*-mers. The type of tree we are considering in this section is sometimes known as "**tries**"—feel free to read section 12.5 of the course text to get a deeper understanding.

Figure 3 illustrates a tree for our type of data. We define one algorithm for constructing it from a single DNA sequence.

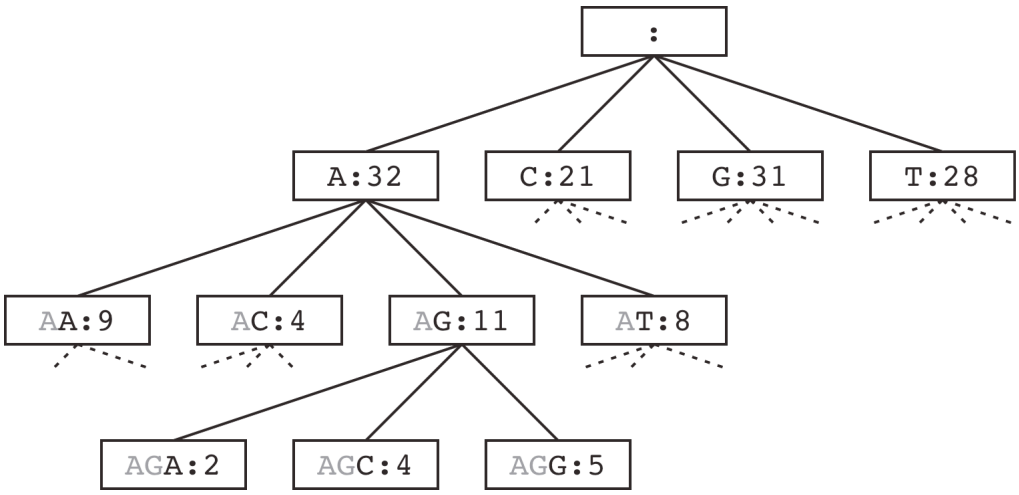

Figure 3: A trie structure for k-mers with counts of occurrences attached.

```
Algorithm buildTrie(DNA, int K)
      Input: DNA = DNA sequence, K = k-mer length
      Output: a tree with maximum depth K, with counts of k-mers in DNA

      Root ← Create empty tree
      For position i in DNA
            For j in [0 .. min(K, |DNA| - i) - 1]
                  kmer[j] ← DNA[i + j]
            { now put this k-mer in the trie }
            Parent ← Root
            For symbol in kmer { for each level of the tree }
                  Current ← get child of Parent for symbol
                  If Current does not exist then
                        Current ← create node for symbol, set count = 0
                        Add Current as child to Parent
                  Update Current with one more occurrence (count + 1)
                  Parent = Current { traverse down a level }
      Return Root
```

We can use a *k*-mer tree to store and access frequencies of all (overlapping) words up to a specified length *K*. In particular, we note that when the tree has been constructed, to access the frequency of any word is very efficient (proportional to the length of the word). In fact this can be used to make a preliminary assessment (a heuristic) on the usefulness of following a path in the median *k*-mer search tree (see *The "branching order" trick* above).

Example: Assume your sequence data is "GGACACA" and *k*=2, then the resulting tree will be Root(A:2(C:2) C:2(A:2) G:2(A:1 G:1)). Requesting the count of "CA" involves visiting Root, then C, then A where "2" is found. You can also request the count of "A" resulting in "2" (after visiting Root then A). If you look back at the original string you note that there is a third A that was not counted—because it never appears first in a 2-mer. Finally, the count of all words that are not represented in the tree is "0" by definition.

Problem 3: Implement a k-mer trie structure (8 marks)

You have access to the beginnings of a class `TrieKMer` *and* `TrieNode` *(for representing a node in the tree structure). Develop and add the methods below to complete Problem 3 and to prepare for Problem 4. You must follow definitions and ideas documented in the code. There are tests in* `TrieKMerTest` *that your code must pass.*

`public TrieKMer(DNASequence[] seqs, int depth)`: The constructor of `TrieKMer` should take an array of sequences, and construct a trie consisting of all words of a length up to and including `depth`.

`public void putKMer(KMer kmer)`: This method in the class `TrieKMer` should update the trie to count the occurrence of the `KMer` including the sub-k-mers (as short as 1) and if required add nodes to the trie to represent the complete k-mer.

Bonus mark (compulsory for COMP7505): Discuss the *theoretical* running times of your implementation for updating the trie and for accessing the counts from a completed trie. Your discussion must be present in the PDF attached to your submission as instructed.

Problem 4: Make improvements to finding the median k-mer (7 marks)

Again refer to the class `MedianKMer`*, and add/change methods as described below. Also remind yourself about the ideas mentioned at the end of Part A.*

Modify `findMedianKMer` to (a) search the most promising *k*-mers first by accessing the trie structure from Problem 3, and (b) re-use the distance computations from parent nodes. Your code must be able to complete a search for an 8-mer in `data/chipseq.fasta` (~2300 sequences) and a 12-mer in `data/ihfA_26.fasta` (~26 sequences) without excessive waiting (<10 minutes each using a standard desktop).

Bonus mark (compulsory for COMP7505): Describe and explore *practical* running times of `findMedianKMer` resulting from your modifications above. Try different sequence files in the data directory and different values of *k* and tabulate the running times and numbers of nodes/*k*-mers in the search tree. Demonstrate the impact of each modification (a vs b). Add your description to a PDF document that is submitted as a separate document with your code (see submission instructions).

Appendix: More curious biology and Where does the data come from?

Bacterial genomes are relatively small and simple. *E. coli* is one of the most popular reference genomes and has a reasonably well-understood "regulatory network", linking proteins to genes they regulate. Check out http://ecocyc.org, click "Tools" and select "Regulatory Overview". This takes you to a map of genes.

Example: Search for malT. malT is a so-called transcription factor—a type of regulatory protein. It activates a number of other genes including malZ, malP, malS, malE and malK. To retrieve relevant sequence data, you can use RSAT http://rsat.ulb.ac.be/rsat. Click "Sequence retrieval". Select organism "Escherichia coli K12 substr MG1655". Paste in the names of the genes that are bound to by your protein (e.g. the list above). Choose "upstream" and from "-120" to "-20" to extract 100 nucleic acid string upstream of your genes. (This region is often referred to as "promoter" and is typically bound by transcription factors; see Figure 1.) Press "GO' to download—save as FASTA. Watch out if you get very short sequences (shorter than *k*). You can find this data in the file `malT_5.fasta`. A slightly longer set is in `arcA_9.fasta`. `chipseq.fasta` contains over 2000 sequences that were extracted from mouse cells using so-called ChIP-Seq technology.

You can see if the *k*-mer you find is a known binding site using the tool TomTom (http://meme.sdsc.edu; click TomTom). Paste in a DNA "motif"—the MedianKMer skeleton class has an option to produce a report for a found *k*-mer (e.g. `MedianKMer -f malT_5.fasta -q GATGAGGGAT`). The last part of this report is a "motif" representation that TomTom accepts.) Select the "E. coli" database (or Mouse if you are studying mouse data) and "Search". TomTom finds statistically significant matches with known binding sites.

"Motif" discovery is an active field of research involving both computer science and biology. For more information about one set of computational tools see our recent paper.

MEME SUITE: tools for motif discovery and searching. Bailey TL, Boden M, Buske FA, Frith M, Grant CE, Clementi L, Ren J, Li WW, Noble WS. *Nucleic Acids Res*. 2009 37:W202-8. http://www.ncbi.nlm.nih.gov/pubmed/19458158