Java Advanced Data Structures and Introduction to Bioinformatics

Submitted: Sep 11, 2018
Decision: Sep 13, 2018

Downtown Magnets High School (051576)

Submission Feedback

APPROVED

Basic Course Information

Title:	Java Advanced Data Structures and Introduction to Bioinformatics
Transcript abbreviations:	JAVA ADV DS & INTR BINF AB CTE / 604309/10
Length of course:	Full Year
Subject area:	College-Preparatory Elective ("g") / Mathematics - Computer Science
UC honors designation?	Yes
Non-honors equivalent course:	My institution does not offer a non-honors equivalent
Non-honors exemption details:	The Prerequisites are AP Computer Science A and Biology. Semester 2 of the course (Bioinformatics or Computational Biology) is an integrated Biology/CS course.
Prerequisites:	AP Computer Science A (Required) Biology (Required)
Co-requisites:	None
Integrated (Academics / CTE)?	Yes

Grade levels:	11th, 12th
Course learning environment:	Classroom Based

Course Description

Course overview:

The course is the 3rd in a 3-year sequence, preceded by two **JAVA** programming courses: (1) **Computer Science 1** and (2) **AP Computer Science A**.

Semester 1 covers advanced programming topics: **Advanced Data Structures** (The Collections Interface, Lists, Sets, Maps, Iterators), **GUI** (Graphic User Interfaces), **Recursion** and **Binary Search Trees**. Students utilize the advanced data structures in an open-ended project that algorithmically designs a Word Cloud **on the fly** from any parsed text.

Semester 2 is an introduction to **Bioinformatics**, covering three open-ended topics. The second two, **Genome Assembly** and **Sequence Alignment**, have various solutions, each with its own trade-offs:

1. Students write and leverage a simple **DNA-to-protein translation** program to computationally map (**data visualization**) the **Introns and Exons** of Human **Betaglobin**. Students search the NCBI database for a gene of their own interest, map it, and create a Presentation (PPTX or Google-doc), describing the gene, the function of its protein product, and its importance. Students occasionally discover genetic anomalies, such as an exon that is out-of-order.

2. **Genome Assembly**. Students implement standard varieties of (1) **Shortest Common Superstring** (SCS); and (2) **De Bruijn Graphs**.
   As culmination, students implement a custom form of SCS that can operate with improved efficiency on larger sequences in real-time by early pruning of branches that would yield implausible assemblies.

3. **Sequence Alignment**. Local and Global variations are implemented.

The culminating comprehensive project is a more sophisticated algorithm for Sequence Alignment called **Affine Gap Penalty**, that uses concepts from units in both semesters.

Course content:

Unit 1. Advanced Data Structures: The Collections Interface, Lists, Sets, Maps, Iterators

Unit 1 closely follows **Chapter 11** (Java **Collections** Interface) of the textbook BJP (Building Java Programs, 2nd edition).

Section 1 takes another look at the **List** interface, comparing its implementations **LinkedLists** and **ArrayLists**. It introduces **iterators** for traversing any **Collection** implementation. Iterators allow not only read access, but the ability to change, remove or insert items into a list while it is being traversed.

Section 2 covers the **Set** interface, specifically its implementations **HashSet** and **TreeSet**. The former stores elements in no particular order, the latter in **sorted** order due to its internal implementation using a **binary search tree**.

Section 3 covers the **Map** interface, again looking at its implementations **HashMap** and **TreeMap**, whose properties mirror that of **HashSet** and **TreeSet**. Maps are equivalent to the Python data structure **Dictionary** - both are **tables** with a list of **keys** and their corresponding **values**.

💻 **Unit Assignment(s):**

Word Cloud Construction. There are several steps to this open-ended program.

1. Using the data structures they've studied (LinkedLists, HashSets and TreeMaps), students write a program/method that will read in and parse any text file. and return a **map** (table) of its most frequently occurring words. Students will conflate words that are the same, but differ only by case, or by syntactic inflection - students will handle only simple cases, as they learn how complicated this task can quickly become were they to try to handle all exceptional cases that the language presents.

2. Students pass the **map** to a word cloud class that - on the fly - will design and draw a word cloud one word at a time, so that the process can be observed and better understood. They will use the arithmetic-spiral algorithm developed by the author or **Wordle**, found in the discussion in Chapter 3 of the supplementary textbook "Beautiful Visualizations" (http://downtownmagnets.org/ourpages/users/srp4379/JADSBINF/Wordle_Feinberg.pdf (http://downtownmagnets.org/ourpages/users/srp4379/JADSBINF/Wordle_Feinberg.pdf)). However, prior to applying the algorithm, students must make several design decisions: (a) Deciding how to vary the size of words depending upon their frequency; and (b) calculating the approximate area that the words will take up in the window, and then adding a cushion.

3. Finally, students attempt to improve the efficiency of their program, as they observe how the application slows down as it tries to place words in a drawing space that is increasingly full.

Unit 2: GUI (Graphic User Interfaces)

Unit 2 closely follows BJP **Chapter 14**. Students learn about Components: JOptionPanes, JFrames, JPanels, JButtons, JTextFields and JLabels. A frame (or sub-element) will implement the **ActionListener** interface, which requires implementation of the **event-handler actionPerformed**(). Various **layout managers**, for layout of the elements in a frame, are considered. Additional components, including JTextAreas and JScrollPanes are introduced, as well as Icons (properties) and Fonts. The

MouseInputListener interface can be implemented, allowing the application to respond to mouse events - clicked, dragged, moved, pressed, released, and so on. Finally, students study how to draw in JPanels via the **paintComponent**() method.

💻 **Unit Assignment(s):**

Students complete Exercises 1-9 at the end of the chapter, giving them relatively simple practice in writing programs that utilize GUI components and respond to keystroke and mouse input. They then write programs to implement the four suggested projects that follow: (1) a Notepad-like editor; (2) a graphics Hangman game; (3) a graphics maze through which a "rat" can crawl through and escape, using various algorithms for escaping; and (4) a Paint-like program.

Unit 3: Recursion

Recursion (which was not covered extensively in AP Computer Science A) is defined as a method which calls itself. The technique is suited to easily and concisely - but not particularly efficiently (per machine time) - solve certain problems that are not amenable to iterative solutions. It is an altogether different way of thinking about how to solve problems computationally other than what is normally meant as *computational thinking* (sequentially-arranged statements). Recursion involves solving a portion of a problem and then ***internally*** calling the same method to solve the rest of the problem. A recursive method contains two parts: (a) the **base case**; and (b) the **recursive call**. The base case (when written successfully) is a condition that prevents the method from infinitely recursing. The unit gives students several weeks of practice in solving a variety of problems of progressively increasing difficulty.

💻 **Unit Assignment(s):**

The website **codingbat.com** (Stanford University CS: cs.stanford.edu) contains 2 Recursion sections. The difficulty level of the 30 problems in **Recursion-1** ranges from easy to mid-level. The difficulty level of the 8 problems in **Recursion-2** is HIGH. All in all, they give students a variety of problems whose solutions comprise a range of recursive algorithmic solutions. Students complete these practice problems, then share solutions and problem-solving strategies in a whole-class setting. (Note: Solutions to all codingbat problems can easily be located on the Internet.)

The summative assessment, however, is a page of recursive problems ***written by the instructor*** using the **authoring** feature of the website for writing custom problems (these solutions are **NOT** available on the Internet).

Unit 4: Binary Search Trees

Unit 4 closely follows BJP **Chapter 17**. Students learn that Binary Trees (**BT**s) are linked structures, like LinkedLists, composed of **Nodes** arranged in a tightly-constrained **Tree** structure with exactly two branches. Students learn the recursive traversal algorithms: **inorder**, **preorder** and **postorder,** and the value/uses of each. They learn the recursive algorithm for building ***sequential trees***, and common operations (count nodes/leaves and levels).

Students then turn their attention to Binary ***Search*** Trees (**BST**s), which have similarities to the **Binary Search algorithm** they studied in APCS-A.

Finally, they learn **advanced** tree operations: (a) **balancing** unbalanced trees - *unbalanced* defined as the levels of left and right branches differing by more than 2; (b) **searching** trees and **big O** behavior (efficiency) values(); (c) and **inserting** and **deleting** nodes.

💻 **Unit Assignment(s):**

Students complete exercises 1-19 at the end of the chapter, all of which perform some sort of specific operation on a BT or BST. Students implement several projects:

(a) Huffman encoding and compression using a binary tree. Students test it by reading a file, and converting its characters to a compressed bit format, and saving as a file. Students then read in the compressed file, and decode it back to its original form;

(b) A similar program that converts characters to Morse code, and back;

(c) A **calculator** that evaluates (nested) numeric expressions, in which the subtrees of a node that contains an **operator**, contain **operands**; and

(d) Adding an **Iterator** to a BST. The Iterator will implement the **hasNext()**, **next()** and **remove()** methods, and will require that students modify the BST structure to implement movement **UP** to parent nodes (much like a doubly-linked list can move forward and backward).

Unit 5: Bioinformatics - Mapping Introns and Exons

Students build a simple **DNA-to-protein direct translation program** (circumventing the RNA intermediate forms), using a Java **Map** data structure to hold the **Codon : Amino-Acid Translation Table**. In a traditional CS course, this is generally as far as these programs are ever developed. However, the program can be leveraged / expanded to map the **exons** (protein coding regions) and **introns** (non-coding regions) of a **gene** as follows. Translations of the gene are done for **all three frameshifts**, for a total of **3 possible translation sequences**. Searches of these sequences to ever increasingly shortened substrings of the gene protein eventually will result in a match, and **exons** can be identified one-by-one. Note that intervening introns are not necessarily multiples of 3; i.e. subsequent exons may be identified in any of the three frameshifts in that same direction.

Student Facing Instructions and **Java Source code** for the unit/project can be accessed at:
http://downtownmagnets.org/ourpages/users/srp4379/CS3SW/Exons/Assignment/Exons.htm
(http://downtownmagnets.org/ourpages/users/srp4379/CS3SW/Exons/Assignment/Exons.html)

💻 **Unit Assignment(s):**

Students implement the algorithm described above, and test it on a sample gene (e.g. human hemoglobin) to generate a list of exons. The **class** for each exon **object** includes its **starting position** and **length**, and (optionally) a **method** for generating the corresponding sub-sequence. Students implement a method to visualize the **map** of the exons (and introns) relative to the original DNA gene. Crucially, the output for the printed map will be **formatted** for ease of reading and understanding, and will contain a parameter for line length. Examples can be seen using the hyperlink below:

https://downtownmagnets.org/ourpages/users/srp4379/CS3SW/Exons/index.html (https://downtownmagnets.org/ourpages/users/srp4379/CS3SW/Exons/index.html)

Students will visit **NCBI** (the National Center for Biotechnology Information) and search the **Nucleotide/Gene** and **Protein** databases for a **gene / protein pair** that has multiple exons (i.e. more than one). They will input these sequences into their program, generate an exon / intron map of their gene, and include it in a PPTX or Google-doc Presentation about the importance of the gene and the function of its protein. The link below demonstrates one of these presentations.

https://downtownmagnets.org/ourpages/users/srp4379/CS3SW/Exons/DopamineReceptor.html (https://downtownmagnets.org/ourpages/users/srp4379/CS3SW/Exons/DopamineReceptor.html)

Unit 6: Bioinformatics - Genome Assembly

The unit considers several approaches - each with its own particular trade-offs - to the **Genome Assembly** (also known as **Sequence Reconstruction**) problem. Students listen to the sequence of 13 online Johns Hopkins lectures **Algorithms for Assembly** (https://www.coursera.org/learn/dna-sequencing/lecture/GaIMi/module-4-introduction (https://www.coursera.org/learn/dna-sequencing/lecture/GaIMi/module-4-introduction)). The lectures cover SCS (Shortest Common Superstring algorithm), the problem of repeating DNA sequences, DeBruijn Graphs, and Eulerian walks.

Students create a program to generate random sequences of DNA, which are cut into a list of overlapping fragments. Using this program as a base, they then implement 4 different ways to reassemble the fragments:

(1) A **naive** algorithm. Find the two fragments with the largest overlap, combine them, delete them from the list, and add back the new combined fragments. Repeat until only 1 fragment remains - the completed assembly. The speed in real-time is quick, but the error rate is significant as occasionally two fragments may be combined in error when the overlap for an incorrect fragment is >= the overlap for the correct fragment. When this happens, it often becomes impossible to combine the last two fragments or so. If not, and all fragments can be used, this generates an assembly that is longer than the original sequence.

(2) **SCS**. Generate all of the possible permutations of fragment combinations. Real-time speed varies, exponentially increasing with the number of fragments (N Factorial). The most common permutation methods do not allow pruning. An original variant permutation method, however, does allow earlier pruning, greatly improving performance. The trade-off with SCS is that it cannot accurately handle **repeats**.

(3) **DeBruijn Graphs**. Students implement the DeBruijn Graph and Eulerian walk algorithms.
The DeBruijn Graph algorithm does handle repeats, but ambiguities - particularly when there is more than one Eulerian walk - limit its utility.

💻 **Unit Assignment(s):**

Students implement two versions of SCS, using different forms of the key method **permutate()**. This method can generate all of the possible permutations from a list of fragments. Students first implement the Steinhaus–Johnson–Trotter (https://en.wikipedia.org/wiki/Steinhaus%E2%80%93Johnson%E2%80%93Trotter_algorithm) algorithm, but when testing realize that it quickly gets bogged down as N increases. When students work out mathematically that the number of permutations is N factorial (**1!** = 1; **2!** = 2; **3!** = 6; **4!** = 24; **5!** = 120; **6!** = 720; **7!** = 5 ,040; **8!** = 40,320; **9!** = 362,880; **10!** = 3,628,800; **11!** = 39,916,800; **12!** = 479,001,600; etc.), they are asked to think about some way to reduce the number of permutations generated.

Students implement an **original alternate version of permutate()** that allows one to eliminate (prune) early on whole branches of combined fragments using as a criterion a minimum overlap value. This allows the program to process much larger fragment lists in real-time. The method is described below:

> Algorithm: N = # of elements in input list.
> **Property that allows pruning**: Elements appended to each list are placed in their **FINAL** positions!
>
> Iteration 1 (I1): Generate N lists size=1, each containing one of the elements of the input list
>
> Iteration 2 (I2): For each I1 list, generate N-1 lists, size=2; to each is appended one element from the input list
> that has not already been used in that list AND whose overlap with the final element is >= minOverlap
>
> Iteration 3: For each I2 list, generate N-2 lists, size = 3; to each is appended one element from the input list
> that has not already been used in that list AND whose overlap with the final element is >= minOverlap, etc.

Unit 7: Bioinformatics - Sequence Alignment

The unit closely follows the discussion in Chapter 6, pp. 147-226, of the supplementary textbook **Introduction to Bioinformatics Algorithms** (Jones, Pevzner). The main focus is on two algorithms for Sequence Alignment: (a) Local Alignment and (b) Global Alignment. However, students learn how sequence comparison can have important ramifications, such as inferring/discovering the function of a newly discovered gene (particularly for one related to a disease) based on its resemblance to a gene whose function is already known. Sequence comparison can also be used to inform evolutionary relationships (the field of phylogenomics).

Students study the "Manhattan Tourist" problem, which serves as a model for how to think about the basic sequence alignment algorithm, i.e. finding the longest path between 2 vertices in a directed acyclic graph (DAG). Alignments use **scoring matrices** - either **PAM** (Point Accepted Mutations) or **BLOSUM** (Block

Substitution), based on the likelihood of mutating - as a criterion for identical and near matches between amino acids. An alignment matrix can then be used to build an "edit-distance" sequence, which details how one parsimoniously converts one sequence to the other and vice-versa.

💻 **Unit Assignment(s):**

Students implement both the **Local** and **Global** Alignment algorithms in several stages.

(1) They write a method to (a) prompt the user (using an **Open File Dialog** box) for the file path/name of a **Blosum** scoring matrix, (b) read in and parse the file, and (c) place the data into a **Map**. They then write a method - given two amino acids - to query the Map and return a Blosum score. Finally, to check that the Blosum table was read in and processed correctly, students write a method to print out the matrix using the Map-query method.

(2) Students write a method to allocate the 2-D **Alignment Matrix**, with one protein sequence across the top and the other protein sequence down the left side, and fill it from top to bottom, left to right. They allocate a duplicate matrix, but fill it with direction characters, indicating whether the value in each cell was calculated using either (a) the cell diagonally to the upper-left, (b) the cell directly above, or (c) the cell directly to the left. (**LCS** algorithm, p. 176 top).

(3) Students implement an **iterative** version of the (recursive) traceback method (**printLCS** algorithm, p. 176 bottom) to print out the edit-distance sequence. It will be formatted in three lines, with line 1 being one of the protein sequences, line 2 being the edit-distance sequence, and line 3 the other protein sequence.

Honors Final Exam Details:

The culminating comprehensive project for the course is to write a much more sophisticated and nuanced **Sequence Alignment** program called the **Affine Gap Penalty**, which corrects a serious deficiency in the alignment algorithms previously studied in Unit 7, namely, how to handle substantial gaps between two otherwise homologous sequences. The algorithmic model resembles a 3-D chess board (see **Figure 1** immediately below). The algorithm is described in Chapter 6, pp. 184-187, of the supplementary textbook **Introduction to Bioinformatics Algorithms** (Jones, Pevzner)

Affine Gap **Figure 1** (http://www.downtownmagnets.org/ourpages/users/srp4379/pubs/UCOP/AffineGap.png)

The alignment algorithm begins in the middle level. One traverses a gap in either sequence/strand by jumping to either the level above or below, per **Figure 1**. A jump to the "**vertical**" matrix corresponds to a gap in the sequence along the left **column** of the alignment matrix, and a jump to the "**horizontal**" matrix corresponds to a gap in the sequence along the top **row** of the matrix. Once a gap has been traversed, the alignment process jumps back to the middle level.

The project is **comprehensive** in that it uses select concepts and skills studied throughout the year-long course: (1) advanced data structures, (2) a GUI interface for selecting input data (sequences and parameters, including an Open File Dialog to select files containing the data) and output of results, and (3) combinations of algorithmic strategies.

Students will test their programs by performing an alignment of a **pre-mRNA gene** (composed of exons and introns) with its **mature mRNA** counterpart (from which introns have been spliced out), revisiting the examples they studied in Unit 5 (Mapping Exons). Students will run the program by systematically using a host of different values for the gap penalties. When these penalty parameters are properly selected/optimized, the program will print a map of a gene's introns and exons - a real-world application.

Course Materials

Textbooks

Title	Author	Publisher	Edition	Website	Primary
Building Java Programs	Reges, Stuart; Stepp, Marty	Addison-Wesley Publishing Company , USA	2nd/2010	http://www.buildingjavaprograms.com/	Yes

Other

Title	Authors	Date	Course material type	Website
Bioinformatics for Biologists	Pevzner, Pavel; Shamir, Ron	Cambridge University Press 2011	Supplementary Textbook	[empty]
An Introduction to Bioinformatics Algorithms	Jones, Neil C; Pevzner, Pavel A.	MIT Press, 2004	Supplementary Textbook	[empty]
Beautiful Visualization, Chapter 3 (Wordle)	Feinberg, Jonathan	O'Reilly, 2010	Chapter in Supplementary Textbook	http://static.mrfeinberg.com/bv_ch03.pdf
Introduction to Algorithms	Cormen, T; Leiserson C; Rivest R; Stein C	3rd Ed / MIT Press 2009	Supplementary Textbook	[empty]

Additional Information

Course Author:

Scott Portnoff
Teacher
srp4379@lausd.net
3236271152 ext.