Guidelines

Programming Project 2 is a laboratory that you complete on your own, or in collaboration with one partner, but without the help of others. It is a form of take-home exam. You may consult your text, your notes, your lab work, our on-line examples, and the course web pages, but no other sources for code are permitted. You are encouraged to reuse the code from your labs or our class examples, however. While you may discuss this project with the course instructors or your programming project partner, you may not discuss it with TAs, tutors, classmates, friends, etc. The use of any outside help or sources is a violation of the Honor Code.

You may work with one partner on this project. Your partner can be anyone in your class, even someone from a different lab section.

Project Options. You have your choice of two programs to implement:

- RushHour: This is an objectdraw version of the classic game. It builds on our graphics-based approach to learning programming. While covering everything we have learned all semester, it emphasizes concepts from the second half of the semester, including one- and two-dimensional arrays, recursion, and reading from files.

- Sequence Alignment: This program also exercises everything you have learned, but in a very different context — that of a protein sequence alignment algorithm from the field of Computational Biology. Sequence alignment is central to DNA reconstruction, gene identification, determining the phylogeny (or evolutionary history) of species, and so on. While we have focused on graphics in many of our examples and labs, the programming skills learned in this class transfer to many other domains as well, and this project is geared for those who may wish to explore a different type of programming problem altogether.

You may choose either program as your final project. While each has its own particular challenges, we have attempted to make them as similar in scope and complexity as we could. Both are also extensible in many, many ways, and we encourage you to explore and have fun with them!

Due Dates and Grading. You will submit your work in two parts. The first is a design of the program you choose to implement, and the second is the code itself:

- **Design Due:** Monday, 20 November, in class
- **Due:** Thursday, 7 December, 11 PM.

The projects will be graded out of 100 points, with roughly half the points given for correctness and half the points given for style, organization, efficiency, etc. Thus, even imperfect programs can receive close to full credit, but poor design and style can lead to quite low final scores. A more detailed breakdown appears at the end of each program option.

Submitting Your Work

Design. Your design should be either neatly written or typed, and it should be turned in on paper at the beginning of class. We will return the design to you on the Monday following Thanksgiving Break. Keep a copy of your design if you plan to work on the program over the break. If you are working with a partner, turn in one design with both of your IDs on it.
Code. Once you have saved your work in BlueJ, please perform the following steps to submit your assignment. Make sure that your ID appears in the title of your project folder. (Make sure that both IDs appear in the title of the project folder if you are working with a partner.)

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder. You can do this by clicking on the smiling Macintosh icon in your dock.

- From the “Go” menu at the top of the screen, select “Connect to Server...”.

- For the server address, type in “afp://Guest@fuji” and click “Connect”.

- A dialog box will pop up. Connect as a guest by selecting the “Guest” option.

- A selection box should appear. Select “Courses” and click “OK”.

- You should now see a Finder window with a “cs134” folder. Open this folder by double clicking on it.

- You should now see the drop-off folders for the three lab sections. Drag your “Project2IDnnnn” folder into the appropriate lab section drop-off folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click “OK”.

- Log off of the computer before you leave.
Rush Hour is a popular and addictive “sliding block” puzzle game, where you start with a parking lot with a number of cars parked both horizontally and vertically. The player’s task is to move the cars forward and backward within the lot to get the “goal” car out of the lot.

For Programming Project 2, you will write a version of “RushHour.” We have simplified the game somewhat to make the assignment more manageable. You can find a working version of our game on the handouts page of our course website.

If you want to experience the actual board game, we have it and will bring it to lab.

Our RushHour game presents the player with one game set up to play. Several cars and trucks are placed vertically or horizontally in a square parking lot grid that is six units tall and six units wide. One of these is the red “target” car – the one the player needs to move to the “exit” grid square. Other cars are obstacles that make this task more challenging. Horizontally-oriented cars are allowed to move left and right, so long as they do not run into a grid square occupied by another car or run off the boundary of the grid. Vertically-oriented cars may move only up and down. Cars always occupy exactly two or three grid cells, and they must always be aligned exactly within the grid cells they occupy.

The image below shows the “Novice 2” game configuration, which includes four cars and two trucks. The target red car is the leftmost vertically oriented car.

The program allows the player to use the mouse to move cars around the parking grid until the target car reaches the grid square adjacent to the exit. The image on the left below shows the “Novice 2” game after the player has used the mouse to make a few moves. The image on the right shows what happens when the player finally places the target car at the parking lot exit.
The game would be awfully dull if there were only one possible game configuration. Our version of the game therefore provides six options – two novice-level games, two intermediate-level games, and two expert-level games – that the player can select from a menu.

Our version of the game also provides two “Undo” buttons. The first undoes the player’s most recent legal move. Pressing the Undo button repeatedly allows the user to undo moves all the way back to the starting configuration. The “Undo All” button undoes all moves back to the starting configuration, showing each step along the way with short pauses between them.

**Implementation Details**

You should begin the game by setting it up. This involves constructing the parking lot and the two buttons, as well as a JComboBox that allows the player to select a game. You should also set up the vehicle configuration for the game that is the default selection on the JComboBox.

The starter folder contains six files representing the game configurations that appear in our demo. Those files are called novice1.txt, novice2.txt, intermediate1.txt, intermediate2.txt, expert1.txt, and expert2.txt. Each line of a game configuration file represents a vehicle to be placed in the parking lot as follows:

- `image-file column row length orientation targetCar?`

For example, the intermediate1.txt file begins with the following line:

```
4.gif 0 2 3 HORIZONTAL false
```

meaning that a car looking like the image in 4.gif should be placed in column 0 and row 2. Because it is horizontal, it will extend to the right. In the end, it will cover 3 grid spaces as shown on the next page. This vehicle is not the target red car.
In the starter folder you'll find all of the image files whose names appear in the game configuration files.

The parking lot is visually a grid. We constructed our parking lot on the canvas out of FramedRects. Our grid starts at Location (10, 10), and each grid square is $30 \times 30$ pixels. Note that we refer to the parking lot grid squares by column and row values. Though there is a relationship between the grid's column and row numbers and the canvas's $x$ and $y$ locations, they are not the same. Our grid position $(1, 0)$ – i.e., column 1 and row 0 – has its upper left corner at canvas Location $(40, 10)$. Your parking lot should know how to convert grid positions (columns and rows) into their corresponding canvas locations and vice versa. The parking lot should also know which of its spots are occupied and which aren’t, so that when moves are made, vehicles cannot run over each other.

Each vehicle on the grid has a visual representation – a VisibleImage perhaps? – but it also has behavior that is somewhat more interesting than a VisibleImage alone. It should be able to move, but only horizontally or vertically, depending upon the way it is oriented. The user should be able to “pick one up” – i.e., remove it from the parking lot – and “place it” in a new spot by moving it. A vehicle should also know whether it is the special red vehicle.

There will be many vehicles in a game of RushHour, so you will need a way to keep track of a collection of them.

The player will be able to move the vehicles with the mouse. If the player presses the mouse on a car or truck that is horizontal, the player can drag it from left to right, provided there are no other vehicles in the way. (Hint: Consider the way the paddle could be moved horizontally.) Similarly for vertical vehicles. When the user releases the mouse, the vehicle should snap into position. A vehicle that is intended to take up two spaces should take up no more than two; a vehicle that is intended to take up three spaces should take up no more than three.

Each time the player makes a move in the game, you will need to record it so that it can potentially be “undone” by the player later. Some RushHour games are quite complex; the player might make many moves before finding a solution. Therefore, a recursive list is a good choice for helping you remember the moves that are made.

The player can undo one move at a time, or the player can undo everything. This can be accomplished with an ActiveObject that plays the moves back to the beginning of the game.

Your program will be comprised of several classes, corresponding to the objects just described.

**RushHour** This is your WindowController class. It should set up the game and respond to mouse and GUI component events.

**ParkingLot** This class represents the actual grid. It should draw the picture of the parking lot. It should keep track of which grid spaces are currently occupied. You might also find it useful to have this class be able to convert back and forth between mouse Locations and parking grid coordinates.

**Car** You should have an instance of this class for each car in the parking lot. (Note that we’re calling all of the vehicles Cars.) A car should be responsible for drawing itself and being able to move around. A car should know its orientation (horizontal or vertical), should know if it is the target
car or not, and should know which parking lot spaces it currently occupies. When you construct a car, you should also specify its length as either 2 or 3.

**CarCollection**  This class maintains a list of all of the cars currently in the parking lot.

**Position**  This very simple class represents positions in the parking lot grid – i.e., it provides a way to represent (col, row) coordinate pairs, where col and row are integer values. We have provided the implementation of this class for you.

**MoveInfo**  This class represents the information about a single move (i.e, placement of a car in a new grid position). It should include any information you will need to implement the undo of a move.

**MoveListInterface, EmptyMoveList, NonEmptyMoveList**  These classes are used to maintain the complete history of all moves made (but not yet undone) in the game so far.

**MovePlayer**  This class is an ActiveObject that animates the “Undo All” functionality. It should undo each move in a move list, with a short pause between each step.

You may also want to define other classes if you believe they will simplify your design.

---

**The Design**

As indicated in the heading of this document, you will need to turn in a design for your Rush Hour program well before the program itself. This design should be a clear and concise description of your planned organization of the program.

You should include in your design a sketch of each class including the types and names of all instance variables you plan to use, and the headers of all methods you expect to write. You should write a brief description of the purpose/function of each instance variable and method.

In addition, you should provide pseudocode for any method whose implementation is at all complicated. In particular, if a method is complicated enough that it will invoke other methods you write (rather than invoking methods provided by Java or by our library), then include pseudocode for the method so that we will see how you expect to use your own methods.

From your design, we should be able to find the answers to questions like the following easily:

1. What information is passed to the constructor for Cars, Moves, etc.?
2. How do you prevent the player from moving a Car on top of another one?
3. How do you determine which Car, if any, the user is selecting with the mouse?
4. How will you determine whether the player has won?
5. What will you need to re-set when the player chooses to start a new game?

The more time you spend on the design, the easier it will be to complete the program. Also, we will grade and return the designs to you on the Monday after Thanksgiving. We will be able to give much more useful feedback and comments for designs that are well-written and complete.

---

**Constants**

The following constants appear in the starter code. In the Car class:

```java
// public constants that represent the orientation of the car
public static final int HORIZONTAL = 1,
    VERTICAL = 2;
```

In the CarCollection class:

```java
// maximum number of cars in a RushHour game
private static final int MAX_CARS = 16;
```
In the `ParkingLot` class:

```java
// the number of columns and rows in the parking lot grid
private static final int COLS = 6;
private static final int ROWS = 6;

// the x and y coordinates of the grid’s upper right corner
private static final int START_X = 10;
private static final int START_Y = 10;

// the width and height of a grid square
private static final int SPACE_SIZE = 30;

// the Position (col, row) of the "exit" spot
private static final Position goal = new Position(2, 0);
```

In the `MovePlayer` class:

```java
// number of milliseconds to wait between "undo move" animations
private int DELAY = 500;
```

Feel free to adjust these as you wish. However, our game configuration files assume that the “exit” position will always be (2, 0).

---

**Implementation Order**

Begin by downloading the starter project from the handouts web page. We strongly encourage you to proceed as suggested below to ensure that you can turn in a running program. While a partial program will not receive full credit, a program that does not run at all generally receives a lower grade. Moreover, it is easier to debug a program if you know that some parts do run correctly.

- Experiment with the demonstration program.
- Implement the drawing of the parking grid in your `ParkingLot` class. Test it by creating a new `ParkingLot` in the `begin()` method of the `RushHour` class. **If your images do not appear, ensure there are no spaces in the directory path containing your images.**
- Implement a `Car` with limited functionality. Make sure a car can draw itself appropriately in the parking lot.
- Add functionality to the `Car` class so that horizontal cars can move horizontally and vertical cars can move vertically. Remember that cars should not be able to run over other cars, nor should they be able to move outside the parking lot. Test this by implementing the appropriate mouse event-handling methods in the `RushHour` controller class.
- Implement “winning” the game by checking whether the target car has been moved adjacent to the exit after each move of the target car.
- Add the `JComboBox` menu that allows the player to select from multiple game configurations. Our demo includes six options. You should include at least three. When the player selects a configuration, you will need to clear your parking lot and remove all existing cars.
- Add the “Undo” and “Undo All” buttons.
- Implement a simple version of the “Undo” functionality. Remember just the most recent move made, and test whether you can undo it.
- Next implement the full “Undo” functionality. This will involve remembering each move you make in a `MoveList`.  

---

7
• Now add the “Undo All” functionality. This will require that you track each move using the three `MoveList` classes. When you have that structure implemented and tested functionality, then you can create a `MoveListAnimator` that can take your MoveLists and undo it, with pauses inserted between moves.

• At this point, you can safely begin to add your “bells and whistles” if you so desire.

There is a great deal of functionality to aim for in this programming project. **Do not worry if you cannot implement all of the functionality.** Get as much of it working as you can. As we have done throughout the semester, we will consider both issues of correctness and issues of style when grading your program. It is always best to have full functionality, but you are better off having most of the functionality and a beautifully organized program than all of the functionality with a program that is sloppy, poorly commented, etc.

**Extra Credit.** There are many additional features you could add to your program. We will give 1-2 points for each extension, for a maximum of 6 points extra credit. Some possible extensions are:

• Add more game configuration files.

• Add a button that allows the player to select the game configuration file they’d like to use.

• Improve the look of the parking lot, use better images, or add sound effects.

• Implement a timer to tell the player how long it took them to solve each puzzle.
Grading Guidelines

Points will be assigned roughly as follows:

**Design (14 pts)**

- Plausibility
- Instance variable and constant names and types
- Method signatures
- English descriptions
- Pseudocode for complex methods

**Style (44 pts)**

- **Presentation (14 pts)**
  - Descriptive and helpful comments
  - Good names
  - Good use of constants
  - Appropriate formatting
  - Appropriate use of public/private

- **Programming (15 pts)**
  - Proper use of boolean conditions
  - Proper use of if/while/for-loops
  - Proper use of variables
  - Proper use of parameters
  - Appropriate selection of arrays or recursive data structures
  - Efficiency issues

- **Organization (15 pts)**
  - Appropriate methods for each class
  - Appropriate parameters for each method
  - Appropriate instance variables and constants

**Correctness (42 pts)**

- **Setup (12 pts)**
  - Parking grid drawn correctly
  - Designated exit
  - One car drawn correctly
  - All cars in a game configuration drawn correctly
  - Two buttons
  - Game choice menu

- **Car Movement (10 pts)**
  - Horizontal cars move horizontally
  - Vertical cars move vertically
  - Cars don’t run over other cars
  - Cars don’t run off lot
  - Car fully in grid locations at end of move
• Winning the game (4 pts)
  Win detected
  Win message displayed

• Selection of new game (8 pts)
  Menu allows selection
  At least three choices
  Old game erased and forgotten
  New game set up correctly

• Undo (8 pts)
  Can undo most recent move
  Can undo series of moves
  Appropriate behavior when nothing left to undo
  Animated undo all

Extra Credit (up to 6 pts)
As our friends in the biology department can tell you, sequence alignment is the process of arranging DNA or protein sequences to identify regions of similarity. This information can then be used for several purposes, including providing clues about the evolutionary relationships between organisms. For Programming Project 2, you may choose to implement a sequence-alignment program.* You can find a working version of our program on the handouts page of the course web site (although that version will only be able to load one specific file of sequences to align.) In our demo, you can enter sequences in the text fields, select similarity metrics for determining alignments, compute alignments, and display them.

As you can see, our program’s interface provides the user with several ways to explore sequences of amino acids:

- The user may enter two sequences in the text fields at the top of the window. If the user clicks “Align Pair”, the two sequences are aligned, and their alignment is displayed on the canvas in the center of the window. We’ll describe a specific algorithm – the Smith-Waterman algorithm – for sequence alignment in great detail below.

- The user may select the particular similarity scheme to be used in performing the alignment. Similarity measures for sequence alignment will be discussed in more detail below.

---

*In designing this project, we were inspired by an assignment developed by Prof. Lisa Meeden at Swarthmore, for her introductory Python programming course. The sections on Cytochrome c and alignment scoring in this document borrow very heavily from her assignment. We are also grateful to Prof. Meeden for providing the “fake” data set as well as the Cytochrome c data set.
Once an alignment is displayed on the canvas, the user can zoom in and out, using the “plus” and “minus” buttons.

The user may also choose to load a text file that contains a list of sequences. These can then be compared in individual pairs by choosing the desired inputs from the menus. The resulting alignments are displayed on the canvas as before. Or the user may choose to compare all of the sequences to each other.

Protein Alignment: Cytochrome c

Though the Smith-Waterman sequence-alignment algorithm will allow you to align any pair of amino acid sequences, we’ll provide you with data for a particular protein: Cytochrome c. Cytochrome c is a highly conserved protein, which means that the amino acid sequence of the protein is very similar across different organisms. For example, the start of the amino acid sequence of Cytochrome c for a human is

GDVEKGGKIFIMKCSQCHTVEKGGKHKTGPNLHGLFGRKRTGQAPGYSYT...

while for a dromedary camel, it is

GDVEKGGKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKRTGQAVGFSYT...

In the Starter folder for the project, we have provided a file of Cytochrome c data. Each line of the file consists of three pieces of space-separated information: a binomial name (e.g., “homo sapiens”), a common name (e.g., “human”), and the amino acid sequence for the Cytochrome c protein for that organism. The last three (partial) lines of the file look like:

cryctolagus_cuniculus european_rabbit GDVEKGGKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRK...
capra_hircus goat GDVEKGGKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKRTGQAAGFS...
equus_burchellii burchell’s_zebra GDVEKGGKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRK...

Scoring an Alignment

There are many algorithms for sequence alignment. For this project, we would like you to implement the Smith-Waterman algorithm. Before describing the algorithm, we’ll begin by discussing what makes a good sequence alignment. Suppose there is a very small imaginary protein protA produced by three animals: a mouse, a fly, and a bee. You would like to compare protA’s amino acid sequence as found in each of these animals to figure out which two animals have the most similar sequences. Here are the amino acid sequences of each animal’s protA:

mouse : AGDVEK
fly : AGWVEK
bee : AGFVEK

You can see that the mouse, fly, and bee all have five amino acids in common and one that is different. If we wanted to compute a score for the similarity of each pair of sequences, a simple scoring metric might be the following: each amino-acid match adds 2 to the pair’s match score, and each mismatch adds -1. Thus the score for the mouse/fly match would be 9, as would the matches for mouse/bee and fly/bee.

However, aspartic acid (Asp/D) has a small, polar side chain, and tryptophan (Trp/W) has a large non-polar side chain. Because of the structural differences of these two amino acids, it is likely that substituting an Asp for a Trp would have an impact on the functionality of the protein. On the other hand, both Trp and Phenylalanine (Phe/F) have large, non-polar side chains terminating in a carbon ring. Therefore, it is likely that substituting a Phe for a Trp would have less of an impact on the protein’s function than substituting an Asp for a Trp or Phe would have.
By studying the variances in proteins across species, biochemists have developed metrics that assign a numeric value to particular amino acid substitutions. For this project, we will use the BLOSUM80 substitution matrix (see the course handouts web site). With this matrix, we can now score the differences between any two sequences. We do this by computing the sum of the scores of all the substitutions necessary to transform one sequence into the other sequence. So for the mouse and fly sequences, we compute the score as follows:

mouse : A G D V E K
fly : A G W V E K
score:  5  6 -6  4  6  5  --> total = 5+6-6+4+6+5 = 20

The score for the fly/bee alignment is 26; the score for the mouse/bee alignment is 22. From this, we determine that the fly and bee are most similar (with a score of 26). We also see that the mouse is more similar to the bee (with a score of 22) than the fly (with a score of 20).

Of course, not all amino acid sequences align as neatly as our imaginary protA sequences. Consider another hypothetical (and very short) protein produced by three animals:

horse: GDVAK
pig: AGDVA
cow: PAGDAER

How can we score the similarities of pairs of sequences when the lengths of those sequences differ?

Let’s begin by introducing the notion of a gap. Whenever one sequence has an amino acid and the other does not, the alignment will incur a gap penalty. For this project, we’d like you to define a gap penalty to be -1. For example:

horse: - - G D V A - K
cow: P A G D - A E R
gives a total score of −1 + −1 + 6 + 6 + −1 + 5 + −1 + 2, which is 15. And

horse: - - G D V A K _
cow: P A G D - A E R
gives a score of −1 + −1 + 6 + 6 + −1 + 5 + 1 + −1, which is 14.

The particular sequence-alignment algorithm we will have you implement will consider all possible alignments between a pair of strings. It will select the alignment with the highest score.

The Smith-Waterman Algorithm

Smith and Waterman described their algorithm for sequence alignment in the Journal of Molecular Biology in 1981. The algorithm is an example of a general technique called dynamic programming, and it is guaranteed to find the optimal local alignment with respect to the scoring system being used. The scoring system varies slightly from the one we described above. It does not allow scores of any subsequences to be negative. If, in the process of determining an alignment, a particular subsequence pairing has a negative score, that score is reset to 0 to indicate “no similarity”. In the remainder of this section, we describe the implementation of the Smith-Waterman algorithm.

- You will need to construct two 2-dimensional arrays: one for the subsequence match scores and another to track the way in which the subsequence scores are computed. Let’s call the first one \( h \) and the second one \( direction \).

Both arrays should be of the same size. Their width (i.e., the number of columns) should be one more than the length of the first sequence to be aligned, which we call \( a \) below. Their height (i.e., the number of rows) should be one more than the length of the second sequence to be aligned, which we call \( b \) below.

The score array, \( h \), should be an array of int. We suggest that the direction array be an array of int as well. There will be three directions you need to account for: “diagonal”, “up”, and “left”. You might give these values of 1, 2, and 3, respectively.
• All of the entries in the first row and the first column of the \( h \) array should be 0. All of the entries in the first row of \( \text{direction} \) should be “left”, and all of the entries in the first column of \( \text{direction} \) should be “up”. The entry at \( \text{direction}[0][0] \) is never used and can be left empty (or set to some default value different from the three direction constants).

• Now you’re ready to do the main work of considering all possible alignments. The entries in the \( h \) array are set as follows. Starting at the upper left of the array,

\[
h[i][j] = \max\left(0, \right.
\begin{align*}
&h[i-1][j-1] + \text{similarity}(a.\text{charAt}(i-1), b.\text{charAt}(j-1)), \\
&h[i-1][j] + \text{gapPenalty}, \\
&h[i][j-1] + \text{gapPenalty}
\end{align*}
\]

where \( a \) is the first sequence to be aligned and \( b \) is the second. That is, \( h[i][j] \) is set to the maximum value of the four specified expressions. The similarity is determined by the similarity scheme (for instance, the BLOSUM80 matrix). **If the maximum value is 0, then there is no meaningful alignment for \( a \) and \( b \). See our notes in step 7 in the “Implementation Order” section below for ideas on how to handle this. Otherwise...**

– If the maximum is \( h[i-1][j-1] \), then the value of \( \text{direction}[i][j] \) is “diagonal”.
– If the maximum is \( h[i-1][j] \), then the value of \( \text{direction}[i][j] \) is “left”.
– If the maximum is \( h[i][j-1] \), then the value of \( \text{direction}[i][j] \) is “up”.

Note that these indicate the direction in \( h \) from which \( h[i][j] \) was derived.

Suppose we wish to align the following two strings:

```
ACACACTA  AGCACACA
```

The algorithm constructs the following \( h \) and \( \text{direction} \) arrays:

\[
\begin{array}{cccccccccccc}
- & A & C & A & C & A & T & A \\
- & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\
G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\
A & 0 & 2 & 2 & 5 & 4 & 3 & 4 \\
C & 0 & 1 & 4 & 7 & 6 & 7 & 6 & 5 \\
A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\
C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\
A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
- & A & C & A & C & A & T & A \\
- & - & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow \\
A & \uparrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow \\
G & \uparrow & \uparrow & \leftarrow & \leftarrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
C & \uparrow & \uparrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow \\
A & \uparrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
C & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
A & \uparrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow \\
\end{array}
\]

As described above, the first row and column contain special default values. Any other entry \( h[i][j] \) contains the optimal score for any alignment in which the letter at index \( i-1 \) in \( a \) is aligned with the letter at index \( j-1 \) in \( b \). For example, the value 2 at \( h[3][3] \) indicates that 2 is the score of the best alignment possible for the prefixes ACA and AGC of the two sequences, respectively. The \( \text{direction} \) array records information that would allow us to reconstruct that optimal alignment, as we describe below.

• Once \( h \) is completely filled in, the highest number in the array gives the score of the best alignment of the sequences. Find the location (i.e., the row and column) of the largest value in \( h \). In the example above, the maximum score is 12 in the bottom right corner of the array.
Next you need to “trace back” over the path that led to the maximum value. To do this, construct a new one-dimensional array, which will hold direction information. The first element of this array should be the direction (from the direction array) corresponding to the row and column found in the preceding step.

- If the direction is “diagonal”, then the next row and column should each be one less than the current.
- If the direction is “left”, then the next column value should be one less than the current, but the row value should stay the same.
- If the direction is “up”, then the next row value should be one less than the current row, but the column value should stay the same.

The next entry in the “traceback” array should now be the direction (again from the direction array) corresponding to the newly computed row and column.

Proceed in this manner until you reach the top-left corner of the table. The “trace back” array for our example is the following:

\[
\begin{array}{cccccccc}
\text{←} & \text{↖} & \text{↖} & \text{↖} & \text{↖} & \text{↖} & \text{↖} & \text{↑} & \text{↖} \\
\end{array}
\]

This backtrace corresponds to the grayed boxes in the direction table above.

You can now begin to construct the aligned versions of the input sequences. To do this, you’ll use the “traceback” array you just constructed. The directions in the backtrace tell us whether

- the best alignment matches up the next pair of amino acids in \(a\) and \(b\) (a “diagonal” entry),
- the best alignment includes a gap in sequence \(a\) (an “up” entry), or
- the best alignment includes a gap in sequence \(b\) (a “left” entry).

Starting at the end of our backtrace, we see that the first entry is a “diagonal”, so the first amino acids of \(a\) and \(b\) are aligned:

\[
a: A \\
b: A
\]

The “up” entry then indicates that the best alignment requires a gap in \(a\) in order for the sequences to align well:

\[
a: A - \\
b: A G
\]

The next five entries in the backtrace are “diagonal”, indicating that the next five amino acids of the sequences line up:

\[
a: A - C A C A C \\
b: A G C A C A C
\]

The next entry in the backtrace is “left”, indicating that the best alignment requires a gap in \(b\):

\[
a: A - C A C A C T \\
b: A G C A C A C -
\]

Finally, we have a “diagonal” entry:

\[
a: A - C A C A C T A \\
b: A G C A C A C - A
\]
• You’re almost done now! It’s possible that the aligned versions of $a$ and $b$ from the preceding step are only partial. You might have lost a suffix of one sequence or the other if the algorithm did not exactly align the last character of $a$ with the last character of $b$. (This occurs when the maximum value in $h$ is not in the lower-right hand corner.) Be sure to insert any missing suffixes back in to the aligned sequences.

---

**Implementation Details**

You should begin by setting up the interface. Don't worry too much at the outset about the details of the layout. You can make cosmetic improvements later.

**Sequence-entry and alignment panel.** In the north of the window, you should place two text fields, as well as a button that the user can click when they want to align the sequences in the text fields.

**Display area control panel.** In the west of the window, you should place three buttons: one that will allow the user to “zoom in” on an alignment displayed on the canvas in the center of the window; one that will allow the user to “zoom out”; and one that will allow the user to clear the canvas.

**Similarity scheme selection panel.** In the east of the window, you should place two buttons: one to allow the user to choose a “simple” default scheme for determining the similarity of individual amino acids in the sequences to be compared; and another button that allows the user to choose a more complex similarity scheme.

**File-oriented sequence alignment panel.** In the south of the window, you should place two JComboBox menus, as well as three buttons. One button will allow the user to select a file of organisms and sequences to be aligned. After such a file is selected, the menus should be made to contain the names of all of the organisms in the files for which sequences can be compared. Note that you will need to add a listener to each of the buttons, but you will not need listeners for the text fields or the menus.

The user should be able to select a scheme for determining similarity of amino acids. You will implement two classes, representing two distinct schemes. They are called SimpleSimilarityScheme and Blosum80. Both implement the SimilaritySchemeInterface interface.

If the user types two sequences into the text fields at the top of the window, your program should construct a new Aligner. An Aligner expects three parameters: the two sequences to be aligned and a similarity scheme. It should provide a method to compute an alignment, as well as methods to return the score of the selected alignment and the aligned sequences themselves.

When an alignment of two individual sequences is computed, it should be displayed on the canvas at the center of the window.

In addition, the user might choose to load organisms from a file. An organism file will have the format described above in the section on Cytochrome c. We have provided two such files in the starter folder: one with real Cytochrome c data, and a very short file with “fake” data on which you might do your initial testing. When the user selects a file, your program should use the information in the file to populate the two menus with the organisms’ common names. It should also construct a collection of organisms that can then be used to construct a matrix of all pairwise organism similarities.

Your program will be comprised of several classes, corresponding to the objects just described.

**SequenceAlignmentController** The controller will set up the user interface. It will also respond when the user clicks on the various buttons in the interface. (You will probably include an instance variable in this class to keep track of the similarity scheme to use when performing alignment. Our default similarity scheme is initialized to be a SimpleSimilarityScheme.)
**SimpleSimilarityScheme** A SimpleSimilarityScheme provides the information necessary to compute the similarity between individual amino acids. If two amino acids are the same, their similarity value should be taken to be 2. If they are different, their similarity should be -1. The gap penalty for this scheme should also be -1.

**Blosum80** Like SimpleSimilarityScheme, this provides a particular similarity metric. We have implemented this class for you.

**Aligner** An Aligner provides the methods necessary to perform an alignment and to retrieve the corresponding alignment scores and aligned sequences. A new Aligner should be constructed for each new alignment.

**Organism** An Organism is an object that describes the binomial and common names of an organism, as well as a particular protein sequence for that organism.

**OrganismCollection** An OrganismCollection describes a collection of organisms. It should provide methods to add an organism to the collection, find an organism in the collection, and so on.

**OrganismScoreMatrix** An OrganismScoreMatrix will do the work of taking a full collection of organisms and computing a complete set of pairwise alignments for that collection. Once a complete matrix of similarities has been computed, it should be able to return for each organism, a String that describes its most closely related other organisms, beginning with the closest relative and ending with the most distant relative.

---

**The Design**

As indicated at the beginning of this document, you will need to turn in a design plan for your Sequence Alignment program well before the program itself. You should include in your design a sketch of each class including the types and names of all instance variables you plan to use, and the headers of all methods you expect to write. You should write a brief description of the purpose/function of each instance variable and method.

In addition, you should provide pseudo-code for any method whose implementation is at all complicated. In particular, if a method is complicated enough that it will invoke other methods you write (rather than just invoking methods provided by Java or our library), then include pseudo-code for the method so that we will see how you expect to use your own methods. It will be especially important that you provide pseudo-code for the Smith-Waterman alignment method.

From your design, we should be able to find the answers to questions like the following easily:

1. What information is passed to the constructor for an Organism, OrganismCollection, Aligner, etc?

2. How do you handle the user's clicks on all of the various buttons in the window?

3. Once an alignment is complete, how will you get the score that was computed? How will you get the aligned sequences?

---

**Implementation Order**

Begin by downloading the starter project from the handouts web page. We strongly encourage you to proceed as suggested below to ensure that you can turn in a running program. While a partial program will not receive full credit, a program that does not run at all generally receives a lower grade. Moreover it is easier to debug a program if you know that some parts do run correctly.

1. Experiment with the demonstration program.
2. Write a program that constructs a window with the appropriate user interface. Don’t worry too much about layout at this point. If you’d like to add labels to your panels, as we have in our demo, you can include something like the following line of code. It adds a label to a JPanel named panel, as seen in our version of the program:

```java
panel.setBorder(BorderFactory.createTitledBorder("Fancy Label For Panel"));```

3. Add code to construct two text objects on the canvas, representing two sequences of amino acids. Write the code to handle zooming in on those sequences, zooming out, and clearing the canvas.

4. Write the SimpleSimilarityScheme class. Test it by implementing the code to handle the case where the user clicks the button to select the scheme. Once it’s constructed, test the `getSimilarity` and `getGapScore` methods.

5. Next write the code to read an organism file and populate the menus.

6. Next add to the previous code the ability to construct a collection out of the organisms in the file.

7. Now implement the Aligner. Follow the Smith-Waterman algorithm description very closely. Start out by testing on a sequence pair for which you know the desired answer. [Hint: Try the two sequences we used as examples in our Smith-Waterman algorithm description.] After each major step of the algorithm, print the relevant arrays to be sure you’re calculating values correctly.

Then test your aligner on some short sequences, such as those in the “fake” data file provided in the starter. You can also test your code on some longer sequences, such as:

```
GAATTCAGTTA
GGATCGA
```

aligned (with both similarity schemes):
```
G A A T T C A G T T A
G G A - T C - G - - A
```

or

```
GCGCATGGATTGACGA
TGCGCCATTGATGACCA
```

aligned (with the BLOSUM80 scheme):
```
- G C G - C A T G G A T T G A - G C G A
T G C G C C A T T G A - T G A C - C - A
```

or

```
PAWHEAE
HEAGAWGHEE
```

aligned (with both similarity schemes):
```
- - - P A W - H E A E
H E A G A W G H E - E
```

**Implementation Notes:** There are a number of subtleties in how this algorithm works, particularly when it comes to deciding what alignment to use when there is a tie for the best. Below are three suggestions that should guarantee your implementation behaves the same as ours in that regard:
(a) When computing $h[i][j]$, the maximum may not be unique among the choices. In that case break the tie as follows:
   • The diagonal direction should be chosen over up or left.
   • Left should be chosen over up.

(b) When computing $h[i][j]$, if you arrive at a maximum value of zero because all of the other options yielded negative values, the corresponding direction should be the diagonal direction.

(c) You should set all the values in the left-most column of $h$ and top-most row of $h$ to their default values and never recompute them during alignment.

(d) After filling in $h$, you will search for the largest number in that array to determine the best alignment. Again, there may be ties. To match our implementation, select the right-most, bottom-most maximum value in the case of a tie.

8. In the previous step you were probably testing the algorithm by typing sequences into the text fields. Now try selecting two sequences from the menus and aligning those.

9. Your very last step should be to work on the OrganismScoreMatrix class.

Extensions

There are many additional features you could add to your program. We will give 1-2 points for each extension, for a maximum of 6 points extra credit. Some possible extensions are:

• Allow a user to display the original sequence for an item in one of the menus.

• Replace our “similarity scheme” selection buttons with a menu of similarity scheme options.

• Use the information in the OrganismScoreMatrix to suggest a possible evolutionary tree for a set of organisms.

• You already allow the user to zoom in and out. Consider creating a magnifying glass feature. When passed over the sequences on the canvas, it would enlarge or otherwise highlight the selected areas.

• Provide text fields to allow a user to create their own simple similarity scheme.
Grading Guidelines

Points will be assigned roughly as follows:

**Design (14 pts)**
- Plausibility
- Instance variable and constant names and types
- Method signatures
- English descriptions
- Pseudocode for complex methods

**Style (44 pts)**
- **Presentation (14 pts)**
  - Descriptive and helpful comments
  - Good names
  - Good use of constants
  - Appropriate formatting
  - Appropriate use of public/private
- **Programming (15 pts)**
  - Proper use of boolean conditions
  - Proper use of ifs/whiles/for-loops
  - Proper use of variables
  - Proper use of parameters
  - Appropriate selection and use of arrays or recursive data structures
  - Efficiency issues
- **Organization (15 pts)**
  - Appropriate methods for each class
  - Appropriate parameters for each method
  - Appropriate instance variables and constants

**Correctness (42 pts)**
- **Setup and File Loading (12 pts)**
  - north panel components constructed correctly
  - south panel components constructed correctly
  - east panel components constructed correctly
  - west panel components constructed correctly
  - choice menus populated properly when file selected
  - organism collection populated properly when file selected
- **Display Control (6 pts)**
  - Text item on canvas grows larger when zooming in
  - Text item grows smaller when zooming out
  - canvas is cleared appropriately when clear button is clicked
- **Similarity Schemes (8 pts)**
Correct similarity scheme is constructed upon user click and used until next one selected
Similarity scheme returns correct similarity when amino acids the same
Similarity scheme returns correct similarity when amino acids are different
Similarity scheme returns correct gap penalty

• Aligner (12 pts)
  
  best alignment computed correctly
  correct alignment score returned
  correctly aligned strings returned
  correctly aligned strings displayed on canvas

• Score Matrix (4 pts)
  
  alignments computed for all pairs
  closest relatives computed for an organism
  displays all closest relatives on the canvas

Extra Credit (up to 6 pts)