Programming Project 2

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 web pages associated with the course web page, 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:

- *Centipede*: This is an objectdraw version of the classic Arcade 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 and collections.
- *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, 21 November, in class
Due:	Thursday, 8 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. 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 names on it.

Code. Once you have saved your work in BlueJ, please perform the following steps to submit your assignment. Make sure that your name appears in the title of your project folder. (Make sure that both names appear in the title of the project folder if you are working with a partner.)

- 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 selection box should appear. Select "Courses" and click "Ok".
- You should now see a Finder window with a "cs134" folder. Open this folder.
- You should now see the drop-off folders for the three lab sections. Drag your "Project2LastNames" folder into the appropriate lab section 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.

Centipede

Centipede—with its colorful mushroom patch, tenacious centipedes, bouncing spiders, fleas and scorpions—was one of the earliest and most popular video arcade games. It was brought to life by Atari in 1981. A fully restored game unit from those early days is currently for sale on EBay at over two thousand dollars. (Sadly, the Dean denied our request for discretionary funds to place a bid...)

For Programming Project 2, you will write a version of "Centipede." We have simplified the game somewhat to make the assignment more manageable. You can find a working version of our game at the following URL. (You may notice some screen flashing when you play — this is because web browsers do not always take advantage of graphics card hardware features for mitigating this sort of flashing when running Java programs. Your own program will not flash.)

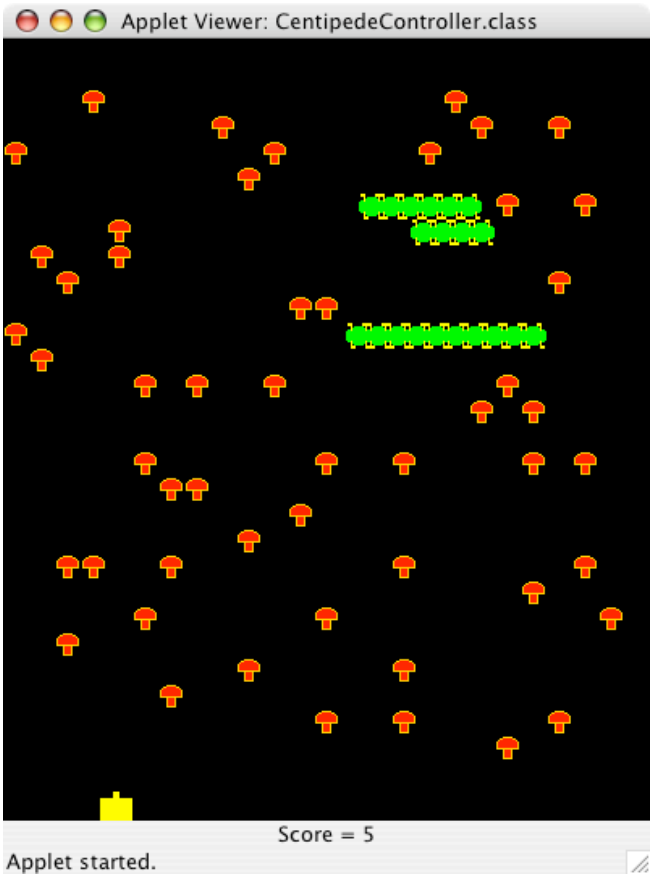
```
http://www.cs.williams.edu/~cs134/handouts.html
```

If you want to experience the original arcade version, visit

```
http://my.ign.com/atari/centipede
```

or for a modern take on the original, visit

```
http://www.atari.com/arcade#!/arcade/centipede/play
```



Our simplified game begins with a centipede at the top of a field of mushrooms. A bug "zapper" sits at the bottom of the screen. The zapper's job is to defend itself against the attacking centipede.

The centipede is actually made up of a bunch of individual segments. Each segment moves horizontally across the screen, and a segment turns around and moves down a row of mushrooms toward the zapper whenever it runs into the edge of the screen or a mushroom. The segments all move at a fixed speed, but each segment keeps track of whether it is moving to the left or the right. At the start of the game, all of the segments are adjacent to each other and heading in the same direction. Thus, they appear to be a single centipede. As the game progresses, the segments may head in different directions and no longer appear as a single centipede.

The zapper moves left and right when the player presses the left and right arrow keys. The zapper shoots a missile when the player hits the space-bar. If a missile hits a mushroom, the mushroom disappears and the player earns one point. If a missile hits a centipede segment, the player earns 5 points. Also, the segment that was shot disappears and a mushroom appears in its place. The centipede moves its remaining segments, as before. If a segment in the middle of the "centipede" is hit, the segments between the new mushroom and the "tail" of the centipede will run into the newly created mushroom and turn around. Thus, the net effect is that there will now appear to be two "smaller centipedes" headed toward the zapper.

The game ends either when the zapper shoots all of the segments or when a segment runs into the zapper. In either case, a message is displayed to the user indicating that the game is over and showing the player's final score.

Implementation Details

You should begin the game by setting it up. This involves creating a black background, a scorekeeping mechanism, a zapper, mushrooms, and all those centipede segments. We provide some suggestions on how to design and implement all of these pieces of the program below.

The starter folder includes six mushroom images and two segment images. You only need to use one of each. ("shroom2.gif" is our personal favorite.) Feel free to use the others if you want to add a little variety or animation to your program. Also, feel free to edit and include your own images, although it will be easiest if you stick to images that are 16x16 pixels.

The scorekeeper should display the score at the bottom of the screen. You must be able to increase the score when a missile hits a segment or a mushroom.

The zapper appears at the bottom of the screen. It provides methods for responding to the player's key strokes. Thus, a zapper should have methods to move left, move right, and shoot a missile. If the zapper is hit by a centipede segment, it should stop being able to move or fire. If you want, you can make the zapper disappear in some interesting way.

The missiles shot by the zapper are active objects. They should move up the screen and stop when they reach the top, hit a mushroom, or hit a segment.

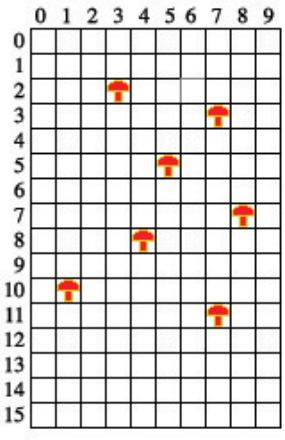
A centipede is an active object that keeps a collection of segments inside it. Each segment can be thought of as a separate entity that will keep track of which direction it is moving. The segments are all initially placed next to each other and facing to the right, so that they appear to move as a single, long "centipede". The centipede animates the segments by making each segment take a step. (The segments themselves are NOT active objects.) A segment takes a step by moving a fixed distance in the direction that that segment is heading. The centipede tells each of its segments to take a step, and should only pause after all of the segments are moved. *Do not pause between moving individual segments*. If you do, the centipede may not appear to move properly- it will stretch out or shrink, or individual segments may move out of alignment and not look quite right.

If a segment is about to step off the screen or hit a mushroom, it should move down the screen by one row of mushrooms. The segment should also change its direction so that it will move in the opposite direction the next time it is asked to take a step. When a segment is hit by a missile, it should disappear from the screen.

You need to be careful about how you keep track of the segments in the centipede. We suggest you use an array of `Segment` objects. Most importantly, DO NOT TRY TO DELETE SEGMENTS THAT HAVE BEEN HIT FROM THE ARRAY AND SHIFT OTHER ELEMENTS IN THE ARRAY OVER TO FILL THE HOLE. If you do this, bad things may happen if a missile tries to rearrange your array to delete a segment that has been hit at the same time that the segment is being moved across the

screen. Instead, when a segment is hit, just set a variable within the object representing the segment to indicate that it is dead and hide it from the screen.

As you can see from the demo, mushrooms never overlap and are placed inside the squares of a grid laid over the canvas. We suggest using a two dimensional array of `VisibleImages` to store the mushrooms. Make the array large enough to hold mushrooms laid out over the whole canvas. For reasons similar to those described above for segments, you should fill all entries in your array with mushroom visible images. Only show images for mushrooms that should be visible, and NEVER DELETE ELEMENTS FROM THE ARRAY OR SET ENTRIES TO NULL. Intuitively, if there is enough room for a grid of mushrooms with 16 rows and 10 columns in the canvas, you would create a 16x10 array and store a mushroom in each cell. The following shows how the screen would look if that grid were displayed while all but 7 mushrooms were hidden:



To "remove" a mushroom that is hit by a missile, simply use the `hide` method to hide the mushroom visible image. The `isHidden()` method may be handy for determining if a specific mushroom is visible on the screen. When a missile hits a centipede segment, you should make the mushroom in the grid location underneath the segment appear after hiding the segment.

Your program will comprise several classes, corresponding to the objects just described.

CentipedeController The controller will set up the game. It will also accept user input in the form of key strokes. In response to the different key presses, it should invoke methods of the zapper, making it move or shoot. We have provided the skeleton for listening to the user's key presses. You should set up the game in `begin` and fill in the lines where the zapper's methods need to be invoked.

Zapper The `Zapper` moves in response to each key press of the left and right arrow keys and should have `moveLeft` and `moveRight` methods to support these operations. When the space-bar is pressed, the `CentipedeController` will invoke the `Zapper`'s `shoot` method to launch a missile.

Segment A `Segment` manages one segment of the centipede, and describes its behavior. It should keep track of the direction it is traveling and record whether or not it is still alive. A segment can take a `step`, and it should be able to `die`. The `step` method should determine how far and in which direction the segment should move, and know how to turn around when it runs into the edge of the screen or into a mushroom. It should also "kill" the zapper if the segment's image overlaps the zapper.

Centipede A `Centipede` is an `ActiveObject` that keeps an array of `Segments`. Its `run` method will make each segment take a `step`.

Field The `Field` class holds the two-dimensional array of mushrooms. The `Field` constructor should create all of the mushrooms, and the class should provide methods that make it easy to implement the interactions between mushrooms and segments, and mushrooms and missiles.

Missile A `Missile` is an `ActiveObject` that moves up the screen, stopping either when it reaches the top or when it hits something. Note that to achieve this behavior, the missile needs to know about the `Field` and `Centipede`, so that it can determine whether it hits a mushroom or segment, respectively.

The easiest way to provide this information to the `Missile` is to have the `Zapper`'s `shoot` method take the `Field` and `Centipede` as parameters, and then pass them to the `Missile`'s constructor when that method creates the new missile.

(Alternatively, you could store the field and centipede in instance variables in the zapper, but this is a little tricky to set up because the centipede will also need to store the zapper so that its segments will be able to kill the zapper. This circularity between the zapper and the centipede can be resolved in a number of ways, but just passing the necessary information to `shoot` is the easiest.)

ScoreKeeper The `ScoreKeeper` class displays the score on the screen. Note that the `Missile`s should probably know about the `ScoreKeeper`, as they will likely need to inform it to increase when they hit a segment or mushroom.

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 plan for your Centipede 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.

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

1. How and when is the scorekeeper updated?
2. What information is passed to the constructor for `Missile`s, `Segment`s, etc.?
3. How do you find the mushroom appearing at a certain location on the canvas in the `Field`'s two-dimensional array? (You will need to use this operation in several places.)
4. How does a `Segment` decide when to turn around? What methods does the `Field` provide so that the segment can decide whether it is about to step into a mushroom?
5. What operations do the `Centipede` and the `Field` provide to help `Missile`s determine when they hit and kill segments and mushrooms?

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. Using them will ensure that your segments move at a reasonable speed and are able to turn correctly.

In the `Field` class:

```
// dimensions of the canvas
public static final int CANVAS_WIDTH = 400;
public static final int CANVAS_HEIGHT = 512;

// size of mushroom pictures
private static final int SHROOM_SIZE = 16;

// dimensions of the mushroom array
private static final int NUM_ROWS = CANVAS_HEIGHT / SHROOM_SIZE;
private static final int NUM_COLS = CANVAS_WIDTH / SHROOM_SIZE;
```

In the `Segment` class:

```
// width of segment image
private static final int SEGMENT_WIDTH = 16;

// number of pixels a segment should travel in the X direction
private static final int SEGMENT_STEP_X = 4;

// number of pixels a segment should move in the Y direction
private static final int SEGMENT_STEP_Y = 16;
```

In the `Centipede` class:

```
// offset between segments when first created.
private static final int SEGMENT_OFFSET_X = 12;

// pause time between moving all of the segments
private static final int CENTIPEDE_PAUSE = 25;
```

Feel free to adjust these constants as you wish. However, `CANVAS_WIDTH`, `SHROOM_SIZE`, and `SEGMENT_OFFSET_X` should always be multiples of `SEGMENT_STEP_X`. If they are not, your centipedes may appear to separate or bunch up after running into obstacles, because some segments will turn around when they are a pixel or two further away from the obstacle than others.

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.
- Write a program that draws a `Zapper`. Run the program in a window that is 400x512 pixels in size. (You will need to click in the window when it first opens.)
- Add code to make the `Zapper` respond to the user's right- and left-arrow key strokes. Do not worry about shooting at this point.
- The next step is to add the mushrooms by creating a `Field` in `begin`. Implement the `Field` constructor to create a two-dimensional array of `VisibleImage`s. Initialize all entries with mushrooms, hide them all, and then randomly select perhaps 60 or 70 of them to show. They should not appear in the top few rows or bottom few rows so that they do not interfere with the centipede as it starts to move or with the zapper.

- Next, start working on `Segment` and `Centipede`. Implement a basic `Segment` class that provides a `step` method that moves a segment's image a little to the right. Write a `Centipede` that creates a single segment and makes it `step` across the screen. (It will work best if you create the centipede in `begin` after creating the field and zapper).

- Change the `Segment` to step to the left and right properly, moving down the height of a mushroom when it reaches the edge of the canvas. Do not worry about running into mushrooms yet. Change the `Centipede` to create and move a whole array of `Segment`s. The x coordinates of consecutive segments should be `SEGMENT_OFFSET_X` pixels apart. Our centipede is created off the screen and gradually steps into view, but this is not required– you can simply create the segments along the top of the canvas.

- Make sure you are passing the mushroom field into the `Centipede` and `Segment` classes, and modify the `Segment` class to turn around when it is about to step into a visible mushroom. The `Segment` will need to ask the `Field` whether a specific (x,y) location on the canvas corresponds to a visible mushroom in your 2D array. We suggest you declare this method in `Field` as follows:

  ```
  public boolean shroomIsVisible(double x, double y)
  ```

  You may implement this test in any reasonable way. You may find two helper methods that we have provided in `Field` for converting screen locations to array indices useful:

  ```
  // Convert a y coordinate in pixels to the corresponding
  //   row in the mushroom array
  private int getRow(double y) {
      return (int)(y / SHROOM_SIZE);
  }

  // Convert a x coordinate in pixels to the corresponding
  //   column in the mushroom array
  private int getColumn(double x) {
      return (int)(x / SHROOM_SIZE);
  }
  ```

  For example, `getColumn` will convert the x coordinate 50 to the column index 3, and `getRow` will convert the y coordinate 100 to the row index 6. Thus, the mushroom image corresponding to the screen location (50,100) on the screen is stored in your 2D array at row 6 and column 3.

  Note that these two methods may return rows and columns that are not within the bounds of the array if x and y are negative or larger than `CANVAS_WIDTH` and `CANVAS_HEIGHT`. So, be sure to check that the x and y coordinates passed into `shroomIsVisible` are within bounds before proceeding. If they are not, we can conclude that no mushroom is visible at that location.

- Change the zapper to shoot missiles. Make them move up to the top and disappear.

- Make the missiles hit segments. First, be sure to pass the centipede to the `Missile` constructor when you create it. The missile shot at the segments should, as it is moving, be continually asking the centipede "Have I hit and killed any of the segments that are still alive?" If the missile does hit a segment, make it disappear. (Do not worry about hitting mushrooms or making them appear yet.)

- Change the `Missile` class so that, in addition to asking the `Centipede` if it overlaps any living `Segment`, it asks the `Field` "Have I hit a visible mushroom?" If the missile does hit a mushroom, it should disappear. A single missile should not be able to kill both a mushroom and a segment, even if it hits them at the same time. So, be sure to check if a mushroom was hit only if no segments were hit.

- Modify the `Segment` so that the mushroom under it becomes visible when the segment becomes hidden. Note that once you add this feature, killing a segment may make some of the segments overlap or completely cover each other, depending on how you implemented moving and turning. This is perfectly fine, and in fact makes the game a little more interesting to play...
- Lastly, change `Segment` to kill the `Zapper` when it runs into it, and set up the score keeper.

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. Since we have deliberately left out many features of the original Centipede game, there are clearly 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:

- Restrict the zapper to shooting only one missile at a time.
- Make the mouse control the zapper.
- Permit the zapper to move up and down in the bottom few rows of the screen. The `Zapper` can then avoid segments that reach the bottom. Such segments should "teleport" several rows back up the screen and continue to move.
- Make it require multiple shots to kill a mushroom.
- Reincarnate the centipede by moving it back up to the top when the segments have all been killed.
- Make the centipede's head look different than the segments. The starter provides additional images for this. If the centipede splits, how do you make both smaller centipedes have heads?
- If you play the original game, fleas drop down the screen creating new mushrooms, spiders run around near the zapper trying to sting it, and scorpions poison some mushrooms. Feel free to implement any of these other creatures. (A very simple addition would be to just randomly select mushrooms to appear occasionally, even if you do not implement the whole flea.)
- Use multiple images for the centipede segment to animate its motion.

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 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 (10 pts)
  one segment drawn correctly
  whole centipede drawn correctly
  one mushroom drawn correctly
  random mushrooms drawn correctly
  initial score or message correct

- Zapper (8 pts)

  zapper moves left, right
  zapper shoots missile
  zapper cannot move off screen

- Missiles (8 pts)

missile moves up
missile stops at top
missile can hit segments
missile can hit mushrooms
segments turn into mushrooms

- Segments and Centipede (10 pts)

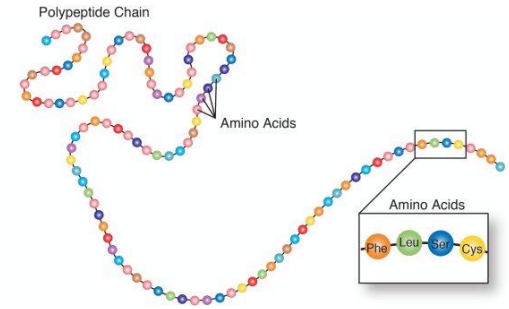
  segment moves right or left
  segment turns around at edges
  segment turns around at mushrooms
  segment moves down when it turns
  array of segments move correctly

- End of Game (6 pts)

  Game ends when all segments are hit
  Game ends when zapper is hit
  Scorekeeper keeps score correctly

Extra Credit (up to 6 pts)

Sequence Alignment

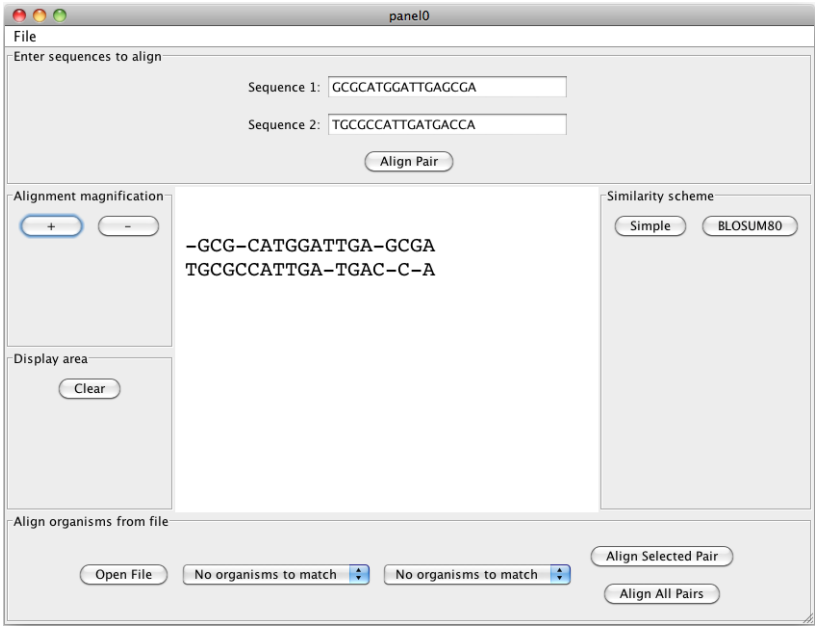


Amino Acids

Ala: Alanine	Gln: Glutamine	Leu: Leucine	Ser: Serine
Arg: Arginine	Glu: Glutamic acid	Lys: Lysine	Thr: Threonine
Asn: Asparagine	Gly: Glycine	Met: Methionine	Trp: Tryptophane
Asp:Aspartic acid	His: Histidine	Phe: Phenylalanine	Tyr: Tyrosisne
Cys:Cysteine	Ile: Isoleucine	Pro: Proline	Val: Valine

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.

The image below shows our program's user interface.



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

```
GDVEKGKKIFIMKCSQCHTVEKGGKHKTGPNLHGLFGRKTGQAPGYSYT...
```

while for a dromedary camel, it is

```
GDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKTGQAVGFSYT...
```

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:

```
oryctolagus_cuniculus european_rabbit GDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRK...
capra_hircus goat GDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKTGQAAGFS...
equus_burchellii burchell's_zebra GDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRK...
```

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 `direction` should be "left", and all of the entries in the first column of `direction` should be "up". The entry at `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( 0,
               h[i-1][j-1] + similarity(a.charAt(i-1), b.charAt(j-1)),
               h[i-1][j] + gapPenalty,
               h[i][j-1] + gapPenalty)
```

  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 `direction[i][j]` is "diagonal".
  - If the maximum is `h[i-1][j]`, then the value of `direction[i][j]` is "left".
  - If the maximum is `h[i][j-1]`, then the value of `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 `direction` arrays:

`h`:

	–	A	C	A	C	A	C	T	A
–	0	0	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	5	4	3	4
C	0	1	4	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

`direction`:

	–	A	C	A	C	A	C	T	A
–	–	←	←	←	←	←	←	←	←
A	↑	↖	←	↖	←	↖	←	←	↖
G	↑	↑	↖	↑	↖	↑	↖	↖	↑
C	↑	↑	↖	←	↖	←	↖	←	←
A	↑	↖	↑	↖	←	↖	←	←	↖
C	↑	↑	↖	↑	↖	←	↖	←	←
A	↑	↖	↑	↖	↑	↖	←	←	↖
C	↑	↑	↖	↑	↖	↑	↖	←	←
A	↑	↖	↑	↖	↑	↖	↑	↖	↖

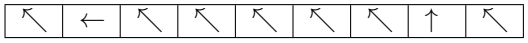
  As described above, the first row and column contain special default values. Any other entry `h[i][j]` contains the optimal score for any alignment in which the letter at index `i-1` in `a` is aligned with the letter at index `j-1` in `b`. For example, the value 2 at `h[3][3]` indicates that 2 is the score of the best alignment possible for the prefixes ACA and AGC of the two sequences, respectively. The `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:

  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), or
  - 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 `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:

   ```
   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

   ```
   GCGCATGGATTGAGCGA
   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)