Handout 12
CSCI 134: Spring, 2015

Programming Project 2

Guidelines

A programming project is a laboratory that you complete on your own, or in collaboration with a partner. 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 use of any other source (human or otherwise) for code is forbidden. You are encouraged to reuse the code from your labs or our class examples. You may not discuss these problems with anyone aside from Steve, Andrea, or your partner,if you are working with one. For example, you may not discuss the programming project with TAs, tutors, or other students in the class. The use of outside help or sources is a violation of the Honor Code.

You may work with a partner on this project. You partner can be anyone in CSCI 134, even someone from a different lab section.

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

- *Space Invaders*: 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 important concepts from the second half of the semester, such as two-dimensional arrays.
- *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 at 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, 4 May, in class
Due:	Thursday, 14 May, 5 PM.

You may not use late days on this project.

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

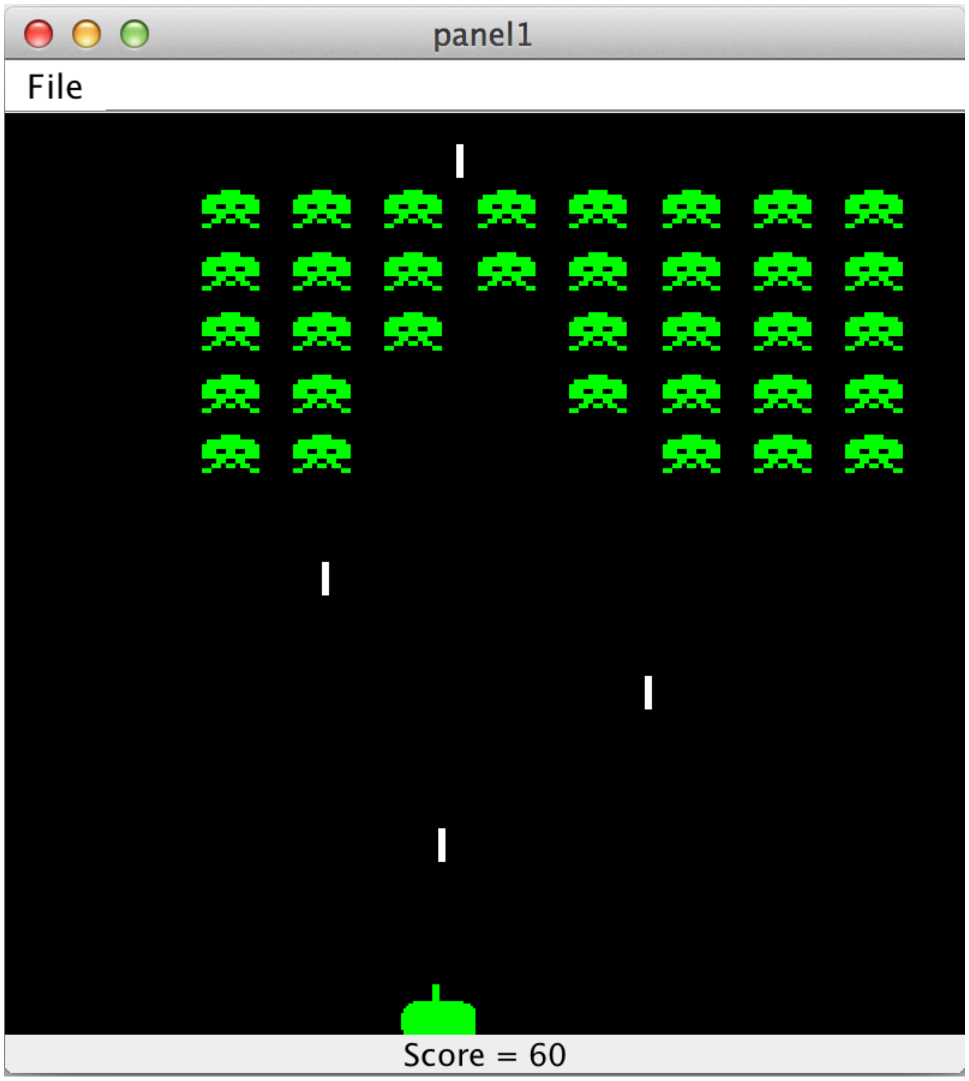
Design. Your design should be either neatly written or typed, and it should be turned in on paper at the beginning of class. We will return the designs to you once we have had a chance to read them all. Keep a copy of your design so that you can work from it while we grade the version you have turned in.

Code. 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 "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 labs sections. As in the past, drag your "Project2LastName" folder into the appropriate lab section folder. (If you are working with a partner, be sure its name includes both last names.) When you do submit your folder, 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.

If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before your lab section's deadline. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like "v2") and the resubmission will work fine.

Space Invaders



Space Invaders has a long and illustrious history, first appearing in video arcades in 1978. By 1980, it had been licensed by Atari and became the first arcade game adapted to Atari's new home video game system.

Your final project will be to write Space Invaders. We have simplified the game somewhat from the original, and you can find a working version of what we have in mind on the course web page. If you want to experience the original version, visit the handouts web page.

The game begins with several rows of aliens at the top of the screen. At the bottom is a lone space ship that must defend the earth from the attacking aliens. The aliens move across the screen from left to right and then back again, occasionally shooting at the good-guy space ship. They move at a constant rate and in sync with each other.

The space ship also moves from left to right, but its movement is controlled by the player. If the player clicks on the right-arrow button on the keyboard, the ship moves to the right; if the player clicks on the left-arrow button, the ship moves the left. In addition to dodging the projectiles shot by the aliens, the ship can shoot back. This is also controlled by the user by clicking the space bar or up arrow.

Each time a defense projectile hits an alien, the player gets 10 points. The game is over either when the player gets all the aliens or when the player is hit. In either case, a message is displayed to the user, indicating "Game Over" and showing the final score.

Implementation Details

You should begin writing the game by setting it up. This involves creating a black sky, a scorekeeping mechanism, a good-guy ship, and all those nasty aliens. The overall look and feel of the game is entirely up to you, as long as it implements the basic behavior outlined below. We have provided image files for the aliens in the starter folder that you may use if you like. (We provide several files in case you want to add variety or a little animation, but you only need to use one of the provided images for the basic game.)

The scorekeeper should display the score at the bottom of the screen. It must to be able to increase the score when an alien is hit.

The good-guy ship is an object that will appear at the bottom of the screen. It must respond to the player's key clicks. That is, it should be able to move to the right and to the left. It must also be able to shoot at aliens. If it is hit, it should stop firing. If you want, you can make it disappear in some interesting way.

The projectiles shot at the aliens will be active objects. They should move up the screen and stop either when they hit an alien or reach the top.

The aliens should move as a group. They move together from left to right; and then they move together from right to left. When an alien is hit by a projectile, it should disappear from the screen. You need to be a bit careful about how you keep track of the aliens. We suggest you use a two dimensional array. Also define a class to represent a single alien. **Don't try to delete aliens 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 projectile tries to rearrange your array to delete an alien that has been hit at the same time that the alien is being moved across the screen. Instead, when an alien is hit, just set a variable within the object that represents the alien to indicate that it is dead and remove it from the screen.

The projectiles shot by the aliens must move down the screen, stopping either when they reach the bottom or when they hit the ship.

To summarize, your program should include the following classes:

SpaceInvaders: The controller will set up the game. It will also accept user input in the form of key clicks. In response to the different key clicks, it should invoke methods of the ship, making it move or shoot. We have provided the skeleton for listening to the user's key clicks. It is your responsibility to set up the game and to fill in the lines where the ship's methods need to be invoked.

ScoreKeeper: The `ScoreKeeper` class displays the score on the screen. Note that the aliens should probably know about the `ScoreKeeper`, as they will likely need to inform it to increase when they've been shot.

SpaceShip: The `SpaceShip` object moves in response to each key press of the left and right arrow keys. When the space key or up arrow key is pressed, it should lauch a "defense projectile." Our spaceship uses a `FilledRoundedRect`. However, any reasonable spaceship representation is fine.

Invaders: The `Invaders` class will extend `ActiveObject`. This object will hold an array of aliens.

Alien: An `Invaders` object keeps track of a bunch of `Alien`s, each with behavior all its own. An alien can move, it can launch a projectile, and it can disappear when shot.

Projectile: `Alien`s shoot projectiles. A `Projectile` is an `ActiveObject` that moves down the screen, stopping either when it reaches the bottom or when it hits the space ship. Note that to achieve this behavior, the projectile needs to know about the space ship. Since projectiles are created by aliens who are members of the group of `Invaders`, the ship must be passed as a parameter through all of these classes.

DefenseProjectile: Last, but not least, are the projectiles shot by the good guy. They move up the screen, stopping either when they reach the top or when they hit an alien.

Consider this: The aliens need to know about the ship so that they can aim for it with their projectiles; but the ship needs to know about the aliens so that it can shoot at them. If this seems circular to you, you're right. You might handle this by creating the ship and passing it as a parameter to the `Invaders` when you create them. Then, after you've created the `Invaders`, you might invoke a method of the ship class (perhaps called `setTarget`), that will pass the `Invaders` as a parameter to the ship.

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 Space Invaders 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 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 `Projectile`s, `Alien`s, etc.?
3. How do you find the `Alien` appearing at a certain location on the canvas?
4. How do the `Invaders` decide when to turn the `Alien`s around?
5. What operations do the `Invaders` provide to help `DefensiveProjectile`s determine when they hit an `Alien`?
6. Which class is responsible for determining that a `Projectile` has hit the ship and for ending the game?

Implementation Order

You will write most of this program from scratch. However, the handouts web page does contain a BlueJ starter project with a file for each class you need to implement as well as image files for the aliens. We have also set up basic code to handle key strokes in the window controller class.

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. (Note that the online demo uses buttons to control the ship's movements and ability to shoot. Your implementation should use key strokes.)
2. Write a program that draws a `SpaceShip` object.

3. Add code to make the `SpaceShip` respond to the user's right- and left-arrow key clicks. Don't worry about shooting at this point.
4. Next construct the aliens. If you don't feel comfortable constructing a two-dimensional array of aliens, start by constructing one row of aliens. If you construct one row of aliens, you should be using a one-dimensional array.
5. Now make the aliens move from left to right and then right to left, etc.
6. Make the aliens shoot at the ship. There are a number of possible ways to do this. Here is one idea: at each time step randomly select an alien to do the shooting.
7. Now make the ship shoot at the aliens. The projectile shot at the aliens should, as it's moving, be asking them "Have I gotten any of you?". Our demo only allows you to shoot one projectile at the aliens at a time, but you do not need to implement this feature.
8. Finally, 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 throughout the semester, we will consider issues of style and design, in addition to correctness, when we grade your final 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 and a sloppy, poorly commented program.

Extensions

Since we have deliberately left out many of the features of the original Space Invaders game, there are clearly many additional features you could add to your program. We will award 1-2 points for each extension, for a maximum of 6 points extra credit. Some possible extensions are:

- Implement continuous, smooth motion for the space ship.
- Only permit the bottom-most alien in each column to shoot.
- Modify the scoring: for each alien in the bottom row, award 10 points; for each in the second row, award 20 points; for each in the third row, award 30 points, etc.
- As aliens near the right and left edges are hit, adjust the motion of the remaining aliens so that they move all the way to the right and left edges of the screen.
- Use multiple images for aliens to animate their 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 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)

- Basic Ship Behavior (9 pts)

  Ship constructed and displayed on canvas
  Ship moves correctly with key presses
  Ship fires projectiles

- Aliens (9 pts)

  Aliens constructed and displayed on canvas
  Aliens move from left to right and back
  Aliens fire projectiles

- Ship Projectiles (9 pts)

  Ship projectile moves up canvas when fired
  Ship projectile stops either when it hits an alien or goes off the canvas
  Ship projectile destroys an alien, when appropriate

- Alien Projectiles (9 pts)

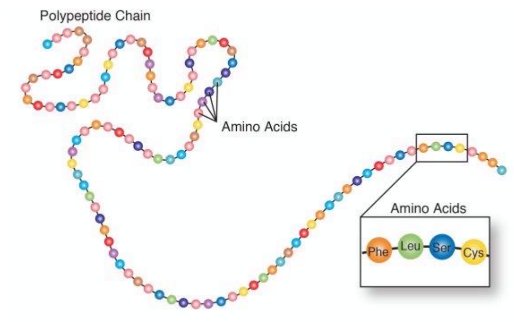
  Alien projectile moves down canvas when fired
  Alien projectile stops either when it hits the ship or goes off the canvas
  Alien projectile destroys ship, when appropriate

- Scorekeeping (6 pts)

  Current score displayed
  Score adjusted and displayed properly when aliens hit

Extra Credit (up to 6 pts)

Sequence Alignment

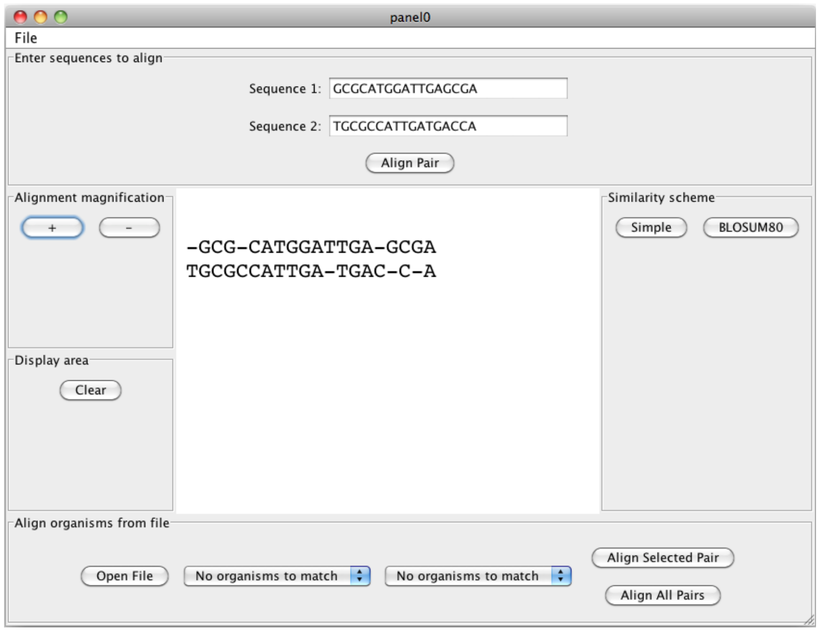


Amino Acids

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.



Sequence 1: GCGCATGGATTGAGCGA

Sequence 2: TGCGCCATTGATGACCA

```
-GCG-CATGGATTGA-GCGA
TGCGCCATTGA-TGAC-C-A
```

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 than 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) 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.

(c) 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)