

A8: Huffman

due Tuesday, August 16 11:59pm

Note: This assignment is worth a total of 30 points. It is divided into two parts, each worth approximately half of the points. **Please note that this assignment cannot be resubmitted and solutions to this homework will not be accepted after 11:59 pm on Friday, August 19th.**

This assignment will assess your mastery of the following objectives:

- Implement a well-designed Java class to meet a given specification.
- Implement, manipulate, and traverse a binary tree.
- Implement the Comparable interface
- Follow prescribed conventions for code quality, documentation, and readability.

Background: ASCII and Binary Representation

(You do not need to fully understand this section to complete the assessment.)

We discovered on an earlier assignment that every char has an equivalent int value between 0 and 255. (Recall, (int) 'a' = 97, (int) 'z' = 122, etc.) In this assignment, we go a step further: computers actually see integers as binary (1's and 0's). Binary is a lot like the normal numbers we use, except it's based around 2 instead of 10. For example, "1204" is written that way, because

$$1204 = \boxed{1} \cdot 10^3 + \boxed{2} \cdot 10^2 + \boxed{0} \cdot 10^1 + \boxed{4} \cdot 10^0$$

Similarly, because

$$97 = \boxed{0} \cdot 2^7 + \boxed{1} \cdot 2^6 + \boxed{1} \cdot 2^5 + \boxed{0} \cdot 2^4 + \boxed{0} \cdot 2^3 + \boxed{0} \cdot 2^2 + \boxed{0} \cdot 2^1 + \boxed{1} \cdot 2^0$$

97 is "01100001" in binary! In fact, all char's can be written using exactly 8 binary digits (bits). This encoding, which is used pretty universally by computers, is called ASCII.

Background: Huffman Encoding

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing data to make a file occupy a smaller amount of space. Unbelievably, this algorithm is still used today in a variety of very important areas. For example, mp3s and jpgs both use Huffman Coding. The general idea behind Huffman Coding is the following:

**What if we used fewer than 8 bits for characters that are more common?**

Consider the following (simple) example. Imagine we have the following data:

```
bbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ccccaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ddaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

How many bits to store with ASCII Coding?

First, we find the ASCII code for each letter:

a → 01100001,      b → 01100010,  
c → 01100011,      d → 01100100

Since each line has 80 letters, and each letter code is 8 bits, the number of bits required is:

$$\boxed{80 \cdot 8 \cdot 4 = 2560}$$

How many bits to store with Huffman Coding?

Since a is most frequent, we use a short code for it, then we use the next longest code for b, etc:

a → 1,              b → 00,  
c → 011,            d → 010

This data has 229 a's, 4 b's, 3 c's, and 2 d's. Since we need one bit for a, two for b, and three for c and d, the total count of bits is:

$$\boxed{229 \cdot 1 + 4 \cdot 2 + 3 \cdot 3 + 2 \cdot 3 = 255}$$

Woah! By changing the coding, we compressed the data by a factor of 10!!

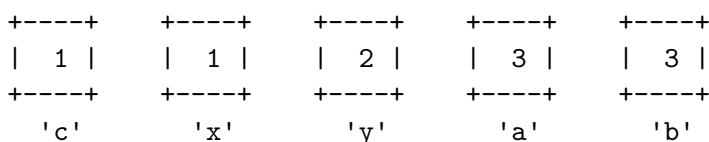
## Program Behavior: Building a Huffman Code

This program will give you practice with binary trees and priority queues. In this program you will explore how text files can be compressed by using a coding scheme based on the frequency of characters. We will use a coding scheme called Huffman coding. The basic idea is to abandon the way that text files are usually stored. Instead of using the usual seven or eight bits per character, Huffman's method uses only a few bits for characters that are used often, more bits for those that are rarely used.

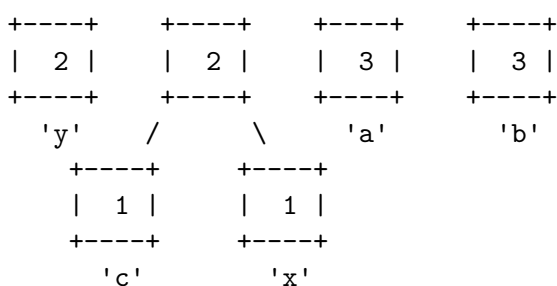
You will solve this problem using a structure known as a priority queue. In a priority queue each value inserted into the queue has a priority that determines when it will be removed. There are many ways to specify the priorities. For this program you will construct objects that implement the Comparable interface, with objects that are less being given a higher priority (to be removed first).

The first step is to compute the frequency of each character in the file you wish to encode. This allows you to determine which characters should have the fewest bits, etc. The next step is to build a coding tree from the bottom up according to the frequencies. An example will help make this clear. To make the example easier, suppose we only want to encode the five letters (a, b, c, x, y) and they have frequencies 3, 3, 1, 1, and 2, respectively.

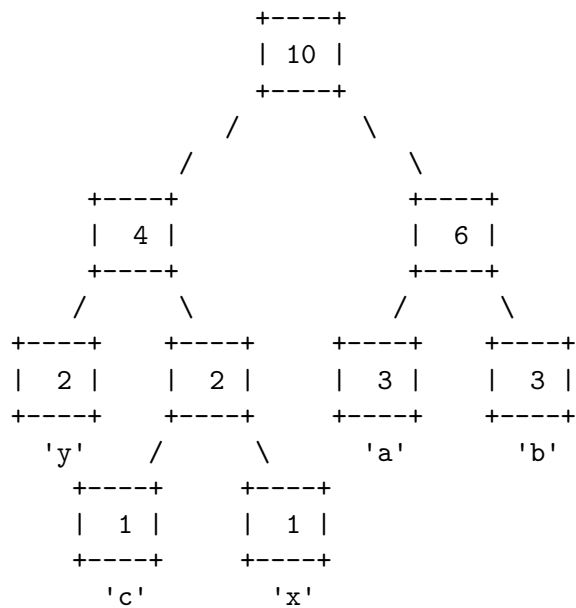
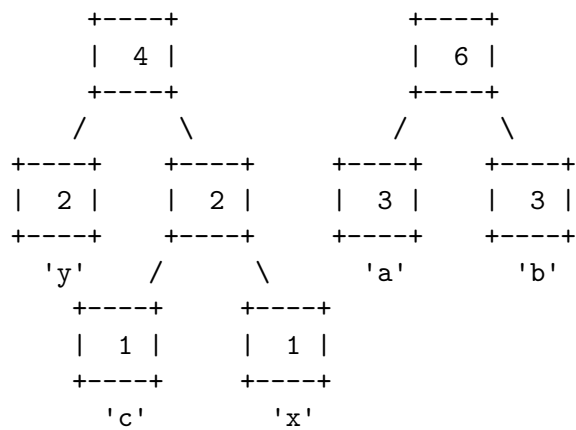
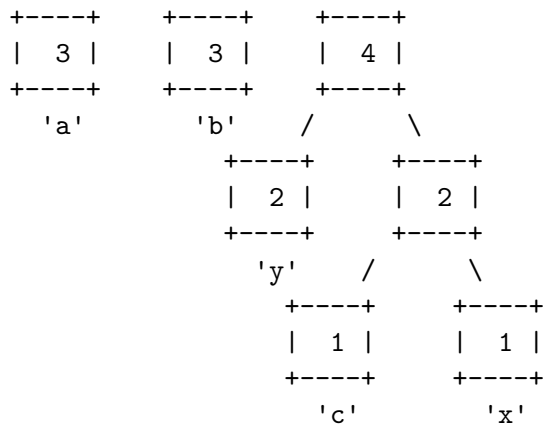
We first create a leaf node for each character/frequency pair and put them into a priority queue, so that the characters with lower frequencies appear first:



Now we pick the two nodes with the smallest frequencies (the two at the front of the priority queue) and create a new node with those two nodes as children (the first value from the queue becomes the left, the second value from the queue becomes the right). We assign this new branch node a frequency that is the sum of the frequencies of the two children. This new node is then put back into the priority queue:



Continuing in this way, we build up larger and larger subtrees. The rest of the steps are shown on the next page:



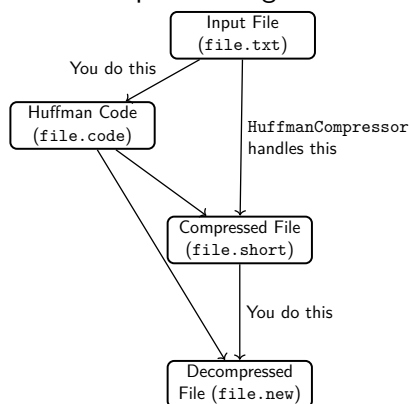
Note that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. It turns out that this structural description is exactly what is needed to create an efficient encoding. The Huffman code is derived from this coding tree simply by assigning a zero to each left branch and a one to each right branch. The code can be read directly from the tree.

The code for a is 10, the code for b is 11, the code for c is 010, the code for x is 011 and the code for y is 00.

An interesting feature of the Huffman code is that delimiters between characters are not stored, even though different characters may be coded with different numbers of bits. The key is that a code created by this method exhibits what is known as the *prefix property*, which means that no code for a character is the prefix of the code of any other character. Thus, to decode a message we need only traverse our tree. When we reach a leaf, we know that we have decoded one character, and can now start decoding the next character.

## Program Behavior: Huffman Compression and Decompression

Now that we know how to construct a Huffman code, we are ready to understand the Huffman compression and decompression algorithms. Here is an overview of how they work:



### Compression Algorithm

- (1) HuffmanCompressor generates the frequencies in a given file.
- (2) HuffmanTree creates the Huffman code from the frequencies.
- (3) HuffmanTree writes out the Huffman codes to a .code file.
- (4) HuffmanCompressor writes a .short file.

### Decompression Algorithm

- (1) HuffmanTree reads in a .code file in standard format.
- (2) HuffmanTree writes out a .new file.

HuffmanCompressor is a client program that we will provide to you. In addition to handling user interactions, it implements some of the steps of the compression and decompression algorithms that we are not asking you to deal with. Specifically, HuffmanCompressor handles:

- computing character frequencies for a given input file (these frequencies are passed to the HuffmanTree(int[] count) constructor described below)
- compressing a text file using a given Huffman tree
- producing a BitInputStream from a given compressed input file (this stream is passed to the decode method described below)

You do not need to implement any of the above behavior. You **only** need to implement the behavior and methods described in Part 1 and Part 2 below)

## Part 1: Making a Code

For our purposes, we will encode what are known as bytes (8 bits). This will allow us to encode standard text files and binary files as well. From the point of view of your Huffman code, you can think about the individual bytes as simple integers in the range of 0 to 255, each representing the integer value of a particular character. In part 1, you are working with option 1 in HuffmanCompressor. This option prompts the user for a file to examine and it computes the frequency of each character in the file. These counts are passed as an array to your HuffmanTree constructor.

The array passed to your constructor will have exactly 256 values in it, but your program should not depend on this. Instead, you can use the length field of the array to know how many there are. In your constructor, you should use a priority queue to build up the tree as described above. First you will add a leaf node for each character that has a frequency greater than 0 (we don't include the other characters in our tree). These should be added in increasing character order (character 0, character 1, and so on).

Then you build the tree. Initially you have a bunch of leaf nodes. Your goal is to get a single tree. While you haven't gotten down to a single tree, you remove two values from the priority queue and combine them to make a new branch node which you put back into the queue, as described above. You continue combining subtrees until you get down to one tree. This becomes your Huffman tree.

You are to define a class called `HuffmanTree` with the following public methods (more methods will be added in part 2 of this assignment):

```
public HuffmanTree(int[] count)
```

This is the method that will construct your initial Huffman tree using the given array of frequencies where `count[i]` is the number of occurrences of the character with integer value `i`.

```
public void write(PrintStream output)
```

This should write your tree to the given output stream in standard format.

In defining your class, you will also define a node class called `HuffmanNode`. The contents of the `HuffmanNode` class are up to you. Though we have studied trees of ints, you *should* create nodes specific to solving this problem. Your `HuffmanNode` must be a `private static` inner class and should have at least one constructor used by your tree. The fields in your `HuffmanNode` *must* be public. `HuffmanNode` should not contain *any actual huffman coding logic*. It should only represent a single node of the tree.

As with 20Questions, we will use a standard format for Huffman trees. The output should contain a sequence of line pairs, one for each leaf of the tree. The first line of each pair should have the integer value of the character stored in that leaf. The second line should have the code (0's and 1's) for the character with this integer value. The codes should appear in traversal order. In other words, they should appear in the order that any standard traversal of the tree would visit them.

For the example above, the output would be as follows (the letter a has integer value 97):

```
121
00
99
010
120
011
97
10
98
11
```

It turns out that Huffman coding works best if one character is designated as end of file, meaning that every file is guaranteed to end with such a character and it will be used for no other purpose. Some operating systems have such a character, but if we want to write a general-purpose program, we have to do something that is not specific to any one operating system. So in addition to encoding the actual characters that appear in the file, we will create a code for a fictitious end-of-file character that will be used only by the Huffman encoding and decoding programs. That means that in addition to all of the legal characters, you are also going to introduce a special character that will be used to signal end-of-file. We will refer to this as the pseudo-eof character. Its value should be one higher than the value of the highest character in the frequency array passed to the constructor. For example, if the character array has entries up to character value 100, then the pseudo-eof should have value 101 and this should be true even if the count for character 100 is 0. It will always have a frequency of 1 because it appears exactly

once at the end of each file to be encoded. You will have to manually add this character to your priority queue because it will not be included as part of the frequency array.

The output listed above does not include the pseudo-eof character. When you include the pseudo-eof character with a frequency of 1, the output becomes:

```
121
00
256
010
99
0110
120
0111
97
10
98
11
```

The `java.util` package includes a `PriorityQueue<E>` class that implements the `Queue<E>` interface. You must use these to build your Huffman tree. The only difference between a priority queue and a standard queue is that it uses the natural ordering of the objects to decide which object to dequeue first, with objects considered less returned first. You are going to be putting subtrees into your priority queue, which means you'll be adding values of type `HuffmanNode`. This means that your `HuffmanNode` class will have to implement the `Comparable<E>` interface. It should use the frequency of the subtree to determine its ordering relative to other subtrees, with lower frequencies considered less than higher frequencies. If two frequencies are equal, the nodes should be considered equal.

It would have been best for Sun to define `PriorityQueue` as an interface because there are many ways to implement a priority queue, but that's not how they did it. In this case we have a tradeoff. We would like to make it clear that we need a priority queue rather than a simple queue. On the other hand, we prefer to use interfaces when possible to keep our code flexible. There isn't necessarily a right choice in this case, so we have to pick one. You should go with interfaces and flexibility. So you are once again required to use an interface (in this case, `Queue<E>`) to define the type of variables, parameters and return values.

The Huffman solution is not unique. You can obtain any one of several different equivalent trees depending upon how certain decisions are made. But if you implement it as we have specified, then you should get exactly the same tree for any particular implementation of `PriorityQueue`. Make sure that you use the built-in `PriorityQueue` class and that when you are combining pairs of values taken from the priority queue, you make the first value removed from the queue the left subtree and you make the second value removed the right subtree.

We often add debugging code that we use while developing a program and remove from the final version. Below are some debugging suggestions:

- Add a `println` to the constructor of your node class that will report the data for every leaf node that it constructs. The output comparison tool will have correct output for the short file and for Hamlet where each line has a character code followed by one space followed by frequency, as in:

```
10 2
13 2
32 22
97 7
```

- Remember that nodes are supposed to be added to the `PriorityQueue` in increasing character order. If any of the frequencies don't match, then you probably have a damaged version of the input file (e.g., you might have used copy/paste instead of directly saving the file).
- You won't be able to figure out the order of elements in the `PriorityQueue` by viewing it in jGRASP or by using a `foreach` loop or iterator. But you can write some testing code that repeatedly calls `remove` on your `PriorityQueue`, printing the frequency of each node as it is removed from the `PriorityQueue`. You should see an increasing sequence of values (from low frequency to high).

## Using jGRASP

Remember that in jGRASP you can use a structure viewer to see what your tree looks like. You do so by dragging one of your fields from the debug window outside the window and jGRASP will launch a viewer. This viewer will show you the structure of the list, but may not show you the contents of the nodes. You can fix this by selecting the wrench icon (Configure the structure to view mapping). Under Value Expressions say:

```
_node_.<field>
```

Where `<field>` is the name of the field you want to view. You can also say:

```
_node_.<field1>#_node_.<field2>
```

Where `<field1>` and `<field2>` are the names of two fields you want to view. Once you have indicated the fields you want to view, click on apply and you should see the names in the nodes. You can also adjust settings like the Width (to see more of the name) or Scale (to stretch/shrink the diagram).

You will find that you can use a viewer to see the contents of the priority queue, but this is unlikely to be helpful to you. The priority queue has an internal structure that won't make much sense to you.

## Part 2: Encoding and Decoding a File

First, you will run option 2 in `HuffmanCompressor`, which uses your `.code` file from Part 1 and the original file to produce a binary file that is the compressed version of the original. Then, the methods you implement in Part 2 will be used to reconstruct the original file from the compressed version when you run option 3.

In particular, you will have to add two new methods to your `HuffmanTree` class:

```
public HuffmanTree(Scanner input)
```

This is your constructor that will reconstruct the tree from a file. You can assume that the `Scanner` contains a tree stored in standard format.

```
public void decode(BitInputStream input, PrintStream output, int eof)
```

This method should read individual bits from the input stream and should write the corresponding characters to the output. It should stop reading when it encounters a character with value equal to the `eof` parameter. This is a pseudo-eof character, so it should not be written to the `PrintStream`. Your method can assume that the input stream contains a legal encoding of characters for this tree's Huffman code.

The first of these methods is an alternative constructor. In part 1 you wrote a constructor that took an array of frequencies and constructed an appropriate tree given those frequencies. In this part you are given a `Scanner` that contains the file produced by your write method from part 1. In other words, the input for this part is the output you produced in part 1. You are using your own output to recreate the

tree. For this second part, the frequencies are irrelevant because the tree has already been constructed once, but you are using the same node class as before. You can set all of the frequencies to some standard value like 0 or -1 for this part.

Remember that the standard format was a series of pairs of lines where the first line has an integer representing the character's integer value and the second line has the code to use for that character. You might be tempted to call `nextInt()` to read the integer and `nextLine()` to read the code, but remember that mixing token-based reading and line-based reading is not simple. Here is an alternative that uses a method called `parseInt` in the `Integer` class that allows you to use two successive calls on `nextLine()`:

```
int n = Integer.parseInt(input.nextLine());
String code = input.nextLine();
```

For the decoding part, you have to read a `BitInputStream`. This is a special class that works in conjunction with another class called `BitOutputStream`. They each have a very simple interface. They allow you to write and read compact sequences of bits.

The only method you'll use for `BitInputStream` is the following which returns the next bit from the file:

```
public int readBit()
```

Your method is doing the reverse of the encoding process. It is reading sequences of bits that represent encoded characters and it is figuring out what the original characters must have been. Your method should start at the top of your tree and should read bits from the input stream, going left or right depending upon whether you get a 0 or 1 from the stream. When you hit a leaf node, you know you've found the end of an encoded sequence. At that point, you should write the integer code for that character to the output file. In doing so, call this method from the `PrintStream` class:

```
public void write(int b)
```

You don't need to cast to `char`. You just write the integer for this particular character (the value between 0 and 255 that is stored in this leaf). Once you've written this character's integer, you go back to the top of your tree and start over, reading more bits and descending the tree until you hit a leaf again. At that point you write again, go back to the top of the tree, read more bits and descend until you hit a leaf, then write the leaf, go back to the top of the tree, and so on.

Recognizing the end of the input is tricky. Remember that we introduced a pseudo-eof character with a special value (256). The `HuffmanCompressor` will write exactly one occurrence of this character at the end of the file. At some point your decoding method will come across this eof character. At that point it should stop decoding. It should not write this integer to the `PrintStream` because it isn't actually part of the original file. The eof value will be 256 for this particular program, but your code isn't supposed to depend on this specific value, which is why it is passed to your decode method as the third parameter.

If you fail to recognize the pseudo-eof character, you might end up accidentally reading past the end of the bit stream. When that happens, the `readBit` method returns a value of -1. So if you see a value of -1 appearing, it's because you've read too far in the bit stream.

You will have to be careful if you use recursion in your decode method. Java has a limit on the stack depth you can use. For a file like `hamlet.txt`, there are hundreds of thousands of characters to decode. That means it would not be appropriate to write code that requires a stack that is hundreds of thousands of levels deep. You might be forced to write some or all of this using loops to make sure that you don't exceed the stack depth.



You are being given two data files for this assignment called `short.txt` and `hamlet.txt`. The file `short.txt` is a short input file suitable for preliminary testing. The file `hamlet.txt` contains the full text of Shakespeare's play *Hamlet*.

The table below describes the naming conventions we use for the files involved in this assignment.

Extension	Example	Description
<code>.txt</code>	<code>hamlet.txt</code>	Original text file
<code>.code</code>	<code>hamlet.code</code>	List of codes to use
<code>.short</code>	<code>hamlet.short</code>	Compressed file (binary—not human readable)
<code>.new</code>	<code>hamlet.new</code>	Decompressed file (should match the original)
<code>.debug</code>	<code>hamlet.debug</code>	A version of the <code>.short</code> file written in 0s and 1s for debugging

You should name your file `HuffmanTree.java` and you should turn it in electronically from the homework link on the class web page. A collection of files needed for the assignment is included on the web page as `ass8.zip`. The zip file will also include the sample data files (`short.txt` and `hamlet.txt`) along with their code files (`short.code` and `hamlet.code`). The code files will also be available with the output comparison tool on the class web page.

It can be challenging to debug the decoding part of the assignment because the `.short` files are binary files that are not readable in a normal text editor. You can run `HuffmanCompressor` in debug mode to write a `.debug` file, which is a version of the `.short` file written in plaintext 0s and 1s (instead of binary), which will help with debugging.

## Bonus Point: Send a Secret Message

Along with your program you should turn in files named `secretmessage.short` and `secretmessage.code` that represent a “secret” compressed message from you to your TA and its code file. The message can be anything you want, as long as it is not offensive. Your TA will decompress your message with your tree and read it while grading. You will receive one point as a bonus if you complete this task.

You will need to *upload* your `secretmessage.short` and `secretmessage.code` files to Ed by either dragging and dropping them into the window or using the “Upload” button. Copying/pasting the file into the Ed editor will **NOT** correctly copy the compressed text.

## Code Quality Guidelines

In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the General Style Deductions, Style Guide, and Commenting Guide. For this assignment, pay particular attention to the following elements:

### `x = change(x)`

An important concept introduced in lecture was called `x = change(x)`. This idea is related to proper design of recursive methods that manipulate the structure of a binary tree. You should follow this pattern where necessary when modifying your trees.

### Avoid Redundancy

Create “helper” method(s) to capture repeated code. As long as all extra methods you create are private (so outside code cannot call them), you can have additional methods in your class beyond those specified here. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.

## Generic Structures

You should always use generic structures. If you make a mistake in specifying type parameters, the Java compiler may warn you that you have “unchecked or unsafe operations” in your program. If you use jGRASP, you may want to change your settings to see which line the warning refers to. Go to Settings/Compiler Settings/Workspace/Flags/Args and then uncheck the box next to “Compile” and type in: `-Xlint:unchecked`

## Data Fields

Properly encapsulate your objects by making data fields in your HuffmanTree class `private`. (Fields in your HuffmanNode class should be `public` following the pattern from class.) Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

## Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implementation details.

## Running and Submitting

While developing your code, you should work locally on your computer in jGRASP. To test your code, you should run the provided test files and copy the printed logs into the Output Comparison Tool to check against the expected output.

If you believe your behavior is correct, you can submit your work by uploading your files and clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You should treat Ed as a final check for your correctness, not as a debugging tool. If you aren't passing the tests, you should go back to jGRASP to debug your work.

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. If you submit a version that you later decide you do not want to have graded, **you must warn your TA not to grade that version and to wait for a later submission from you.**

## Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting the IPL or Taylor's office hours
- Posting a question on the Ed discussion board

## Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.