CSE4102 Programming Language Assignments (PLA)
Due Dates Listed for Each Project

Background and Assumptions for PLA1 and PLA2

A permuted index tracks, for a given word, cross-references to the (a file, line) pairs that represent the occurrence of the word. There may be multiple occurrences of the word in a file, e.g., the word "hello" can be in (FileA, 12), (FileA, 17), etc. However, if a word occurs multiple times within the same line, there is only one entry for that word in the permuted index. The permuted index needs two data structures: one structure tracks the (file name, line number) for each word, and the second structure tracks the list of identified words and links to references. Thus, as you identify new words and insert them into the two data structures, you must perform appropriate operations that insert the words as they are found along with the references to one or more lines they are found in. For PIF you must more clearly identify "words" as sequences of characters and digits; you will need to parse out punctuation and control characters from your counting, to identify meaningful words to form the permuted index.

For PLAs 1 and 2, assume:

- Lines are terminated by a newline character. The last line of any file can be terminated by a newline/EOF combination, or simply an EOF. If there is only one line in a file (a good test case), that means that the file could be the single line with an EOF and no newline.
- Words identified for the permuted index must take into consideration the following:
  - Each word must be at least one character and start with a letter. If a word has 2 or more characters, then the second and successive characters can be letters, digits, the underscore, or the hyphen.
  - This means that you must, for the purposes of computing the permutedvindex, recognize and discard other characters.
  - Also, note that you must eliminate white space (multiple spaces or tabs) between words.
- For the purposes of the permuted index, please ignore case. That means that the following are all equivalent: red, RED, REd, ReD, etc.
- For the purposes of identifying words or computing the permuted index, recognize and discard other characters such as @, #, $, &, (,), etc.

PLA1 Pascal – Implement PIF extending WCF & WFF Solution - Due February 13, 11:59pm

This project is worth 30 points. You will be provided with an implementation in Pascal of the the Word Count Functionality (WCF) and Word Frequency Functionality (WFF) as described in:

http://sdcse.engr.uconn.edu/Cse4102/CommonProbBackground.pdf

For this first project you will implement the Permuted Index Functionality (PIF) that tracks the line(s) within a single input file where each word occurs, as described in the assumptions above and with a sample code in C++ in the project background. The Pascal implementation is in http://sdcse.engr.uconn.edu/Cse4102/pla1.zip

The suggested approach is for each word that is found to build an dedicated output string (include in a data structure) for the word, its occurrences in lines, and final word count. Programmatically, for the word "Goodbye" the string would be built in the following steps:

```
Goodbye in lines:     -- create when the word is first found
```

```
Goodbye in lines: 1      -- append " 1" when the word is first found
Goodbye in lines: 1-2      -- append "-2" when the word is found in line 2
Goodbye in lines: 1-2-5      -- append "-5" when the word is found in line 5
Goodbye in lines: 1-2-5-6    -- append "6" when the word is found in line 7
Goodbye in lines: 1-2-5-6-6    -- append "6" when the word is found again in line 7
                               -- Etc…
```

Dedicated output string is built as the words are recognized on each line. After EOF reached for all words, you print out the new dedicated output string for the word and append **"wc="** and then print out the word count "5" as an integer. You can also convert the word count to string and append. Final output if the word goodbye occurs only five times would be as below.

```
Goodbye in lines: 1-2-5-6-6 wc=5
```

Tests: http://sdcse.engr.uconn.edu/Cse4102/test1.txt, http://sdcse.engr.uconn.edu/Cse4102/test2.txt http://sdcse.engr.uconn.edu/Cse4102/test3.txt

- Utilize Dev-Pascal http://www.bloodshed.net/devpascal.html and upload your .pas file to HuskyCT. If you use a GNU compiler, please make sure it compiles/runs in the IDE. Seems slides 3 to 20 in http://sdcse.engr.uconn.edu/Cse4102/cse4102code.pptx

PLA2 Ada - Implement PIF by extending WCF & WFF Solution - Due February 27, 11:59pm

This project is worth 30 points. You will be provided with an implementation in Ada of the the Word Count Functionality (WCF) and Word Frequency Functionality (WFF) as described in:

http://sdcse.engr.uconn.edu/Cse4102/CommonProbBackground.pdf

For this first project you will implement he Permuted Index Functionality (PIF) that tracks the line(s) within a single input file where each word occurs, as described in the assumptions above and and in the description for PLA1 and with a sample code in C++ in the project background. The Ada implementation is in http://sdcse.engr.uconn.edu/Cse4102/pla1.zip

Tests: http://sdcse.engr.uconn.edu/Cse4102/test1.txt, http://sdcse.engr.uconn.edu/Cse4102/test2.txt http://sdcse.engr.uconn.edu/Cse4102/test3.txt

Utilize AdaGIDE https://sourceforge.net/projects/adagide/ and upload your .adb files to HuskyCT. If you use a GNAT compiler, please make sure it compiles/runs in the IDE. See slides 63 to 86 in http://sdcse.engr.uconn.edu/Cse4102/cse4102code.pptx

PLA Extra Modula-2 or Go – PIF - Due May 1, 11:59pm

You will get no support from either the instructor or the TA on this so that you can really experience what it's like to learn a new language by yourself. This project is worth 30 points. This extra project can replace your lowest PLA1, PLA2 or PLA4 score. For this project you will implement the Permuted Index Functionality (PIF) that tracks the line(s) within a single input file where each word occurs.

CSE4102 Programming Language Assignments (PLA)
Due Dates Listed for Each Project

Tests: http://sdcse.engr.uconn.edu/Cse4102/test1.txt, http://sdcse.engr.uconn.edu/Cse4102/test2.txt http://sdcse.engr.uconn.edu/Cse4102/test3.txt

Utilize Modula-2 or Go

- XDS Modula-2 https://www.excelsior-usa.com/xds.htmland and upload your .mod files to HuskyCT. If you use a GNU compiler, please make sure it compiles/runs in the IDE. There are samples of code related to finding word on slides 21 to 65 in http://sdcse.engr.uconn.edu/Cse4102/cse4102code.pptx and there is sample Modula-2 code in http://sdcse.engr.uconn.edu/Cse4102/PascalModula2Samples.zip
- Go https://www.tutorialspoint.com/codingground.htm for some instructions on using go eliminate examples see slides 102 to 123 in http://sdcse.engr.uconn.edu/Cse4102/cse4102code.pptx
- Three Options for setting up Golang
  - **Windows - See**
  - http://www.wadewegner.com/2014/12/easy-go-programming-setup-for-windows/
  - **Mac OS X – See**
  - https://www.goinggo.net/2013/06/installing-go-gocode-gdb-and-liteide.html
  - **Linux - see**
  - https://golang.org/doc/install
  - http://www.tecmint.com/install-go-in-linux/
  - https://www.digitalocean.com/community/tutorials/how-to-install-go-1-6-on-ubuntu-14-04

CSE4102 Programming Language Assignments (PLA)
Due Dates Listed for Each Project

PLA3 Java Program– Implement Quilting Program - Due March 26, 11:59pm

This project is worth 60 points. From Wikipedia, https://en.m.wikipedia.org/wiki/Quilt, "A **quilt** is a multi-layered textile, traditionally composed of three layers of fiber: a woven cloth top, a layer of batting or wadding, and a woven back, combined using the technique of quilting, the process of sewing the three layers together." A quilt is made by combining different panels or blocks. A sample quilt is shown below. You can see all the different squares that are adjacent to one another.

For this project you were going to develop a quilt program in Java that constructs quilts out of multiple adjacent blocks to computerize the representation of a quilt. This will involve the ability to create new quilts that are comprised of different blocks that are in perfect rows and columns. A two by three quilt would have 2 rows and three columns while a five by two quilt would have five rows and two columns.

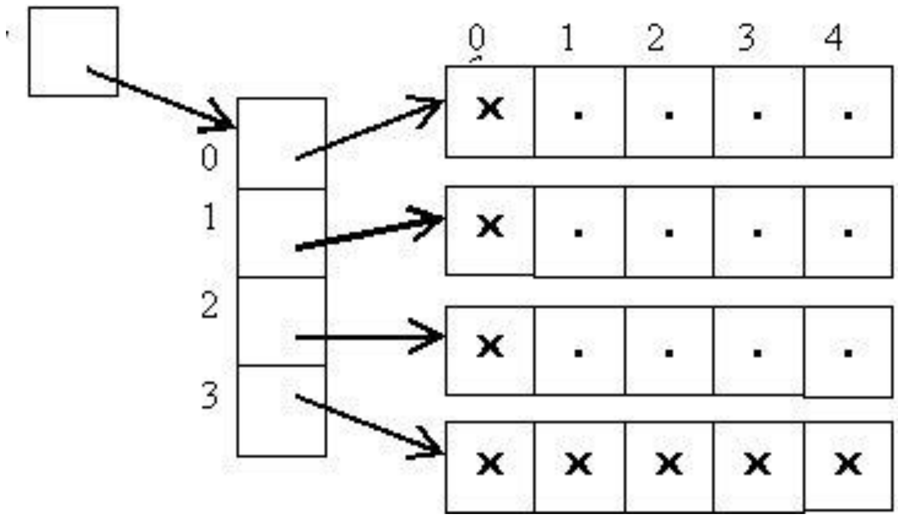
To make the problem a little more interesting, each of the blocks themselves are composed of smaller rows and columns that are patterns. Consider the Java definitions of four blocks below:

```
char[][] myBlock1 = {{'x','=','=','=','='},
{'x','+','+','+','='},
{'x','+','+','+','='},
{'x','x','x','x','x'} };

char[][] myBlock2 = {{'<','@','@','@','@'},
{'<','<','@','@','@'},
{'<','<','<','@','@'},
{'<','<','<','@','@'} };

 char[][] myBlock3 = {{'*','@','@','@','@'},
{'*','*','@','@','@'},
{'*','*','*','@','@'}
{'*','*','*','@','@'} };

char[][] myBlock4 = {{'#', '~', '~', '~', '~'},
{'#', '&','&','&','~'},
```

```
{'#','&','&','&','~' },
```

```
{'#','#','#','#','#' }
```

Each of these blocks has 4 rows and 5 columns. These four blocks are defined as two-dimensional arrays in Java that have the structure as below, so myBlock1 points to a structure with four rows we each row indexes an array of five characters which are the five columns.



When a quilt is created and stored, the resulting myQuilt is create an on the left hand side all the right hand side shows the way that it can be displayed as part of the output of the program.

Actual Quilt Pattern:

```
<@@@@<@@@@<@@@@<@@@@<@@@@
<<@@@<<@@@<<@@@<<@@@<<@@@
<<<@@<<<@@<<<@@<<<@@<<<@@
<<<@@<<<@@<<<@@<<<@@<<<@@
<@@@@<@@@@<@@@@<@@@@<@@@@
<<@@@<<@@@<<@@@<<@@@<<@@@
<<<@@<<<@@<<<@@<<<@@<<<@@
<<<@@<<<@@<<<@@<<<@@<<<@@
<@@@@<@@@@<@@@@<@@@@<@@@@
<<@@@<<@@@<<@@@<<@@@<<@@@
<<<@@<<<@@<<<@@<<<@@<<<@@
<<<@@<<<@@<<<@@<<<@@<<<@@
<@@@@<@@@@<@@@@<@@@@<@@@@
<<@@@<<@@@<<@@@<<@@@<<@@@
<<<@@<<<@@<<<@@<<<@@<<<@@
```

```
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
------------------------------
```

In support of this project, I have put together an initial program for you quilt2.java (http://sdcse.engr.uconn.edu/Cse4102/quilt.java) which has these aforementioned four blocks defined, and a main variable myQuilt that's an array of characters and functions as below. The functions that you create a new quilt, by placing blocks into the quilt, and has the ability to flip blocks which means to reverse it horizontally. This will all be reviewed in class.

- To calculate the number of **rows** use: myQuilt.length or myBlock1.length
- To calculate the number of **columns** use: myQuilt[0].length or myBlock1[0].length

```
Scanner reader = new Scanner(System.in);
System.out.print("Enter quilt rows: ");
int rows = reader.nextInt();

System.out.print("Enter quilt columns: ");
int cols = reader.nextInt();
```

```
char[][] myQuilt = new char[rows * myBlock1.length][cols * myBlock1[0].length];

displayPattern(char[][] myQuilt, int rows, int cols)

public static void createQuilt(char[][] quilt, int rows, int cols, char[][] block1,
char [][] block2)

public static void flipMyQuilt(char[][] quilt, int rows, int cols,  char[][] block)

public static void placeBlock(char[][] quilt, char[][] block, int startRow,
      int startCol)

public static char[][] createFlipped(char[][] block)
```

```
Enter quilt rows: 4
Enter quilt columns: 5
Block rows are: 4
Block cols are: 5
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
-------------------------------
|<@@@@|<@@@@|<@@@@|<@@@@|<@@@@|
|<<@@@|<<@@@|<<@@@|<<@@@|<<@@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
|<<<@@|<<<@@|<<<@@|<<<@@|<<<@@|
-------------------------------
Quiit rows are: 16
Quilt cols are: 25


-------------------------------
|xxxxx|xxxxx|xxxxx|xxxxx|xxxxx|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x====|x====|x====|x====|x====|
-------------------------------
```

```
|xxxxx|xxxxx|xxxxx|xxxxx|xxxxx|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x====|x====|x====|x====|x====|
-------------------------------
|xxxxx|xxxxx|xxxxx|xxxxx|xxxxx|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x====|x====|x====|x====|x====|
-------------------------------
|xxxxx|xxxxx|xxxxx|xxxxx|xxxxx|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x+++=|x+++=|x+++=|x+++=|x+++=|
|x====|x====|x====|x====|x====|
-------------------------------
```

In addition to this main quilting class, you also need to support the following procedures and functions:

1. **`deleteQuiltRow(q, r`**) - delete all blocks in the row r in quilt q and return an updated q. This means it's as if you cut up the quilt with scissors to remove a row and then re-sewed it.
2. **`deleteQuiltCol(q, c)`** - delete all blocks in the column c in quilt q and return an updated q. This means it's as if you cut up the quilt with scissors to remove a column and then re-sewed it.
3. **`sewTwoQuilts(q1, q2)`** - Assume q1 has r rows and c1 columns and q2 has r rows and c2 columns. Sews quilt q2 to the right of quilt q1 and returns a quilt. The two quilts must have an equal number of rows. This function returns a new quilt leaving the other quilts unchanged. The new quilt has r rows and c1 + c2 columns.
4. **`pileTwoQuilts(q1, q2)`** - Assume q1 has r1 rows and c columns and q2 has r2 rows and c columns. Sews quilt q2 below quilt q1 and returns a quilt. The two quilts must have an equal number of columns. This function returns a new quilt leaving the other quilts unchanged. The new quilt has r1 + r2 rows and c columns.
5. **`flipQuilt(q)`** – This totally inverts the entire quilt so that every row of blocks of the quilt regardless of the block goes in the opposite position. The last row of blocks of the entire quilt becomes the first row of blocks, the second to the last row of blocks becomes the second row of blocks, etc. This operates over the entire quilt independent of the blocks. You do not change any content of the block of a quilt. You do not flip the rows within each block.
6. **`flipAllBlocksinQuilt (q)`** - This inverts every individual block of the quilt while leaving them in their current position within the quilt. In this case you are inverting each of the individual blocks, which are 20 in the figure above.
7. **`turnClockwiseQuilt(q)`** - turn a quilt 90° clockwise in return and return an updated q. The 4 rows by 5 column example above will become a 5 rows and 4 columns quilt. To illustrates see the figure below which has been turned 90° clockwise. I used the carrot symbol for an example to show it clearly but you can just use the same symbol. The third column shows with <<<

```
-------       -----------       -----------
|<@@@@|       |^^^^|^^^^|       |<<<<|<<<<|
|<<@@@|       |^^^@|^^^@|       |<<<@|<<<@|
|<<<@@|       |^^@@|^^@@|       |<<@@|<<@@|
|<<<@@|       |^@@@|^@@@|       |<@@@|<@@@|
-------       -----------      -----------
|<@@@@|
|<<@@@|
|<<<@@|
|<<<@@|
```

```
-------
```

8. **`turnCounterCWQuilt(q)`** - turn a quilt 90° counterclockwise and return an updated q. The 4 rows by 5 column example above will become a 5 rows and 4 columns quilt. Counterclockwise is not shown but it would start with all of the @ at the top of the rotated quilt.
9. **`IsBlockinQuilt(q, b)`** - return true if block y is contained in quilt q - false otherwise. This is searching for an individual block in a quilt.
10. **`newcreateQuilt()`** - modify the function **`createQuilt`** by defining a second new function to significantly improve the ability to build a new quilt by adding the option to position the four different blocks in random orders. After you know the number of rows and the number of columns, simply create the quilt by using the random number generator function to generate a number from 1 to 4 in order to figure out which of the four different block patterns you should use as you create the quilt block by block.

Each of these 10 functions is worth 6 points. Your program will be limited to the four blocks that are included above, which are all 4 rows and 5 columns. You should be able to create an arbitrary size quilt of blocks that also have multiple rows and multiple columns. You can see in the output above that 4 rows and 5 columns led to the creation of the different quilts shown. The example above has the same block repeated over and over again. This should be extended to allow different blocks to be perhaps randomly inserted into each row column combination. Once you know the size of the overall quilt in blocks you can modify the create quilt method by randomly choosing one of the four different blocks as you assemble the quilt block by block. As part of the main program, provide test cases that demonstrate your program is working for all of the different options.

Use Java Eclipse https://www.eclipse.org/downloads/packages/ or GNU Java https://www.gnu.org/software/java/java.html and upload your .java files to HuskyCT.

CSE4102 Programming Language Assignments (PLA)
Due Dates Listed for Each Project

PLA4 Database Programming in Prolog –April 9, 11:59pm.

This project is worth 30 points. We will utilize SWI Prolog http://www.swi-prolog.org/ to be downloaded ftom http://www.swi-prolog.org/download/stable See slides 86 to 101 in http://sdcse.engr.uconn.edu/Cse4102/cse4102code.pptx

In the Prolog assignment, each of the tables will be stored as relations of facts. For example, the assignment would have the set of facts as below in http://sdcse.engr.uconn.edu/Cse4102/pla4.pl

```
supp(s1,'Smith',20,'London').
supp(s2,'Jones',10,'Paris').
supp(s3,'Blake',30,'Paris').
supp(s4,'Clark',20,'London').
supp(s5,'Adams',30,'Athens').
part(p1, 'Nut',   'Red',   12, 'London').
part(p2, 'Bolt',  'Green', 17, 'Paris').
part(p3, 'Screw', 'Blue',  17, 'Oslo').
part(p4, 'Screw', 'Red',   14, 'London').
part(p5, 'Cam',   'Blue',  12, 'Paris').
part(p6, 'Cog',   'Red',   19, 'London').
proj(j1, 'Sorter',  'Paris').
proj(j2, 'Display', 'Rome').
proj(j3, 'OCR',     'Athens').
proj(j4, 'Console', 'Athens').
proj(j5, 'RAID',    'London').
proj(j6, 'EDS',     'Oslo').
proj(j7, 'Tape',    'London').
sppj(s1, p1, j1, 200).
sppj(s1, p1, j4, 700).
sppj(s2, p2, j1, 400).
sppj(s2, p2, j2, 200).
sppj(s2, p2, j3, 200).
sppj(s2, p2, j4, 500).
sppj(s2, p2, j5, 600).
sppj(s2, p2, j6, 400).
sppj(s2, p2, j7, 800).
sppj(s2, p5, j2, 100).
sppj(s3, p2, j1, 200).
sppj(s3, p4, j2, 500).
sppj(s4, p6, j3, 300).
sppj(s4, p6, j7, 300).
sppj(s5, p2, j2, 200).
sppj(s5, p2, j4, 100).
```

```
sppj(s5, p5, j5, 500).
sppj(s5, p5, j7, 100).
sppj(s5, p6, j2, 200).
sppj(s5, p1, j4, 100).
sppj(s5, p2, j4, 200).
sppj(s5, p4, j4, 800).
sppj(s5, p5, j4, 400).
sppj(s5, p6, j4, 500).
```

Requirements for PLA6:

Given the database of tables (i.e., lists in Scheme and facts in Prolog), you must now write queries (i.e., functions in Scheme and rules in Prolog) for each of the following:

- When given a pair of projects, find all suppliers who supply both projects. Return the entire entry (i.e., S#, Sname, Status, City) for the supplier.
- When given a city, find all parts supplied to any project in that city. Once again, return the entire entry for the part. Find all parts supplied to any project by a supplier in the same city. In this case, results are organized by all parts for every city in the database.
- Find all projects supplied by at least one supplier not in the same city.
- Find all suppliers that supply at least one part supplied by at least one supplier who supplies at least one red part.
- Find all pairs of city values such that a supplier in the first city supplies a project in the second city.
- Find all triples of city, part#, city, such that a supplier in the first city supplies the specified part to a project in the second city, and the two city values are different.
- When given a supplier, find all projects supplied entirely by that supplier.

As mentioned above, PLA6 involves a functional/Scheme implementation, while PLA6 involves a logic/Prolog implementation.

One possible approach to the problem would be to write functions (rules) that perform the basic operations of selection (finding appropriate rows of tables), projections (printing out an entire row of a table), and join (merging two tables on a column-basis, similar to a Cartesian product). These three basic operations can then be combined to answer the queries posed above. A second approach would be to write functions (rules) specifically to answer each of the queries.

Common Background for PLA6

Consider the relational database shown below, consisting of four tables of information on suppliers (S), parts (P), projects (J), and project profiles (SPJ). Supplier numbers (S#), part numbers (P#), and project numbers (J#), are unique in tables S, P, and J – see http://wiki.c2.com/?SupplierPartsProjectsDatabase. The SPJ table contains which suppliers supply which part (in what quantities) to which project. From a database perspective, the following tables would be defined:

Table name	Purpose	Key

```
S             Suppliers                                (S#)
P             Parts                                    (P#)
J             Jobs                                     (J#)
SPJ           Parts supplied by Suppliers for Jobs     (S#, P#, J#)
S
S#  SNAME  STATUS   CITY
----------------------------
s1  Smith  20       London
s2  Jones  10       Paris
s3  Blake  30       Paris
s4  Clark  20       London
s5  Adams  30       Athens

P
P#  PNAME  COLOR  WEIGHT   CITY
---------------------------------
p1  Nut    Red    12       London
p2  Bolt   Green  17       Paris
p3  Screw  Blue   17       Oslo
p4  Screw  Red    14       London
p5  Cam    Blue   12       Paris
p6  Cog    Red    19       London

J
J#  JNAME    CITY
----------------------
j1  Sorter   Paris
j2  Display  Rome
j3  OCR      Athens
j4  Console  Athens
j5  RAID     London
j6  EDS      Oslo
j7  Tape     London

SPJ
S#  P#  J#  QTY
-----------------
s1  p1  j1  200
s1  p1  j4  700
s2  p2  j1  400
s2  p2  j2  200
s2  p2  j3  200
s2  p2  j4  500
s2  p2  j5  600
s2  p2  j6  400
s2  p2  j7  800
s2  p5  j2  100
s3  p2  j1  200
s3  p4  j2  500
s4  p6  j3  300
s4  p6  j7  300
s5  p2  j2  200
s5  p2  j4  100
s5  p5  j5  500
s5  p5  j7  100
s5  p6  j2  200
s5  p1  j4  100
s5  p2  j4  200
s5  p4  j4  800
s5  p5  j4  400
s5  p6  j4  500
```

CSE4102 Programming Language Assignments (PLA)
Due Dates Listed for Each Project