Due date: **December 1st, 10:30am**

This lab is a sequence of exercises in Prolog. Each exercise requires you to write a program in a file called `exN.pl`, where `N` is the number of the exercise.

General advice and rules:

- Efficiency is not a major worry (within reason); we're exploring what you can do, not how to get the very best solution.
- Don't worry about detecting input errors; assume the input is correct. That implies that both the value and the type of the data supplied are correct.
- **Make sure all programs run on pl on the ECF workstations.**
- Make sure you exactly follow the specifications for each question.
- The specifications below are meant to be precise, unlike real-world requirements. However, there are bound to be places where clarity has not been achieved. If you find yourself confused, please ask.
- A Prolog query can be asked to backtrack after success. For example, in an interactive session you may press the ';' key to see if there is another answer. Your rules should produce ***every* correct answer**, and your rules should (of course) ***never* produce an incorrect answer**. Some exercises may also require that every correct answer is produced ***exactly* once**. You should always check to see what happens when your rule is asked to backtrack.
- Some exercises look ahead to topics not covered at the time when the lab is handed out.
- These exercises are short, but you should still follow the usual style rules, which are just as relevant for all three programming languages used in this course as for any other languages you have used. For example, you should choose helpful names for variables and predicates. Comments are valuable in all languages; even a very short code segment may be worth commenting, if you had to think hard to write it.

Academic offenses:
The standard rule is in effect: don't share your work with anyone else. Complete this lab by yourself.

Submission:
Submit your work using the command `submitcsc326f`. Set the first argument – the "assignment#" – to 3.

Exercise 1 [5 marks]

Write a predicate called **`rmlast(X,Y)`** that succeeds if `X` and `Y` are lists, and `Y` is the same as `X` except that `X`'s last element is not present in `Y`. Neither `X` nor `Y` is required to be instantiated. When backtracking after success, you should produce every correct answer ***exactly* once**. (Under what circumstances can there be more than one correct answer?)

Your submitted file must be named `ex1.pl`.

Examples:

```
?- rmlast(X, [a,b,c]).
X = [a, b, c, _G234] ;
No

?- rmlast([a,b,c], Y).
Y = [a, b] ;
No
```

Exercise 2 [10 marks]

Write a predicate called **`secondlargest(List,Val)`** that succeeds if `List` is a list of integers and `Val` is equal to the second-largest element of `List`. If there is a tie for first place, the second-largest number is the same as the largest. If `List` has fewer than two elements, the second-largest number is defined to be 0. *Note that the definition of second-largest number is **not** the same as the definition used for Nth-smallest number in Lab 1 and Lab 2.*

You may require `List` to be instantiated. Even if more than one entry in `List` has the second-largest value, you should nevertheless produce the correct answer ***exactly* once**.

Your submitted file must be named `ex2.pl`.

Examples:

```
?- secondlargest([5], Val).
Val = 0 ;
No

?- secondlargest([1,-5,3,2,2,0], Val).
Val = 2 ;
No

?- secondlargest([3,3,2,1,0,-1,-2], Val).
Val = 3 ;
No
```

Exercise 3 [10 marks]

Here are some tables representing what we know about various products sold by the Acme Equipment Company, and about Acme's customer orders.

One step in this exercise is to represent this information as a Prolog knowledge base; during that process, you should change all upper-case letters to lower case, combine multiple words into one, and omit punctuation. For example, "Stick of dynamite" will become `stickofdynamite`, and "E. Fudd" will become `efudd`.

part-name	part-number	price
Anvil	423	10.99
Stick of dynamite	567	1.97
Shotgun	128	99.99
Broccoli	256	0.99
Carrot stick	511	0.47

part-number	quantity
423	36
567	200
128	5
256	93
511	4892

customer-name	part-number	quantity
W. Coyote	423	300
E. Fudd	567	1
B. Bunny	511	94
B. Bunny	256	9723
W. Coyote	128	12

Here are your tasks for this exercise:

a) Represent the tables above as Prolog facts. Use predicate names **`part`**, **`inventory`**, and **`order`**.

b) Write a predicate called **`bigorder(Cust)`** that succeeds if `Cust` is the name of a customer who has ordered more than 100 of any item in a single order.

c) Write a predicate called **`notenough(Cust, Part)`** that succeeds if `Cust` is the name of a customer, `Part` is the name of a part, and `Cust` has ordered more of `Part` than Acme has in stock. Again, you only need to consider one order at a time, rather than summing the quantities over multiple orders.

None of the arguments for the predicates **`bigorder`** and **`notenough`** should have to be instantiated, and backtracking should produce all the correct answers. The rules for

`bigorder` and **`notenough`** should work for any data, not just the data in the above tables (in other words, you should ***not*** hard-code answers based on the above data).

Your submitted file must be named `ex3.pl`.

Examples:

```
?- bigorder(Cust).
Cust = wcoyote ;
Cust = bbunny ;
No

?- notenough(Cust, Part).
Cust = wcoyote
Part = anvil ;

Cust = bbunny
Part = broccoli ;

Cust = wcoyote
Part = shotgun ;
No
```

Exercise 4 [10 marks]

A *heap* is a binary tree in which every element is greater than or equal to both its children, if these children exist. Note that a heap is not a binary search tree, and that it is unrelated to the kind of heap used in dynamic memory allocation. (Heaps are used in heap sort, a fast sorting algorithm.)

Write a predicate **`heap(Tree)`** that succeeds if `Tree` is a heap. `Tree` must be instantiated, and the data contained in `Tree` must be integers. If these conditions are violated, any behaviour is acceptable.

For this exercise, use the same tree representation as in the lecture slides: `node(K,L,R)` is a tree with key `K` (an integer), left subtree `L` and right subtree `R`. The atom `empty` represents an empty binary tree.

Your submitted file must be named `ex4.pl`.

Examples:

```
?- heap(node(5, empty, node(-2, empty, empty))).
Yes
?- heap(empty).
Yes
```

Exercise 5 [12 marks]

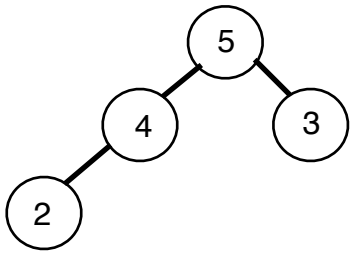
Repeat Exercise 4, but with a different tree representation: a list.

A complete binary tree is a binary tree in which every level is full except possibly the bottom level, where there are only leaves. If the bottom level is not full, the missing nodes must be to the right of the leaves that are present.

A complete binary tree can be represented as an array, or in Prolog as a list. In this list, the first element (at index 1) is the root, and the children of element N are in the nodes at indices 2N and 2N + 1.

Note that the preceding paragraph uses indices starting at 1, not 0, because it's easier to describe the tree representation that way.

Your **`heap(Tree)`** predicate can assume that `Tree` is a valid complete binary tree containing integer data. With the list representation, that just means you can assume `Tree` is a list of integers. For example, the list [5, 4, 3, 2] is the complete binary tree shown below. This tree is, in fact, also a heap.



You ***cannot*** assume there is any special integer that represents a missing node: you know you're at a leaf when **both** children would be past the end of the list.

Your submitted file must be named `ex5.pl`.

Examples:

```
?- heap([5, 4, 3, 2]).
Yes
?- heap([7, 6, -1, -3, 5]).
Yes
?- heap([]).
Yes
```