Writing good psuedocode

Source: http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html

Basic rules:

1. Use the language of the problem domain, not the implementation domain:
    a. i.e. Get the next word in the sentence **NOT** increment the word index to the next index in the word array
    b. Append the file extension to the name **NOT** name = name + extension
    c. For all characters in name **NOT** For i = 0 to length(word) word[i]
2. The logic must be decomposed to the level of a single decision
    a. **BAD** Search the book for the longest word - This is a loop and a decision
3. Typical “psuedocode” instructions that begin a good instruction are constructs such as:
    a. While *condition*
    b. if *condition* then *sequence* else *sequence* endif
    c. Repeat-Until *condition*
    d. CASE *branch 1*
    e. FOR

- Note that flow control statements should be terminated if they are long
  - IF/ENDIF , WHILE/ENDWHILE
- Easiest use of CASE is:
  - CASE expression OF
    - condition 1: *sequence*
    - condition 2: *sequence*
  - OTHERS or DEFAULT: *sequence*
  - ENDCASE
- REPEAT *sequence* UNTIL condition
- FOR
  - FOR EACH MONTH OF YEAR **GOOD**
  - FOR MONTH = 1:12 **OK**
  - FOR EACH employee in LIST **GOOD**

- Nested flow control should be indented AND closed

- Subprocedures
  - Use CALL keyword for maximum clarity, append with RETURNING to explain exactly what it does
  - CALL getBalance RETURNING aBalance

Example - game playing:

```
counter = 1
FOR X = 1 to 10
        FOR Y = 1 to 10
                IF gameBoard[X][Y] = 0
                        Do nothing
                 ELSE
                        CALL theCall(X, Y) (recursive method)
                        increment counter
                END IF
        END FOR
END FOR
```

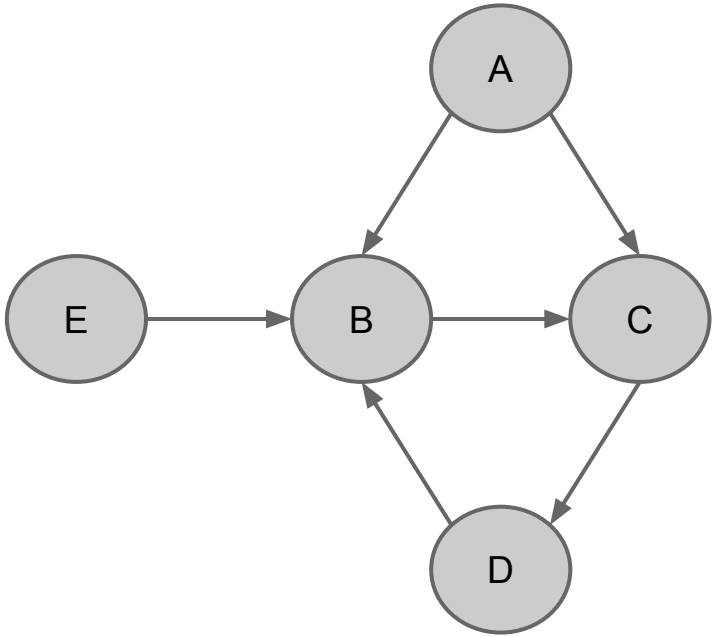
Note, instructions are given in the implementation domain (indices x and y, test of gameBoard[x][y], and variable names are not informative

```
Set moveCount to 1
FOR each row on the board
        FOR each column on the board
                IF gameBoard position (row, column) is occupied THEN
                        CALL findAdjacentTiles with row, column
                        INCREMENT moveCount
                END IF
        END FOR
END FOR
```

Now the language is in the problem domain.

1 b) - Calculate the transitive closure of A

```
a =
matrix([[0, 1, 1, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0],
        [0, 1, 0, 0, 0]])
```




```
pow(a,5) | pow(a,4) | pow(a,3) | pow(a,2) | pow(a,1)
matrix([[0, 1, 1, 1, 0],
        [0, 1, 1, 1, 0],
        [0, 1, 1, 1, 0],
        [0, 1, 1, 1, 0],
        [0, 1, 1, 1, 0]])
```

How can we determine if there is a cycle in the graph?

Each row of the table holds all of the nodes that can be reachable from that node.

If there is a non-zero value in an on diagonal row, the node is reachable from itself. Therefore there is a cycle.

DEREG AIRLINES FLIGHT CONNECTIONS TABLE					
	to city 1	to city 2	to city 3	to city 4	to city 5
from city 1	-	-	305	278	-
from city 2	101	-	234	176	-
from city 3	076	005	-	-	-
from city 4	432	901	675	-	765
from city 5	-	211	303	180	-

a. Since the number of cities is held constant and the number of empty cells of the table is around the same number of non-empty cells, and adjacency matrix is reasonable.

```
class RoutingTable
{
        private int[][] routingTable = new int[5][5];
         //! returns an array containing the number of flights out of the city in the
        0th index and the cities that it connects with in the subsequent indices.
        public int[] getRoutesOut(int city_number)
        ...
```

- This is a feasibility problem - there is no cost given to each flight, and you aren't told to minimize the number of flights.

- In this case, depth first search is sufficient
  - But there might be cycles, and we didn't specify a way of "marking" nodes
    - Make flight values negative to mark them.

```
private bool findRoute(int from, int to, int flight[ ], int flightarray_index)
{
        bool succeeded = False;
        if (from == to)
                return True;
        int routes[] = new int[5];
        for(int i = 0; i < 5; ++i)
        {
                if(routingTable[from][i] < 0)
                        return False;
                if (!routingTable[from][i]) // If no flight exists, go to next
neighbor.
                        continue;
                flight[flightarray_index] =  routingTable[from][i]
                //! mark all flights into the node we're about to visit as
negative
                for(int j = 0; j < 5' ++j)
                        routingTable[i,j] *= -1;
                if(findRoute(i, to, flight[]. flightarray_index +1)
                {
                        succeeded = True;
                        break;
                }
        }
  // clean up flight table if we are at the 0th level
  if(flightarray_index ==0)
{
        for(int i =0; i < 5; ++i)
                for(int j=0; j < 5; ++j)
                        if(routingTable[i,j] < 0)
                                routingTable[i,j] *= -1;
}
return succeeded;
}
```

- That solution uses DFS - doesn’t find the shortest route, but works easily with recursion.

- We have an implicit stack, but to do BFS we need a queue.

- What are our options, without changing the function signature and without adding more members?

c) Make it a List[][] -> This requires a little extra work because you can’t have a generic container array

Virgin records catalogue

Assumptions: Artist names and work names are unique

We need both forward and reverse lookup.

Many fewer artists than works.

```
class Record{
int type;
int year;
string author;
string name;
}
```

Virgin records catalogue

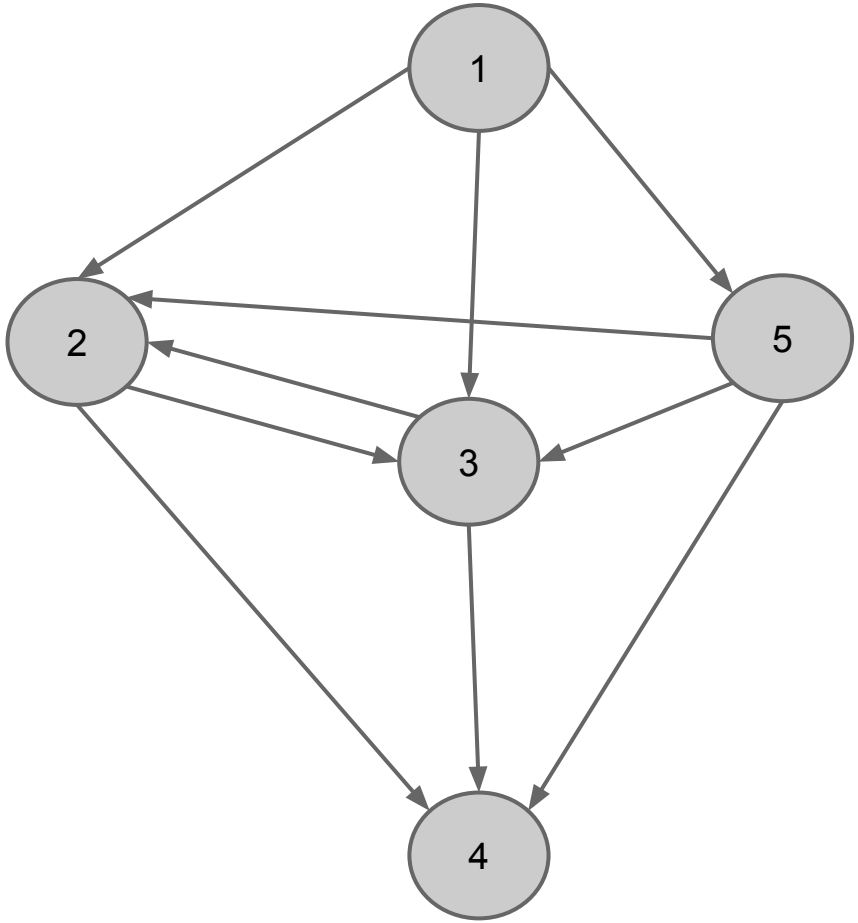
For forward lookup by artist name, we could use something like a prefix trie. The leaves can be lists of records sorted by year. The lists can be filtered by type when the works are requested. Artists will be added and removed less often than works, so the structure of the trie will not change that much.

For reverse lookup, it may be better to have a hashmap to the artist name using separate chaining, since the records are added often and are small, keeping the hashmap sparse will not be too expensive. Removing items from a seperate chained hashmap is not so bad.

4-b)1,2,3,4,5

4-c)1,2,3,5,4

4-d)See text book



City	Denver	Salt Lake	Albuquerque	Oklahoma City	Cheyenne	Omaha
Salt Lake	512					
Albuquerque	422	611				
Oklahoma City	615	1108	525			
Cheynne	102	462	522	706		
Omaha	540	955	895	454	493	
Dodge City	301	682	425	212	355	336

5a - This is a MST problem. We can use the greedy algorithms to solve is
SaltLake-Cheyenne is the cheapest connection to salt lake
Cheyenne-Denver is the next cheapest connection
Denver-Dodge is the next cheapest
Dodge-OC is the next
Dodge-Omaha
Albuqurque-Denver

Problem 7

This problem should remind you of quicksort - it is basically a pivoting problem. The algorithm goes something like this

Set H to the Head of the list of objects

Set T to the Tail of the list of objects

While H and T do not overlap

    While H does not point to a 1 key object and H does not overlap T

        Move H forward

    While T does not point to a 0 key object and H does not overlap T

        Move T backward

    Swap H and T