B561 Assignment 8. (Sample problems and solutions)
Database Programming

1. **In this problem, you can not use arrays**.

   Consider the relation schema `Graph(source int, target int)` representing the schema for storing a directed graph G as a set of edges.

   Recall that G is *connected* if for each pair of vertices (s, t) in G, there exists a path in G from s to t.

   An *articulation vertex* of G is a vertex $\mathbf{v}$ of G such that removing the edges in G with source or target $\mathbf{v}$ results in a graph that is **not** connected. More formally, $\mathbf{v}$ is an articulation vertex of G, if the graph

   $$G - (\{(s,t)|(s,t) \in G \wedge (s = \mathbf{v} \vee t = \mathbf{v}\})\}$$

   is **not** connected.

   We say that G is *bi-connected* if G does not have any articulation vertices.

   Write a PostgresQL program `biConnected()` that returns true if G is bi-connected and false otherwise.

2. **In this problem, you can not use arrays**.

   Consider the relational schema `PC(parent int, child int)` representing the schema for storing a set of parent-child pairs.

   We now inductively define a predicate `SG`(p_1,p_2) to denote that person p_1 and p_2 are in the *same generation* according to the parent-child relation:

   Rule 1: if p is a person in the `PC` relation then `SG`(p, p)
   Rule 2: if, for persons p_1, p_2, p_3, and p_4 in the `PC` relation,
   (a) `PC`(p_1, p_2), (b) `PC`(p_3, p_4), and (c) `SG`(p_1, p_3), then `SG`(p_2, p_4)

   The first rule states that a person is in the same generation as him or herself. The second rule states that two children p_2 and p_4 are in the same generation if they have parents p_1 and p_3, respectively, that are in the same generation.

   Write a PostgresQL program `sameGeneration()` that implements the same generation predicate `SG`.

3. **In this problem, you can not use arrays**.

Consider the following relational schemas. (You can assume that the domain of each of the attributes in these relations is `int`.)

```
partSubpart(pid,sid,quantity)
basicPart(pid,weight)
```

A tuple (p, s, q) is in `partSubPart` if part s occurs q times as a **direct** subpart of part p. For example, think of a car c that has 4 wheels w and 1 radio r. Then $(c, w, 4)$ and $(c, r, 1)$ would be in `partSubpart`. Furthermore, then think of a wheel w that has 5 bolts b. Then $(w, b, 5)$ would be in `partSubpart`.

A tuple (p, w) is in `basicPart` if basic part p has weight w. A basic part is defined as a part that does not have subparts. In other words, the pid of a basic part does not occur in the pid column of `partSubpart`.

(In the above example, a bolt and a radio would be basic parts, but car and wheel would not be basic parts.)

We define the *aggregated weight* of a part inductively as follows:

(a) If p is a basic part then its aggregated weight is its weight as given in the `basicPart` relation

(b) If p is not a basic part, then its aggregated weight is the sum of the aggregated weights of its subparts, each multiplied by the quantity with which these subparts occur in the `partSubpart` relation.

Comments:

(a) In the above example, bolt is a **direct sub-part** of wheel, but not of car. Furthermore, bolt would appear with its weight in `Basic_Parts`, but car nor wheel would appear in this relation.

In other words, only the PId's of parts that have no sub-parts in `Parts_SubParts` are in `Basic_Parts`.

(b) If the weight of a part is in `Basic_Parts`, the aggregated weight of that part is that weight. Otherwise, the aggregated weight of a part is the sum of the aggregated weights of all its **direct** sub-parts. So the weight function of a part is recursively defined.

Example tables: The following example is based on a desk lamp with `pid` 1. Suppose a desk lamp consists of 4 bulbs (with `pid` 2) and a frame (with `pid` 3), and a frame consists of a post (with `pid` 4) and 2 switches (with `pid` 5). Furthermore, we will assume that the weight of a bulb is 5, that of a post is 50, and that of a switch is 3.

Then the `partSubpart` and `basicPart` relation would be as follows:

partSubPart

pid	sid	quantity
1	2	4
1	3	1
3	4	1
3	5	2

Parts

pid	weight
2	5
4	50
5	3

Then the aggregated weight of a lamp is $4 \times 5 + 1 \times (1 \times 50 + 2 \times 3) = 76$.

Write a PostgreSQL function `aggregatedWeight(p integer)` that returns the aggregated weight of a part `p`.

4. **In this problem, you can use arrays, but only as a mechanism to represents subsets of `A(x, int)`.**

   Consider the relation schema `A(x int)` representing a schema for storing a set of integers A.

   Using arrays to represent sets, write a PostgreSQL program

   $$\texttt{superSetsOfSet(}X\ \texttt{int[])}$$

   that returns each subset of A that is a superset of X, i.e., each set Y such that $X \subseteq Y \subseteq A$.

   For example, if $X = \{\}$, then `superSetsofSets(X)` should return each element of the powerset of A.

5. **In this problem, you can use arrays, but only as a mechanism to represents sets of words**.

   Consider the relation schema `document(doc int, words text[])` representing a relation of pairs (d, W) where d is a unique id denoting a document and W denotes the set of words that occur in d.

   Let **W** denote the set of all words that occur in the documents and let t be a positive integer denoting a *threshold*.

   Let $X \subseteq \mathbf{W}$. We say that X is t-frequent if

   $$\texttt{count}(\{d|(d, W) \in \texttt{document}\ \text{and}\ X \subseteq W\}) \geq t$$

   In other words, X is *t-frequent* if there are at least t documents that contain all the words in X.

   Write a PostgreSQL program `frequentSets(t int)` that returns each t-frequent set.

   In a good solution for this problem, you should use the following rule: if X is not t-frequent then any set Y such that $X \subseteq Y$ is not t-frequent either. In the literature, this is called the *Apriori* rule of the frequent itemset mining problem.

6. **In this problem, you can not use arrays**.

   Consider the relation schema `Graph(source int, target int)` representing the schema for storing a directed graph G of edges.

   Now let G be a directed graph that is acyclic, a graph without cycles.[1]

   A *topological sort* of a graph is a list (array) of **all** its vertices $(v_1, v_2, \ldots, v_n)$ such that for each edge (s, t) in G, vertex s occurs before vertex t in this list.

   Write a PostgresQL program `topologicalSort()` that returns a topological sort of G.

   In this problem, you are **not** allowed to use arrays!

[1] A cycle is a path $(v_0, \ldots, v_k)$ where $v_0 = v_k$.

7. **In this problem, you can not use arrays**.

   Let `Graph(source int, target int, weight int)` be the schema for storing a connected undirected weighted graph G. The weight of an edge is a non-negative integer.

   A *spanning tree* T of G is a sub-graph of G that is acyclic and such that for each vertex v in G there is an edge in T of the form (v, w) or (w, v). I.e., each vertex of G is the end point of an edge in G. The *weight* of a sub-graph of G is the sum of the weights of the edges of that sub-graph. A *minimum spanning tree* of G is a *spanning tree* of G of minimum cost.

   Write a PostgreSQL program `minimumSpanningTree()` that determines a minimum spanning tree of a graph G. You can use Prim's Algorithm to determine a spanning tree. Consult

   https://en.wikipedia.org/wiki/Minimum_spanning_tree

   and

   https://en.wikipedia.org/wiki/Prim's_algorithm.

   In these pages you can find some graphs on which you test your program.

8. **In this problem, you can not use arrays**.

Consider the heap data structure. For a description, consult

https://en.wikipedia.org/wiki/Binary_heap.

(a) Implement this data structure in PostgreSQL. This implies that you need to implement the `insert` and `extract` heap operations.

In this problem, you are **not** allowed to use arrays to implement this data structure! Rather you must you relations.

(b) Then, using the heap data structure developed in question 8a, write a PostgreSQL program `heapSort()` that implement the `Heapsort` algorithm. For a description of this algorithm, see

https://en.wikipedia.org/wiki/Heapsort

You are **not** allowed to use arrays to implement this the `Heapsort` algorithm!

The input format is a list of integers stored in a binary relation `Data(index,value)`. For example, `Data` could contain the following data.

Data

index	value
1	3
2	1
3	2
4	0
5	7

The output of `heapSort()` should be stored in a relation `sortedData(index,value)`. On the `Data` relation above, this should be the following relation:

sortedData

index	value
1	0
2	1
3	2
4	3
5	7

9. **In this problem, you can not use arrays**.

   Consider the relation schema `Graph(source int, target int)` representing the schema for storing a directed graph G of edges.

   Let 'red', 'green', and 'blue' be 3 colors. We say that G is *3-colorable* if it is possible to assign to each vertex of G one of these 3 colors provided that, for each edge (s, t) in G, the color assigned to s is different than the color assigned to t.

   Write a PostgresQL program `threeColorable()` that returns true if G is 3-colorable, and false otherwise.

   (Hint: use a backtracking algorithm that finds a 3-color assignment to the vertices of G if such an assignment exists.)