Assignment 5 — Greedy Algorithms

Due: Friday, November 1st

Credit: This is a lab by Christopher Siu.

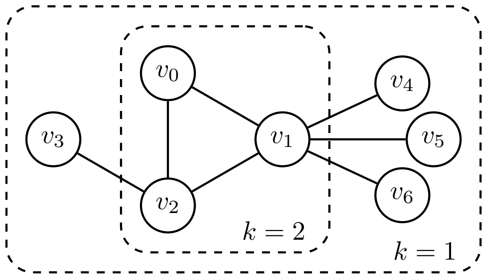
A greedy algorithm is one that assumes it can repeatedly make locally optimal choices, never having to backtrack, always arriving at a globally optimal solution. The greedy approach can be applied to a wide variety of problems, including those that involve graphs.

Deliverables:

GitHub Classroom:	`https://classroom.github.com/a/WfTVjKB2`
Required Files:	`compile.sh`, `run.sh`
Optional Files:	`*.py`, `*.java`, `*.clj`, `*.kt`, `*.js`, `*.sh`

Part 1: *k*-Cores

A *k-core* is a maximal connected subgraph within which every vertex has degree at least k. They have applications in graph-based problems as a computationally inexpensive way of approximating the most densely connected regions of a graph. For example, consider the following undirected graph:



All seven vertices in this graph can be found within a 1-core, since they all have a degree of at least 1. However, the 2-core contains only vertices v_0, v_1, and v_2. Note that although the graph contains a vertex of degree 3 (as well as one of degree 5), there is no 3-core — there is no subgraph within which *every* vertex has degree 3. Vertex v_2 has degree 3, but its neighbor v_0, for example, only has degree 2. This means that v_0 cannot be in a 3-core, and without v_0, v_2's degree falls below 3.

Perhaps more importantly, from the perspective of a greedy approach, note that the k-cores form a nesting hierarchy of vertices: any vertices in the i-core must necessarily also be in the $(i-1)$-core, but there can be vertices in the $(i-1)$-core that are not in the i-core.

In your programming language of choice per Assignment 1, implement a greedy algorithm to compute the k-cores of a graph. Think carefully about what metric your algorithm will be greedy over[1] and what data structures it will use to keep track of that information. You should be able to produce *all* of the k-cores of a graph in linear $O(|V| + |E|)$ time[2].

Each input graph will be provided as an *edge list*: each edge in the graph will be represented by a comma-separated pair of vertex identifiers, indicating an edge between the first vertex and the second.

You may assume that vertex identifiers are contiguous natural numbers — they begin at 0, and there will be no "gaps" in the identifiers used. You may also assume that the graph will be simple and will not contain any isolated vertices[3].

[1]Consider *pruning*: removing vertices and edges that cannot possibly be part of some solution.
[2]Formatting and printing the cores will naturally require more time.
[3]Since there are no isolated vertices, there are no meaningful 0-cores; they will always be identical to the 1-cores.

For example, the above graph could be represented as:

```
0, 1
0, 2
1, 2
1, 4
1, 5
1, 6
2, 3
```

Your program must accept as a command line argument the name of a file containing an edge list as described above, then print to `stdout` the k-cores according to the following format:

- Every non-empty k-core must be printed, starting with $k = 1$.
- For each value of k, all vertices in such k-core(s) must appear on a single comma-separated line.
- Each line's vertices must be sorted in ascending order. Note that vertex identifiers are integers, not strings. The lines themselves must be sorted in ascending order by k.

For example:

```
>$ ./compile.sh
>$ ./run.sh in1.txt
Vertices in 1-cores:
0, 1, 2, 3, 4, 5, 6
Vertices in 2-cores:
0, 1, 2
```

Your program will be tested using `diff`, so its printed output must match *exactly*.

Part 2: Submission

The following items must be demonstrated in lab on the day of the deadline:

- Pseudocode for an efficient greedy algorithm to find the k-cores of an undirected graph.
- A brief description of the locally optimal choices your algorithm greedily makes, together with a brief justification of why it is safe to assume that those choices will lead to a globally optimal[4] solution.

The following files are required and must be pushed to your GitHub Classroom repository by 8pm on the due date:

- `compile.sh` — A Bash script to compile your submission (even if it does nothing), as specified.
- `run.sh` — A Bash script to run your submission, as specified.

The following files are optional:

- `*.py`, `*.java`, `*.clj`, `*.kt`, `*.js`, `*.sh` — The Python, Java, Clojure, Kotlin, Node.js, or Bash source code files of a working program to find k-cores, as specified.

Any files other than these will be ignored.

[4]In this problem, the optimized quantity is the *size* of the k-cores: for correctness, they must be maximal.

Additionally:

- On the date before the due date, the grader will be run after 8pm and provides feedback containing only what your program scores. Only submissions made before 8pm are guaranteed to receive feedback. What your program scores on the official run (the due date) is the final score.
- Late submissions made within 24 hours of the deadline receive a .7 multiplier. If you decide to make a late submission, please notify me by sending an email to vnguy143@calpoly.eduwith both the subject and the body in the format: "CSC349,late,asgn<i>", i being the assignment number.