- *All codes must be written in <u>one</u> programming language, amongst C++, Python, Java.*
- *Source files must be sent at guillaume.ducoffe@fmi.unibuc.ro.*
- *Students must solve <u>one</u> subject. Partial solutions of multiple subjects will not be taken into account.*
- *All classroom material is allowed. Students may dispose of their own code, but they are not allowed to use others' codes. If two students share, even partially, the same code (and it is not the one provided in the correction), then both students automatically fail.*
- *Codes are tested using files with the following syntax:*
  - the number n of nodes and the number m of edges are given on the first line
  - every other line represents one edge. The two end-vertices of the edge are given and, if the graph is weighted, the line ends with the integer weight of the edge.
  Students may use in their code any implementation for their Graph class. However, they must provide a main function that reads on the terminal a file name, opens the corresponding graph file, transforms the latter in a Graph object, which is used as input for the students' program.
- *If additional inputs are needed, then they can either be set as constants in the code, or they can be asked to the user on the terminal.*
- *The notation is as follows:*
  - compilation 1p
  - execution 1p (for Python programmers, 2p)
  - correct output for the example graph file 2p
  - correctness of the algorithm 2p
  - time complexity (serial) or correct use of parallelism 2p
  - presence of justifications/explanations as comments in the code 2p

---

1) Consider an undirected unweighted connected graph. In a Median-Cardinality Search (MCS), vertices are visited one at a time. At evey step i, let C_i contain all unvisited vertices with at least one visited neighbour. For every vertex $u_i \in C_i$, let $p(u_i)$ be its number of visited neighbours. The next vertex to be visited must be a *median* of C_i, i.e., a vertex v_i such that at most half of the vertices $x \in C_i$ satisfy $p(x) \leq p(v_i)$, and at most half of the vertices $y \in C_i$ satisfy $p(y) > p(v_i)$.

   Implement MCS.
   <u>Complexity</u>: O(m*log(n))

2) Consider an undirected weighted graph. Algorithm LPA* is a variation of A*, where, for every (open or closed) node x, we retain two values:
   - g(x): the current best estimate of the distance from source node s to node x
   - rhs(x): the minimum value of g(y)+w(y,x) over all arcs yx (by convention, rhs(s) = 0).
   Initially, g(s) = ∞, rhs(s) = 0 and g(x) = rhs(x) = ∞ for every other node x.

   A node is inconsistent if rhs(x) ≠ g(x). During the execution of the algorithm, the set of OPEN nodes must contain exactly all nodes that are inconsistent (in particular, when the algorithm starts, source node s is put in OPEN because rhs(s) = 0).

   As for A*, we are given a heuristic function h (that must be one of the arguments in your function). Let us define, for every open node x, the following two-dimensional key:

key(x) = [min{ g(x), rhs(x) } + h(x), min{ g(x), rhs(x) }]

While the OPEN set is nonempty, we choose one open node x with minimum key (for the lexicographic order). Then, we update g(x), we update rhs(y) for every arc xy, and we close x.

Implement LPA*.
Complexity: at most O(d(x)+log(n)) to find and process the next open vertex x, with d(x) being the out-degree of x.

3) In Algorithm HCS for computing connected components, one has initially v.p = v for every vertex v (every tree contains one node). Then, the following loop is repeated until no more vertex v modifies her parent vertex v.p:
- `strong-parent-connect`
- repeatedly apply `shortcut` operation until no more vertex v modifies her parent vertex v.p
- `cleanup`
The operations `strong-parent-connect` and `cleanup` are new operations, presented below:

strong-parent-connect:
- for every vertex v in parallel, we first initialize v.n = ∞ (arbitrarily large value).
- we consider every edge vw in parallel. If v.p ≠ w.p, then we set: v.p.n = min{v.p.n,w.p}; and w.p.n = min{w.p.n,v.p}. Conflicts are handled with the Min. rule.
- finally, we consider every vertex v in parallel. If v.n ≠ ∞, then we set v.p = v.n.

cleanup:
- we consider every vertex v in parallel. If v.p > v and v.p.p = v (i.e., there is a loop), then we set v.p = v.

Implement Algorithm HCS. The min-rule must be simulated by using a reducer. No other synchronization mechanism should be used.

4) Recall the array implementation of a queue, using two variables in order to retain the front and back elements. Modify the bitmap PBFS algorithm in order to use this array implementation for the queue. In doing so, there should be no synchronization mechanism for dequeue operations.

5) Implement UFSCC, the parallel version of Purdom-Monroe algorithm seen in class in order to compute strong connected components.