1. We will give a simple randomized algorithm for the Global Min-Cut problem.
   (a) Recall the Global Min-Cut problem: given an unweighted, undirected graph $G$, find a cut $(A, B)$ (both nonempty) that minimizes the number of edges crossing.
   (b) Note that the Global Min-Cut problem is distinct from the Minimum $s$-$t$ Cut problem
      i. The graph is undirected and unweighted, whereas the Minimum $s$-$t$ Cut problem uses a weighted, directed graph
      ii. The value of the cut is just the number of edges, not the total weight of edges crossing from $A$ to $B$.
      iii. We are not required to return a cut with $s \in A$ and $t \in B$.
   (c) Nevertheless, we can solve the Global Min-Cut problem by creating directed edges of weight 1 in both directions, enumerating over all pairs of vertices, and taking the best $s$-$t$ cut. We will improve on this algorithm.

2. The basic Contraction Algorithm solves the Global Min-Cut problem with probability $1/(|V|^2)$.
   (a) When we contract an edge $(u, v)$ of our (undirected) (multi)graph $G$, we replace the vertices $u$ and $v$ with a new vertex $w$ such that for each edge $(u, x)$ or $(v, x)$ incident to $u$ or $v$ (but not joining $u$ and $v$), we add a $(w, x)$ edge. (In particular, e.g., if both $u$ and $v$ had an edge to $x$, both of these add a $(w, x)$ edge.)
   (b) The contraction algorithm is: $|V| - 2$ times, pick a random edge and contract it. Finally return the sets of vertices corresponding to the two remaining vertices.
   (c) The algorithm succeeds if and only if we never pick one of the edges of a minimum cut, since then the two supernodes must correspond to the sides of this cut.
   (d) If the minimum cut had size $k$, then the cut induced by each of these “supernodes” and the rest of the graph must have at least $k$ edges crossing, i.e., each has degree at least $k$; therefore, the number of edges on the $i$th iteration is at least $(|V| - (i-1))k/2$.
   (e) Therefore, the probability of not picking an edge of the cut on the $i$th iteration, given that we didn’t pick one previously, is at least $1 - \frac{2}{|V|-(i-1)}$. Thus, the probability that we never pick such an edge is at least $\prod_{i=1}^{\lceil \frac{|V|-2}{2} \rceil} (1 - \frac{2}{|V|-(i-1)}) = \frac{2}{|V|(|V|-1)}$

3. We can take an algorithm for an optimization problem that succeeds with (polynomially) small probability, like the contraction algorithm, and “amplify” its correctness to probability $1 - \delta$
   (a) If we find an optimal solution with probability $\epsilon$, and we repeat $\ell$ times and take the best solution, then we only fail to find an optimal solution with probability $(1 - \epsilon)^\ell$. 

1
(b) Therefore, here, \( \epsilon = 1/(|V|^2) \), and so if we pick \( \ell = \frac{1}{(1/2) \ln \frac{1}{3}} \), (using \( 1 + x \leq e^x \) for all \( x \)), we find that the probability of failing to find a minimum cut is at most \( \delta \), i.e., the probability of succeeding is at least \( 1 - \delta \).

(c) A similar trick can amplify correctness of a decision algorithm using the Chernoff bound.

4. Good data structures for maintaining the “supernode” sets are only good in an “amortized” sense – that is, we only bound the total time for a sequence of operations, not the worst-case time per operation.

(a) For the Contraction algorithm (and Kruskal’s Minimum Spanning Tree algorithm) we needed a “Union Find” data structure that supports the following operations

i. MakeSet\((v)\): returns the “name” of a set containing simply the element \( v \)

ii. Find\((\{v\})\): returns the “name” of the set currently containing the set that originally contained \( v \). Note that we can use this to check if two elements are contained in the same set by testing if \( \text{Find}(\{u\}) = \text{Find}(\{v\}) \).

iii. Union\((S_1, S_2)\): replaces the sets \( S_1 \) and \( S_2 \) with a new set \( S_1 \cup S_2 \) and returns the name of this new set (i.e., contracting two supernodes or joining two trees from a spanning forest into a single tree in Kruskal’s algorithm).

(b) There is a relatively simple implementation of Union Find that uses \( O(1) \) time for both MakeSet and Find, and takes \( O(\log n) \) amortized time over \( n \) Union operations. This is adequate to make Kruskal’s algorithm competitive with Prim’s algorithm (for example).

i. We represent sets by linked lists, with an extra field that points to the head of the list and an extra field that, in the head of the list, contains the size of the list.

ii. The head of the list will be the “name” of the set. So, MakeSet and Find are easily seen to be \( O(1) \) time with this setup.

iii. The Union operation is performed as follows. We take the head of the longer list as the head of the union set; we update its size to the sum of the sizes of the two lists, and walk the shorter list, updating each element to point to this new head. At the end, we attach the old second node of the larger set to the end of the list. The head’s next field is updated to point to the old head of the smaller set. Next time we’ll see the amortized analysis of this operation.