Remarks There is a number of Python libraries that are designed to deal with graphs. Some of them are NetworkX, graph-tool, and igraph. You are free to play with them if you want. (It is quite possible that NetworkX is already included in your Python distribution.) Nevertheless, in this lab session we can keep things simple. We will suppose that a graph $G = (V, E)$ has n vertices denoted $V \coloneqq \{0, 1, \ldots, n-1\}$. As a result, a graph G can be represented as a list of edges, where an edge is a tuple (u, v) containing two vertices, $0 \leq u < v \leq n-1$.

Erdős–Rényi model

The Erdős–Rényi model is a simple model of generating a random graph. In this model, we take a number $0 < p \leq 1$ and we construct a graph with n vertices as follows. For every possible edge (u, v) we make a random decision: with probability p the edge belongs to the graph, and with probability $1-p$ it does not belong to the graph. A graph obtained in this way is often denoted $G(n, p)$.

Exercise 1.

(a) Write a function `erdos_renyi(n,p)` that creates a random graph from the Erdős–Rényi model $G(n, p)$. (Hint: you may want to use `numpy.random.binomial` and `itertools.combinations`. For efficiency, you should use `numpy.random.binomial` only once.)

(b) Suppose that $p \coloneqq 4/n$. How many edges, on average, does this graph have? Test your function for different values of n. How efficient it is? Can you use it to quickly construct a random graph with 5 000 vertices? 10 000? What about 30 000 vertices?

(c) Implement the function `sparse_erdos_renyi(n,p)` given in Figure 1 (fell free to correct all the bugs that you find). This function also computes a graph from the Erdős–Rényi model. (Do you see why?) Test its performance. How large graphs can you compute quickly? Can you explain the difference in efficiency between the two functions on our examples?

Connected components and spanning trees

In this section, we investigate the Kruskal algorithm. It is a greedy algorithm that is capable of finding a spanning tree of minimal weight of a given graph. Nevertheless, we may also use it to find the connected components of a graph. The basic algorithm that we are going to use is presented in Figure 2. For every vertex it uses two auxiliary data: Representative(v) is a vertex that represents the connected component of v and Component(v) is a (partial) list of vertices that are in the same component as the vertex v. At the end of the algorithm, Component(Representative(v)) is the list of vertices of the connected component of v.

Exercise 2.

(a) Write a function `connected_components(n,E)` that implements the algorithm of Figure 2 and test it. Then, use it on Erdős–Rényi graphs (as previously, we take $p \coloneqq 4/n$). How efficient is this function?

(b) Modify the function `connected_components(n,E)` between the lines denoted by 3 and 4 in such a way that when the merging operation of line 4 is performed, the smaller component is added to the bigger one. How does this influence the efficiency of your function?

(c) By taking different values of n, try to find out what is the average size (number of vertices) of the largest connected component of a graph $G(n, 4/n)$.

```
def all_pairs(k,n):
# Enumerates all possible edges of a graph with n vertices
    m = int(n*(n-1)/2)
    k = int(m - k - 1)
    w = math.floor((math.sqrt(1+8*k) - 1)/2)
    t = int((w**2 + w) / 2)
    u = n - 2 - w
    v = n - 1 - k + t
    return (u, v)

def sparse_erdos_renyi(n,p):
# Computes an Erdos-Renyi graph
    E = []
    m = int(n*(n-1)/2)
    L = np.random.geometric(p,int(2*n*n*p))
    i = 0
    t = L[0] - 1
    while t < m:
        E.append(all_pairs(t,n))
        i += 1
        t += L[i]
    return E
```

Figure 1: Computing sparse Erdős–Rényi graphs.

```
  Data: List of edges $E$ of a graph $G = (V, E)$
  Result: List of connected components of $G$
  for $v \in V$ do
      Representative($v$) $\leftarrow v$;
      Component($v$) $\leftarrow [v]$;
  end
1 for $(u, v) \in E$ do
2     $u_{\text{aux}}, v_{\text{aux}} \leftarrow$ Representative($u$), Representative($v$);
3     if $u_{\text{aux}} \neq v_{\text{aux}}$ then
4         Component($u_{\text{aux}}$) $\leftarrow$ Component($u_{\text{aux}}$) + Component($v_{\text{aux}}$);
          for $w \in$ Component($v_{\text{aux}}$) do
              Representative($w$) $\leftarrow u_{\text{aux}}$;
          end
      end
  end
  Connected_components $\leftarrow$ [];
  for $v \in V$ do
      if $v =$ Representative($v$) then
          Connected_components.append(Component($v$))
      end
  end
```

Figure 2: Computing connected components.

Exercise 3. We now turn our attention to finding a minimum spanning tree of a connected graph using the Kruskal algorithm. To do so we will suppose that an edge of a graph is represented as (u, v, z), where $u, v \in V$ are vertices of a graph and $z \in \mathbb{Q}$ is the weight of the edge.

(a) Write a function `minimum_spanning_tree(n,E)` that outputs the minimum spanning tree T of a graph. To do so, do the following modifications to the function `connected_components(n,E)`. First, sort the edges by their weights, and read them from the lightest to the heaviest (line `1` of the algorithm). Second, whenever the line `4` is reached, add the edge (u, v) to T. (Hint: you may want to read `https://wiki.python.org/moin/HowTo/Sorting` to know how to sort the edges.)

(b) To visualize your results do the following test. Pick n random points from the square $[0, 1] \times [0, 1]$. Then, construct an edge between every pair of points, with the weight equal to the distance of the two points. Compute a minimum spanning tree and plot the resulting image.