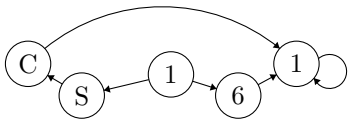
CS 161 **Problem Set 5**
Spring 2017 **Due:** May 19, 2017, 3pm

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy,** as well as for **helpful advice** for how to write up your solutions.

Note: For all problems, if you include pseudocode in your solution, please also include a brief English description of what the pseudocode does.

Drawing graphs: You might try `http://madebyevan.com/fsm/` which allows you to draw graphs with your mouse and convert it into LaTeX code:



---

1. **(Examples.)** (4 points) Give one example of a directed graph on four vertices, A, B, C, and D, so that both depth-first search and breadth-first search discover the vertices in the same order when started at A. Give one example of an directed graph where BFS and DFS discover the vertices in a different order when started at A. Above, *discover* means the time that the algorithm first reaches the vertex. Assume that both DFS and BFS iterate over outgoing neighbors in alphabetical order.

   [We are expecting: a drawing of your graphs and an ordered list of vertices discovered by BFS and DFS.]

2. **(In-order traversal)** (6 points) In class, we stated the fact that Depth-First Search can be used to do in-order traversal of a binary search tree. That is, given a binary search tree T containing n distinct elements $a_1 < a_2 < \cdots < a_n$, DFS can be used to return an array $[a_1, a_2, \ldots, a_n]$ containing these elements in sorted order. Call this algorithm `inOrderTraversal`. Work out the details for this algorithm, and answer the following questions:

   (a) (3 pts) Give pseudocode for `inOrderTraversal`. For your pseudocode, assume that each node in T has a key value (one of the a_i) and pointers to its left and right children, and that if a vertex has no child this is represented as a `NIL` child.

   (b) (2 pts) What is the asymptotic running time of `inOrderTraversal`, in terms of n? We're looking for a statement of the form "the running time is $\Theta(\_\_\_)$."

   (c) (1 pt) Does the algorithm need to look at the values $a_1, \ldots, a_n$ (other than to return the values)?

   [We are expecting: just the answers to these questions, no justification needed.]

3. **(Too good to be true)** (6 points) Your friend claims to have built a new kind of binary search tree with $O(1)$ (deterministic) insertion time (better than the $O(\log(n))$ of red-black trees), which can handle arbitrary comparable objects. How do you know they are wrong?

   [We are expecting: a short but formal proof that your friend is wrong.]

4. **(Dijkstra and negative edge weights)** (15 points) Let $G = (V, E)$ be a weighted directed graph. For the rest of this problem, assume that $s, t \in V$ and that **there is a directed path from s to t.** The weights on G could be anything: **negative, zero, or positive.**

   For the rest of this problem, refer to the implementation of Dijkstra's algorithm given by the pseudocode below.

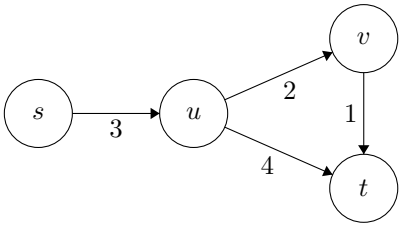
```
Dijkstra_st_path(G, s, t):
    for all v in V, set d[v] = Infinity
    for all v in V, set p[v] = None
    // we will use the information p[v] to reconstruct the path at the end.
    d[s] = 0
    F = V
    D = []
    while F isn't empty:
        x = a vertex v in F so that d[v] is minimized
        for y in x.outgoing_neighbors:
            d[y] = min( d[y], d[x] + weight(x,y) )
            if d[y] was changed in the previous line, set p[y] = x
        F.remove(x)
        D.add(x)

    // use the information in p to reconstruct the shortest path:
    path = [t]
    current = t
    while current != s:
        current = p[current]
        add current to the front of the path
    return path, d[t]
```

   Notice that the pseudocode above differs from the pseudocode in the notes: first it runs Dijkstra's algorithm (as presented in the notes), and then it uses this to compute the shortest path from s to t. It returns the shortest path and the cost of that path.

   (a) (2 points) Step through `Dijkstra_st_path`(G, s, t) on the graph G shown below. Complete the table below to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is.

G



   [We are expecting: the table below filled out, as well as the final shortest path and its cost. No further justification is required.]

	d[s]	d[u]	d[v]	d[t]	p[s]	p[u]	p[v]	p[t]
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	∞	None	s	None	None
Immediately after the second element of D is added, the state is:								
Immediately after the third element of D is added, the state is:								
Immediately after the fourth element of D is added, the state is:								

(b) (3 points) **Prove or disprove**: In every such graph G, the shortest path from s to t exists. Here, a *path* from s to t is formally defined as a sequence of edges

$$(u_0, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_{M-1}, u_M)$$

so that $u_0 = s$, $u_M = t$, and $(u_i, u_{i+1}) \in E$ for all $i = 0, \ldots, M-1$. A *shortest path* is a path $((u_0, u_1), \ldots, (u_{M-1}, u_M))$ so that

$$\sum_{i=0}^{M-1} \text{weight}(u_i, u_{i+1}) \leq \sum_{i=0}^{M'-1} \text{weight}(u'_i, u'_{i+1})$$

for all paths $((u'_0, u'_1), \ldots, (u'_{M'-1}, u'_{M'}))$.

(c) (3 points) **Prove or disprove**: In every such graph G in which the shortest path from s to t exists, `Dijkstra_st_path`(G, s, t) returns a shortest path between s and t in G.

(d) (3 points) **Prove or disprove**: In every such graph G in which there is a negative-weight edge, and for all s and t, `Dijkstra_st_path`(G, s, t) does not return a shortest path between s and t in G.

(e) (4 points) Your friend offers the following way to patch up Dijkstra's algorithm to deal with negative edge weights. Let G be a weighted graph, and let w^* be the smallest weight that appears in G. (Notice that w^* may be negative). Consider a graph $G' = (V, E')$ with the same vertices, and so that E' is as follows: for every edge $e \in E$ with weight w, there is an edge $e' \in E'$ with weight $w - w^*$. Now all of the weights in G' are non-negative, so we can apply Dijkstra's algorithm to that:

```
modifiedDijkstra(G,s,t):
     Construct G' from G as above.
     return Dijkstra_st_path(G',s,t)
```

Prove or disprove: Your friend's approach will always correctly return a shortest path between s and t if it exists.

[We are expecting: for each "prove or disprove," either a proof or a (small) counterexample. In the previous sentence, "(small)" means no more than 5 vertices.]

5. **(Social engineering)** (13 points) Suppose we have a community of n people. We can create a directed graph from this community as follows: the vertices are people, and there is a directed edge from person A to person B if A would forward a rumor to B. Assume that if there is an edge from A to B, then A will always forward any rumor they hear to B. Notice that this relationship isn't symmetric: A might gossip to B but not vice versa. Suppose there are m directed edges total, so $G = (V, E)$ is a graph with n vertices and m edges.

   Define a person P to be *influential* if for all other people A in the community, there is a directed path from P to A in G. Thus, if you tell a rumor to an influential person P, eventually the rumor will reach everybody. You have a rumor that you'd like to spread, but you don't have time to tell more than one person, so you'd like to find an influential person to tell the rumor to.

   In the following questions, assume that G is the directed graph representing the community, and that you have access to G as an array of adjacency lists: for each vertex v, in $O(1)$ time you can get a pointer to the head of the linked lists v.`outgoing_neighbors` and v.`incoming_neighbors`. Notice that G is not necessarily acyclic. In your answers, you may appeal to any statements we have seen in class, in the notes, or in CLRS.

   (a) (2 pts) Show that all influential people in G are in the same strongly connected component, and that everyone in this strongly connected component is influential.

   [We are expecting: a short but formal proof.]

   (b) (8 pts) Suppose that an influential person exists. Give an algorithm that, given G, finds an influential person in time $O(n + m)$.

   [We are expecting: pseudocode, a proof of correctness, and a short argument about the runtime.]

   (c) (3 pts) Suppose that you don't know whether or not an influential person exists. Use your algorithm from part (b) to give an algorithm that, given G, either finds an influential person in time $O(n + m)$ if there is one, or else returns "no influential person."

   [We are expecting: pseudocode, and a short argument for both correctness and runtime. You do not need to re-write your algorithm from part (b), you can just call it.]