Problem Set 6

All parts are due on October 25, 2018 at 11PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

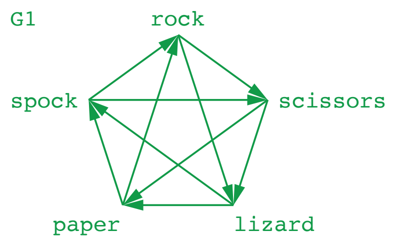
Problem 6-1. [16 points] Graph Mechanics

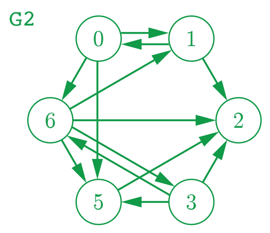
(a) [4 points] Draw the **directed** graph associated with each of the following graph representations.

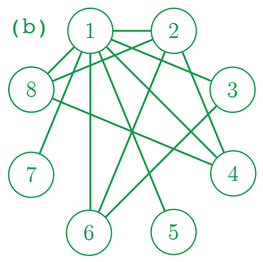
```
G1 = {"rock"     : ["scissors", "lizard"],
      "paper"    : ["rock"    , "spock"],
      "scissors" : ["paper"   , "lizard"],
      "lizard"   : ["paper"   , "spock"],
      "spock"    : ["rock"    , "scissors"]}

G2 = [[1,5,6], [0,2], [], [6,2,5], None, [2], [1,2,3,5]]
```

Solution:







Rubric:

- 2 points for any graph drawing depicting correct vertices for G_1
- -1 point for each incorrect or missing edge in G_1, minimum zero points
- 2 points for any graph drawing depicting correct vertices for G_2
- -1 point for each incorrect or missing edge in G_2, minimum zero points

(b) [4 points] The ***order-k division graph*** is the undirected graph on vertices $\{1, \ldots, k\}$, with an edge connecting two distinct vertices if and only if one vertex is a factor of the other. For example, the order-4 division graph contains the edge $\{4, 2\}$ but not $\{4, 3\}$. Draw a picture of the order-8 division graph, and write its graph representation as a direct access array of adjacency lists, as in `G2` above. Double check your graph: it should have exactly 12 edges.

Solution:

```
O8 = [None, [2,3,4,5,6,7,8], [1,4,6,8], [1,6],
      [1,2,8], [1], [1,2,3], [1], [1,2,4]]
```

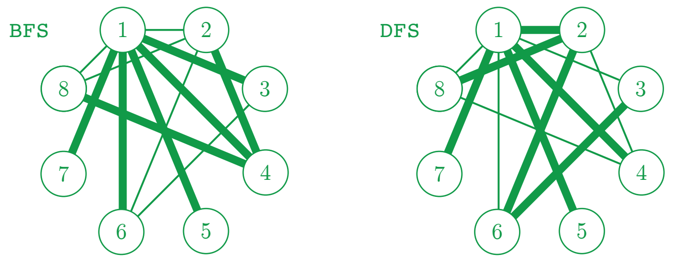
Rubric:

- 2 points for any graph drawing depicting correct vertices
- -1 point for each incorrect or missing edge, minimum zero points
- 2 points for any array of adjacency lists
- -1 point for each mistake in lists, minimum zero points

(c) [8 points] Run breadth-first search and depth-first search on the order-8 division graph, starting at node 4. When performing each search, visit the neighbors of each vertex in increasing order. Draw the parent tree constructed by each search, and list the order in which nodes were first visited.

Solution:

```
BFS: [4, 1, 2, 8, 3, 5, 6, 7]
DFS: [4, 1, 2, 6, 3, 8, 5, 7]
```



Rubric:

- 2 points for each parent pointer tree on correct vertices
- -1 point for each incorrect tree edge, minimum zero points per graph
- 2 points for each first-visited list
- -1 point for each mistake in a list, minimum zero points per list

Problem 6-2. [7 points] **Graph Radius**

In any undirected graph $G = (V, E)$, the ***radius*** $r(u)$ of a vertex $u \in V$ is the distance to the farthest other vertex v, i.e. $r(u) = \max\{\delta(u, v) \mid v \in V\}$; while the ***radius*** $R(G)$ of an undirected graph $G = (V, E)$ is the smallest radius of any vertex, i.e. $R(G) = \min\{r(u) \mid u \in V\}$. Describe a $O(|V||E|)$-time algorithm to compute the radius of a given **connected** undirected graph.

Solution: For each vertex u: run breadth-first search to compute $\delta(u, v)$ for all $v \in V$ in $O(|E| + |V|)$ time, and then loop through the $\delta(u, v)$ and compute the maximum, $r(u)$, in $O(|V|)$. Since the graph is connected, $|V| = O(|E|)$, so $O(|E| + |V|) = O(|E|)$, so computing all $\delta(u, v)$ in this way takes $O(|V||E|)$ time. Lastly, loop through $r(u)$ for each $v \in V$ and compute the minimum in $O(|V|)$ time.

Rubric:

- 5 points for description of a correct algorithm
- 2 points for correct running time analysis
- Max 4 points for only $O(|V|(|V| + |E|))$ analysis
- Partial credit may be awarded

Problem 6-3. [8 points] **Tramak Railway**

Railey P. Trainston has acquired many local railroad networks across the country, and has combined them to form **Tramak**, a single national railway system. A Tramak ***route*** is a two-way train service between two cities having no other train stop between them. A city is ***reachable*** from another city on the Tramak network if there is a sequence of connected routes from one to the other. Each of the country's major cities participates in **at least** one route from some acquired local railroad, but not all cities are reachable from each other. Given a list of routes from the acquired local networks, describe a linear-time algorithm to determine the minimum number of new routes that Railey must add to ensure that every city is reachable from the capital. Specify whether your algorithm's running time is worst-case or expected, and state any assumptions you make.

Solution: We observe that the existing railroad network can be represented by a disconnected, undirected graph, where vertices are cities, with an edge between two cities if there is an existing Tramak route between them. If there are k connected components in this graph, we must add $k - 1$ new routes between pairs of connected components to make the whole graph connected. Thus this problem reduces to counting the number of connected components in a disconnected undirected graph. We can do this by looping through all vertices, searching from any vertex not already visited, using either breadth-first or depth-first search. Each time the loop searches from a not-yet-visited vertex, it must exist in a different component from any other vertex already processed, as searches from previous vertices will visit all vertices reachable from them. Both search algorithms take linear time to explore a connected component, and by not searching previously visited vertices we ensure that no search explores any vertex more than once, so this algorithm runs in $O(r)$ time, where r is the number of original routes. Note the number of cities is upper bounded by twice the number of routes, as each city is connected to at least one route and each route connects two cities. We can run breadth-first search in worst-case linear time by assigning each city a unique integer identifier in a linear range with respect to the number of cities, using a direct access array to access vertex adjacency lists. Alternatively, we can run the search in expected linear time by using a hash table to look up cities (say by name).

Rubric:

- 5 points for description of a correct algorithm
- 2 points for correct running time analysis
- 1 point for correct discussion of worst-case/expected
- Partial credit may be awarded

Problem 6-4. [12 points] **Anti-Aloft Aviation**

The Wleft brothers have just launched a new airline featuring their own fixed-wing aircraft, with flights between n locations. They make a list of the f flights operated by their airline, where each flight is defined by a source airport, a destination airport, and the number of minutes of flying time: an integer between 1 and k inclusive. The Wleft brothers want to plan a route from their hometown of Dayton, Ohio to Cambridge, Massachusetts for their upcoming talk at MIT LSC. To maximize safety, they want to minimize the total number of minutes they spend flying, not caring about the number of connections nor any layover time.

(a) [4 points] Describe a $O(n + kf)$-time algorithm to help the brothers compute the minimum possible flying time from Dayton to Cambridge, using breadth-first search.

Solution: Construct a weighted, directed graph G with a vertex for each of the n airports, and a directed edge for each of f flights connecting from its source airport to destination airport, weighted by the flight time in minutes. Then replace each edge (u, v) having weight x, with an unweighted directed path of x edges connecting from u to v, to form graph G'. After this transformation, G' has at most fk edges and at most $n + f(k - 1)$ vertices. Then the minimum time to fly from Detroit to Cambridge then corresponds to the length of a shortest path in G', which we can find using breadth-first search in $O(n + fk)$ time.

Rubric:

- 2 points for description of a correct algorithm
- 1 point if correct algorithm runs in $O(kn + f)$ time
- 1 point for correct running time analysis
- Partial credit may be awarded

(b) [8 points] Describe a $O(kn + f)$-time algorithm to help the brothers compute the minimum possible flying time from Dayton to Cambridge, using breadth-first search. (Hint: replace each vertex with a path.)

Solution: Construct a graph G in the following way. For each airport a, construct a directed path $\pi_a = (a_0, a_1, \dots, a_{k-1})$. For each flight from a source airport a to destination airport b lasting m minutes, add a directed edge from vertex a_{m-1} to vertex b_0. After this transformation, G has $n(k - 1) + f$ edges and nk vertices. Then the minimum time to fly from Detroit to Cambridge then corresponds to the length of a shortest path in G, which we can find using breadth-first search in $O(nk + f)$ time.

Rubric:

- 4 points for description of a correct algorithm
- 2 points if correct algorithm runs in $O(kn + f)$ time
- 2 points for correct running time analysis
- Partial credit may be awarded

Problem 6-5. [12 points] **Three-Heart Challenge**

Princess Lelda is on a quest to save her friend Zink from the evil force of Anon. Luckily, the roaming shopkeeper Beetle sold her a map of Anon's underworld headquarters, which consists of various ***caves*** and ***tunnels***: bidirectional connections between the caves. Lelda's goal is to reach the cave where Zink is being held, starting at one of many possible cave entrances. The map marks some caves as ***dangerous***, which indicates that if Lelda enters that cave, she would need to fight enemies resulting in exactly one heart of damage. Lelda can take at most two hearts of damage on her way to Zink; taking three hearts of damage would lead to a "game over" situation. Describe a linear-time algorithm to help Lelda find a route to Zink, if such a route exists.

Solution: Construct a graph G where each cave c corresponds to three vertices c_0, c_1, and c_2 representing a state where Lelda enters cave c with 0, 1 or 2 hearts respectively. Then, add a directed edge from vertex a_i to vertex b_i if cave a is not dangerous and there is a tunnel connecting caves a and b. These edges correspond to entering cave a and leaving along a tunnel to b without loosing a heart. Then add a directed edge from vertex a_i to vertex b_{i+1} if $i < 2$, cave a is dangerous, and there is a tunnel connecting caves a and b. These edges correspond to entering cave a and leaving along a tunnel to b while losing exactly one heart. Let z be the cave where Zink is being held. Add a vertex s connected by a directed edge to c_0 for each cave c which is a cave entrance. Lastly, add a vertex t connecting to a directed edge from z_i for $i \in \{0, 1\}$, and from z_2 if cave z is not dangerous.

We now argue that there is a route to Zink that loses at most two hearts if and only if there is a path from s to t in G. First, given a path from s to t, list the caves corresponding to the vertices visited in order along the path. This list of caves corresponds to a path from a cave entrance to Zink that loses at most two hearts by construction. Second, given a path π from a cave entrance to Zink which loses at most two hearts, we can construct a path in G by taking each cave c in π and mapping it to cave c_i in G where i indicates the number of hearts Lelda has when entering cave c along π (and adding in s and t).

Thus this problem reduces to finding a path from s to t in unweighted graph G, which we can solve using breadth-first search. If C is the number of caves and T is the number of tunnels, G has $O(C)$ vertices and $O(C + T)$ edges, so this algorithm takes $O(C + T)$ time.

Alternative solution sketch: Because we are only looking for the existence of a path and not a shortest path, one can also solve this problem by identifying connected components of caves that are reachable from each other without losing any hearts, contracting each to a single node, and then running BFS in the remaining graph, returning if there exists a path traversing no more than two edges. This algorithm also take $O(C + T)$ time by a similar analysis.

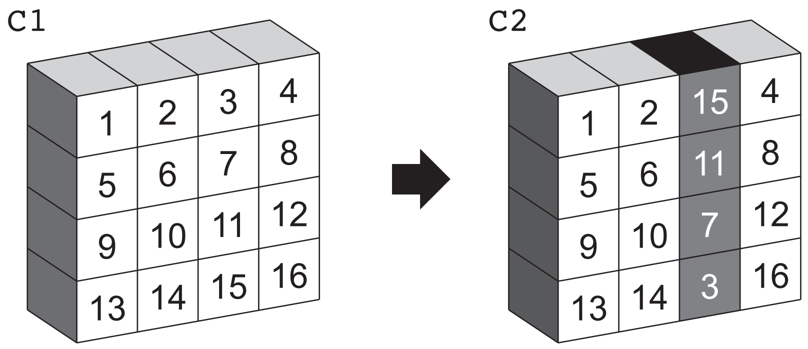
Rubric:

- 8 points for description of a correct algorithm
- 2 points if correct algorithm runs in linear time, i.e. $O(C + T)$
- 2 points for correct running time analysis
- Partial credit may be awarded

Problem 6-6. [45 points] **Rubik's Squares**

The ***Rubik's cube*** is a puzzle which consists of a $3 \times 3 \times 3$ grid of smaller cubes called ***cubies*** which can be rotated relative to each other to form different multi-colored configurations of the cube. In the solved configuration, each 3×3 face of the Rubik's cube is monochromatic, and each of the six faces is a different color. The goal of the puzzle is to return the cube to the solved configuration via a sequence of moves.

In this problem, we will study ***Rubik's k-Square*** puzzles, consisting of a $k \times k \times 1$ grid of cubies, where the solved configuration consists of six monochromatic faces. From any Rubik's k-Square configuration, there are $2k$ possible ***moves*** to transform the configuration into a new configuration. Each move changes the position and orientation of exactly one row or column of cubies from a input configuration in the following way: to move row (or column) i of a configuration, reverse the order of the cubies within that row (or column), and flip the orientation of each cubie in the row (or column) upside-down. Below shows the result of a column-2 move applied to the solved configuration of a Rubik's 4-Square.



We will represent a Rubik's k-Square configuration via a length-k tuple of length-k tuples, where the y-th inner tuple corresponds to the y-th row of the configuration, and the x-th element of row y corresponds to the cubie positioned at row y and column x. We represent each cubie as an integer between 1 and k^2 inclusive, where integer i represents the cubie that exists in row $y = \lfloor (i-1)/k \rfloor$ and column $x = i - ky - 1$ in the solved configuration. In addition, we use positive i when the cubie in the configuration has the same orientation as in the solved configuration, and negative i when the cubie is upside-down. For example, the two configurations shown above have the following representations:

```
C1 = (( 1,  2,  3,  4),         #      C2 = (( 1,  2,-15,  4),
      ( 5,  6,  7,  8),         #            ( 5,  6,-11,  8),
      ( 9, 10, 11, 12),         #            ( 9, 10, -7, 12),
      (13, 14, 15, 16))         #            (13, 14, -3, 16))

C2 = move(C1, ("col", 2))
```

(a) [4 points] Describe an $O(k^2)$-time algorithm to determine whether a given Rubik's k-Square configuration is solved.

Solution: The solved configuration has all positive numbers, with each number in reading order across rows then columns. To check if a configuration is solved, loop through the rows, then loop through the columns, making sure that the first number is 1 and each subsequent number is exactly one more than the previous. This procedure checks the value of each of the k^2 element in the configuration in $O(1)$ time, so it takes $O(k^2)$ time in total.

Rubric:

- 3 points for description of a correct $O(k^2)$-time algorithm
- 1 point for correct running time analysis
- Partial credit may be awarded

(b) [4 points] Describe an $O(k^3)$-time algorithm to compute all configurations reachable from a given Rubik's configuration via a single move.

Solution: There are $2k$ possible moves `(s, i)` for `s`∈`("row", "col")` and `i`∈ $\{0, \dots, k-1\}$. For each move, copy the input configuration in $O(k^2)$ time, and then reverse and negate the row or column affected by the move. Doing this for all moves then takes $O(k^3)$ time.

Rubric:

- 3 points for description of a correct $O(k^3)$-time algorithm
- 1 point for correct running time analysis
- Partial credit may be awarded

(c) [4 points] Show that a Rubik's k-Square has at most c^{k^2} distinct configurations, for some constant c.

Solution: Each coordinate of a cubie originally at position (x, y) is transformed by a move by a reflection, so can only ever be in row y or $(k-1)-y$ and column x or $(k-1)-x$, i.e. one of four positions. In addition, a cubie may be rightside-up or upside-down, so each cubie may exist in at most one of eight oriented positions, leading to at most 8^{k^2} configurations of the puzzle. A tighter upper bound of 4^{k^2} is possible by noting that a cubie can only exist in a position in one orientation.

Rubric:

- 3 points for correctly bounding number of states of a cubie by some fixed constant
- 1 points for correctly arguing the requested bound on top of that
- Partial credit may be awarded

(d) [8 points] Describe an $O(c^{k^2}k^3)$-time algorithm to find the shortest sequence of moves to solve a Rubik's k-Square.

Solution: Run breadth-first search on a graph having vertices corresponding to configurations of the Rubik's k-Square and an edge between two configurations when one can be transformed into the other via a single move. Start the search with the intial configurations, and when the solved configuration is found, return a shortest solving sequence of moves to reach the solved state by following parent pointers along the BFS tree back to the original configuration. When processing a configuration, compute its neighboring configurations using the algorithm from part (b) in $O(k^3)$ time. Since breadth-first search processes each of $O(c^{k^2})$ vertices at most once, and $O(k^3)$ work is performed per vertex, this search takes $O(c^{k^2})$ time.

Rubric:

- 5 points for description of a correct algorithm
- 2 points if correct algorithm runs within requested bound
- 1 points for correct running time analysis
- Partial credit may be awarded

(e) [25 points] Implement your algorithm by writing functions `is_solved`, `move`, and `solve_ksquare` in the code template provided. The input to `solve_kquare` will be a Rubik's k-square configuration, and it should return a list of moves where: (1) each move is of the form `(s, i)` for some `s` $\in \{$`"row"`,`"col"`$\}$ and `i` $\in \{0, \ldots, k-1\}$, and (2) applying the sequence of moves in order to the input configuration results in the solved configuration. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
def is_solved(config):
    i, k = 1, len(config)
    for y in range(k):
        for x in range(k):
            if i != config[y][x]:
                return False
            i = i + 1
    return True

def move(config, mv):
    k = len(config)
    (s, i) = mv
    new_config = [[config[y][x] for x in range(k)] for y in range(k)]
    if s == "row":
        row = [new_config[i][x] for x in range(k)]
        for x in range(k):
            new_config[i][x] = -row[k - x - 1]
    else:
        col = [new_config[y][i] for y in range(k)]
        for y in range(k):
            new_config[y][i] = -col[k - y - 1]
    return tuple([tuple(row) for row in new_config])

def solve_ksquare(config):
    parent_move = {config: None}
    level = [[config]]
    while 0 < len(level[-1]):
        level.append([])
        for current in level[-2]:
            if is_solved(current):
                moves = []
                while parent_move[current]:
                    (s, i) = parent_move[current]
                    moves.append((s, i))
                    current = move(current, (s, i))
                return moves[::-1]
            for s in ("row", "col"):
                for i in range(len(config)):
                    neighbor = move(current, (s, i))
                    if neighbor not in parent_move:
                        parent_move[neighbor] = (s, i)
                        level[-1].append(neighbor)
    return None
```