Problem Set 4

This problem set is due **Wednesday, November 7** at **11:59PM**.
Note the one day extension!!

Code for problem 4, as well as a solution template for problem 4 and a LaTeXtemplate for the written problems, have been posted to the course website. Solutions should be turned in at `https://alg.csail.mit.edu` (our submission site).

Programming questions will be graded on a collection of test cases (including example test cases to help you debug). Unless you see an error message, *you will be able to see your grade immediately*. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within certain time and space bounds. You may submit as many times as you'd like, and only the final submission counts! **Therefore, make sure your final submission is what you want it to be.**

For written questions, full credit will be given only to correct solutions that are described clearly *and concisely*.

Problem 4-1. [20 points] **Search**
For each of the following proposed algorithms or statements, prove its correctness or give a small counterexample to show that it is false. You may use LaTeXto draw example graphs if necessary (the solution template contains a drawing to get you started).

(a) [5 points] **Algorithm.** Suppose there is a vertex *v* in a directed graph $G(V, E)$ such that there exists a directed path from *v* to all other vertices in G. To determine the presence of a directed cycle in G, do a BFS from *v*. If a vertex is seen twice by the BFS, then there is a directed cycle. If no vertices are seen more than once, then there is not a directed cycle.

(b) [5 points] **Algorithm.** Suppose there is a vertex *v* in an undirected graph $G(V, E)$ such that there exists a path from *v* to all other vertices in G. To determine the presence of a cycle in G, do a BFS from *v*. If a vertex is seen twice by the BFS, then there is a cycle. If no vertices are seen more than once, then there is not a cycle.

(c) [5 points] **Statement.** Suppose there is a directed graph $G(V, E)$ with nodes *r, u, v*. If a DFS traversal of G from *r* has a lower finishing time for *u* than for *v*, then *v* is an ancestor of *u* in the DFS tree rooted at *r*.

(d) [5 points] **Statement.** An undirected graph is called *bipartite* if its nodes can be assigned one of two colors (*i.e.* red or blue) such that no two nodes of the same color are adjacent. Consider the modified BFS algorithm on the next page. It correctly determines whether a graph is bipartite.

Hint: You may use the fact (which you may remember from 6.042) that an undirected graph is bipartite if and only if there is no odd-length cycle.

Algorithm 1 Problem 4-1 Part D: Modified BFS

```
procedure BFS($G$)
    for $u \in G.V$ do
        $u.color = WHITE$
        $u.d = \infty$
        $u.\pi = NIL$
    end for
    $s = G.V[0]$
    $s.color = BLUE$
    $s.d = 0$
    $Q = \emptyset$
    ENQUEUE(Q,S)
    while $Q \neq \emptyset$ do
        u = DEQUEUE(Q)
        for $v \in G.Adj[u]$ do
            if $v.color == WHITE$ then
                //v's color may already be red or blue
                $v.color = \{RED, BLUE\} - u.color$
                $v.d = u.d + 1$
                $v.\pi = u$
                ENQUEUE(Q,V)
            else if v.color == u.color then
                PRINT 'NOT BIPARTITE!'
                RETURN FALSE
            end if
        end for
    end while
    RETURN TRUE
end procedure
```

Problem 4-2. [20 points] **Numerics**

(a) [10 points] Recall that Newton's method can be used to estimate $\sqrt{2}$ by searching for the minimum of the function $f(x) = x^2 - 2$. The iterative step for this optimization is

$$x_{i+1} = \frac{x_i + \frac{2}{x_i}}{2}.$$

Suppose that the initial guess for x is $x_0 = n > 10$. What is the number of iterations k needed for x_k to fall within the range $[1, 10)$? Provide an asymptotic (i.e. $\Theta(.)$) bound for k and justify your answer by providing a formal argument.

(b) [10 points] Newton's method, although simple to understand and easy to implement, does not guarantee convergence.

Consider

$$f(x) = \begin{cases} x^{\frac{1}{2}} & : x > 0 \\ -|x|^{\frac{1}{2}} & : x < 0 \\ 0 & : x = 0 \end{cases}$$

Explain why Newton's method fails to converge to the root $x_r = 0$ for this function, unless the initial guess was $x_0 = 0$.

Problem 4-3. [30 points] **Shove-aside Hashing**

Consider the following variant on cuckoo-hashing, called shove-aside hashing. As in the case for double hashing, the probe sequence for a key k in a table $T[0, ..., m-1]$ is defined as

$$h(k, i) = h_1(k) + i * h_2(k) \mod m,$$

for $i = 0, 1, ..., m-1$ such that for all k, $h_2(k)$ is relatively prime to m.

Lookup: Like in double hashing, we follow the probe sequence until k is found, or the probe finds an empty slot.

Insertion: The key k being inserted is forced into $h(k, 0)$; if it is displacing another key k_1, k_1 is forced into the next slot in its probe sequence. Then if k_2 is displaced by k_1, k_2 is forced into the next slot in its own probe sequence, etc..

Deletion: Deletion is not supported, in shove-aside hashing[1].

Now, answer the following questions about shove-aside hashing:

(a) [5 points] Given hash functions h_1 and h_2 as defined in the problem, provide pseudocode for shove-aside hashing as defined above, assuming that the table still has empty slots.

(b) [10 points] Prove that insertion using this hashing algorithm will always terminate, and that when it terminates, all keys inserted will be findable via the lookup procedure described. Again, assume that the table still has empty slots.

(c) [15 points] Prove or disprove the following statement: the result of inserting n keys $k_1, k_2, ...k_n$ into the hash table using shove-aside hashing will result in the same hash table as if double hashing were used with the keys inserted in reverse order, $k_n, k_{n-1}, ..., k_1$.

[1]But it's possible to modify shove-aside hashing so it is possible!

Problem 4-4. [40 points] **Land of the Zoombinis: Bubblewonder Abyss**

The zoombinis have reached the final leg in their journey to Zoombiniton, Bubblewonder Abyss. Now they must conquer the abyss, and you must help them.

To help you, we've mapped Bubblewonder Abyss into an algorithms problem. The abyss can be represented by a large rectangular $n \times n$ grid. Objects occupy cells in the grid. The only things in the abyss are the zoombinis, boulders, and exits (they fell through a hole into the abyss so there was no entrance). The goal is for the all the zoombinis to get to an exit (when a zoombini reaches an exit, they leave the abyss and no longer exist on the map). There is a catch, however. The floor of the abyss is extremely slippery, such that if a zoombini moves in one direction, they will continue moving in that direction either until they encounter an obstruction (i.e. a boulder or another zoombini), in which case they'll stop in the space immediately prior to the obstruction, or until they fall through an exit, in which case they'll escape the abyss.

You must write a function `escape(abyss)`, which should use the ideas of **breadth first search** to find a minimal-length sequence of moves that will allow the zoombinis to escape (there will always exist such a sequence).

You will receive as input an $n \times n$ matrix, representing an abyss, where each cell is one of:

- ' ', an empty space
- 'X', a boulder
- '*', an exit, or
- '0', '1', etc., a number representing some individual zoombini.

You are guaranteed that n is at most 20 and k, the number of zoombinis, is at most 5. Don't worry about worst case running time. Just do what you think will pass the test cases as quickly as possible!

Your output should be a list of tuples, (x, d), where x is an integer representing the zoombini being moved, and d is a cardinal directionality, in the set $\{'N','E','S','W'\}$.

To be clear, if we are at position in the matrix with indices (i, j), then north, east, south, and west correspond to $(i-1, j)$, $(i, j+1)$, $(i+1, j)$, and $(i, j-1)$, respectively.

Example 1

For example, we might get the input:

```
[[' ',' ','0','X','*'],
 [' ',' ',' ',' ',' '],
 ['X','1',' ','X',' '],
 [' ',' ',' ',' ',' '],
 [' ','2','X',' ',' ']]
```

One solution to would be the list,

```
[(2,'N'),(2,'E'),(2,'N'),(1,'E'),(1,'N'),
 (1,'E'),(1,'N'),(0,'S'),(0,'E'),(0,'N')]
```

Another would be the list

```
[(0,'S'),(0,'E'),(0,'N'),(1,'S'),(1,'E'),
 (1,'N'),(2,'W'),(2,'N'),(2,'E'),(2,'N')]
```

Example 2

With the input:

```
[['X',' ',' ',' ',' ',' ',' ',' ','1'],
 [' ',' ',' ',' ','X',' ','X',' ',' '],
 [' ',' ',' ',' ',' ',' ',' ','2','X'],
 [' ',' ',' ',' ',' ',' ',' ','X',' '],
 ['X',' ',' ',' ','*',' ',' ',' ',' '],
 [' ',' ','X',' ',' ',' ',' ',' ',' '],
 [' ','X',' ',' ',' ',' ',' ',' ',' '],
 [' ',' ',' ',' ','X','0',' ',' ',' '],
 [' ',' ',' ',' ',' ','X','*','X',' ']]
```

One solution would be the list,

```
[(0, 'N'), (0, 'W'), (0, 'S'), (1, 'W'), (1, 'S'), (2, 'N'),
 (2, 'W'), (2, 'S'), (1, 'E'), (0, 'N'), (0, 'E'), (2, 'E'), (2, 'S')]
```