1. (a) Describe an algorithm to solve the Baguenaudier puzzle.

Solution (recursion): We can use two mutually recursive algorithms.

- RingsOff(n) prints a reduced sequence of moves that changes the n-ring Baguenaudier from state 1^n to state 0^n (taking the first n rings off).
- RingsOn(n) prints a reduced sequence of moves that changes the n-ring Baguenaudier from state 0^n to state 1^n (putting the first n rings on).

Both algorithms can be applied to puzzles with more than n rings; all rings after the nth are left unchanged.

```
⟨⟨Transform $w1^n$ into $w0^n$⟩⟩
RingsOff($n$):
  if $n = 1$
      print 1
  else if $n > 1$
      RingsOff($n-2$)
      print $n$
      RingsOn($n-2$)
      RingsOff($n-1$)
```

```
⟨⟨Transform $w0^n$ into $w1^n$⟩⟩
RingsOn($n$):
  if $n = 1$
      print 1
  else if $n > 1$
      RingsOn($n-1$)
      RingsOff($n-2$)
      print $n$
      RingsOn($n-2$)
```

Alternatively, RingsOn(n) could just print the output of RingsOff(n) in reverse order.

To show that this algorithm is correct, we prove by induction that for any non-negative integer n and any bitstring w, RingsOff(n) (prints a sequence of moves that) changes $w1^n$ into $w0^n$, and RingsOn(n) changes $w0^n$ into $w1^n$.

Let n be an arbitrary non-negative integer and let w be an arbitrary bit string. Assume for any non-negative integer $k < n$ and any bitstring x that RingsOff(k) changes $x1^k$ into $x0^k$, and that RingsOn(k) changes $x0^k$ into $x1^k$. There are three cases to consider:

- If $n = 0$, both algorithms correctly do nothing.
- If $n = 1$, both algorithms correctly toggle the 1st (rightmost) bit.
- Suppose $n > 2$. First we prove RingsOff(n) correct:
  - RingsOff($n-2$) changes $w1^n$ into $w110^{n-2}$, by the inductive hypothesis.
  - Toggling the nth bit changes $w110^{n-2}$ into $w010^{n-2}$.
  - RingsOn($n-2$) changes $w010^{n-2}$ into $w01^{n-1}$, by the inductive hypothesis.
  - RingsOff($n-1$) changes $w01^{n-1}$ into $w0^n$, by the inductive hypothesis.

  A symmetric argument implies that RingsOff(n) is also correct.

■

Solution (clever[a]): In part (c), we proved that the configuration graph G_n of the n-ring puzzle is a simple path through all 2^n possible configurations, with endpoints $\mathtt{0}^n$ and $\mathtt{1}\mathtt{0}^{n-1}$. We can solve the puzzle *backwards in time* by starting at $\mathtt{0}^n$ and moving along this path to $\mathtt{1}^n$. In other words, starting with all rings off, repeatedly make the only legal move that is not the reverse of the previous move, until all rings are on. Reversing the resulting sequence of moves is straightforward. ■

[a]This is *not* a complement.

Rubric: 5 points. No time analysis is necessary because that's part (b). The clever solution only works if you actually proved in part (c) that the configuration graph is a path.

(b) *Exactly* how many moves does your algorithm perform, as a function of n? Prove your answer is correct.

Solution (via Wikipedia): Let $T(n)$ be the number of moves performed by either RingsOn(n) or RingsOff(n). This function obeys the following recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ T(n-1) + 2T(n-2) + 1 & \text{otherwise} \end{cases}$$

Wikipedia[a] claims that

$$T(n) = \begin{cases} \dfrac{2^{n+1}-2}{3} & \text{if } n \text{ is even} \\ \dfrac{2^{n+1}-1}{3} & \text{if } n \text{ is odd} \end{cases}$$

Let's prove by induction on n that Wikipedia is correct.

Let n be an arbitrary non-negative integer. Assume for all non-negative integers $k < n$ that

$$T(k) = \begin{cases} \dfrac{2^{k+1}-2}{3} & \text{if } k \text{ is even} \\ \dfrac{2^{k+1}-1}{3} & \text{if } k \text{ is odd} \end{cases}$$

There are four cases to consider:

- If $n = 0$, then $T(n) = 0 = \frac{2^{0+1}-2}{3}$. ✓
- If $n = 1$, then $T(n) = 1 = \frac{2^{1+1}-1}{3}$. ✓
- If $n \geq 2$ and n is even, then

$$\begin{aligned} T(n) &= T(n-1) + 2T(n-2) + 1 \\ &= \frac{2^n - 1}{3} + 2 \cdot \frac{2^{n-1} - 2}{3} + 1 && \text{[induction hypothesis]} \\ &= \frac{2^{n+1}-2}{3} \checkmark \end{aligned}$$

- Finally, if $n \geq 2$ and n is odd, than

$$\begin{aligned} T(n) &= T(n-1) + 2T(n-2) + 1 \\ &= \frac{2^n - 2}{3} + 2 \cdot \frac{2^{n-1} - 1}{3} + 1 && \text{[induction hypothesis]} \\ &= \frac{2^{n+1}-1}{3} \checkmark \end{aligned}$$

In all cases, Wikipedia's claimed solution is correct. ■

[a] https://en.wikipedia.org/wiki/Baguenaudier

Solution (annihilators): Let $T(n)$ be the number of moves performed by either RINGSON(n) or RINGSOFF(n). This function obeys the following recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ T(n-1) + 2T(n-2) + 1 & \text{otherwise} \end{cases}$$

We solve this recurrence using the annihilator method, described in the recurrences lecture notes. The operator $(\mathbf{E}^2 - \mathbf{E} - 2)(\mathbf{E} - 1) = (\mathbf{E} - 2)(\mathbf{E} + 1)(\mathbf{E} - 1)$ annihilates this recurrence. Thus, the function $T(n)$ has closed form $T(n) = \alpha 2^n + \beta + \gamma(-1)^n$ for some constants α, β, and γ. Small cases give us a system of three linear equations:

$$\begin{aligned} T(0) &= 0 = \alpha + \beta + \gamma \\ T(1) &= 1 = 2\alpha + \beta - \gamma \\ T(2) &= 4 = 4\alpha + \beta + \gamma \end{aligned}$$

Solving this system gives us the constants $\alpha = 2/3$, $\beta = -1/2$, and $\gamma = -1/6$. We conclude that

$$T(n) = \frac{2}{3}2^n - \frac{1}{2} - \frac{1}{6}(-1)^n = \frac{2^{n+2} - 3 - (-1)^n}{6} = \begin{cases} \dfrac{2^{n+1} - 2}{3} & \text{if } n \text{ is even} \\[2ex] \dfrac{2^{n+1} - 1}{3} & \text{if } n \text{ is odd} \end{cases}$$

just like Wikipedia says. ■

Rubric: 5 points = 2 for the closed-form solution + 3 for the proof. The closed forms

$$T(n) = \left\lfloor \frac{2^{n+1} - 1}{3} \right\rfloor = \left\lceil \frac{2^{n+2} - 2}{3} \right\rceil = \frac{2^{n+2} - 2 + (n \bmod 2)}{3}$$

are also correct. These are not the only correct proofs. I don't know whether a proof by weak induction is even possible.

(c) *[Extra credit]* Prove that for any non-negative integer n, there is *exactly one* reduced sequence of moves that solves the n-ring Baguenaudier puzzle.

Solution: Fix an arbitrary non-negative integer n. Let G_n be the graph whose vertices correspond to the 2^n configurations of the n-ring puzzle, and whose edges correspond to transitions. A reduced sequence of moves that solves the puzzle corresponds to a walk in G_n from 1^n to 0^n that never traverses the same edge twice in a row.

If $n = 0$, the unique solution is the empty sequence, and G_n is just a single isolated vertex. So for the rest of the proof, assume $n \geq 1$.

Every vertex in G_n has degree at least 1, because we can always flip the rightmost bit (that is, move the rightmost ring). On the other hand, every vertex in G_n has degree at most 2, because there are at most two legal moves from any configuration. These two facts imply our key observation:

$$\boxed{\text{Every component of } G_n \text{ is either a path or a cycle.}}$$

In fact, exactly two vertices in G_n have degree 1:

- 0^n — because the rightmost 1 doesn't exist.
- 10^{n-1} — because there is no bit to the left of the rightmost 1.

All other vertices in G_n have degree 2. Because 0^n and 10^{n-1} are the only vertices with degree 1, they must be the endpoints of the only path component of G_n; all other components of G_n are cycles.

The algorithm in part (a) implies that the starting configuration 1^n lies in the same component of G_n as the ending configuration 0^n. But we just argued that this component is a simple path! So the only reduced walk from 1^n to 0^n lies along this path.

In fact, the graph G_n is connected. Consider the following pair of mutually recursive algorithms:

```
⟨⟨Transform $w0^n$ into $w10^{n-1}$⟩⟩
OnlyLastRingOn($n$):
  if $n > 1$
      OnlyLastRingOn($n-1$)
      print $n$
      OnlyLastRingOff($n-1$)
```

```
⟨⟨Transform $w10^{n-1}$ into $w0^n$⟩⟩
OnlyLastRingOff($n$):
  if $n > 1$
      OnlyLastRingOff($n-1$)
      print $n$
      OnlyLastRingOn($n-1$)
```

These algorithms print exactly the same sequence of moves, which is also the sequence of moves for solving the Tower of Hanoi problem. Both algorithms performs a reduced sequence of exactly $2^n - 1$ moves, and therefore traverse a single path through all 2^n vertices of G_n. We conclude that G_n consists entirely of a single simple path with endpoints 0^n and 10^{n-1}. ■

Rubric: Up to 5 points.

2. (a) Suppose there are an unlimited number of Tasters. Describe an algorithm to find the poisoned bottle using at most $O(\log n)$ tests. (This is best possible in the worst case.)

Solution (binary search): Arbitrarily label the bottles from 0 to $n-1$. In the following algorithm, each Taster performs at most one test.

```
HALLOMYNAMEIS($n$):
    $lo \gets 0$
    $hi \gets n-1$
    while $lo < hi$
        $mid \gets \lfloor (lo+hi)/2 \rfloor$
        $T \gets$ new Taster
        $T$ tests bottles $lo$ through $mid$
        if $T$ is mostly dead
            $hi \gets mid$
        else
            $lo \gets mid+1$
    return $lo$
```

■

Solution (one bit at a time): Arbitrarily label the bottles from 0 to $n-1$ in binary. For any non-negative integer i, let S_i denote the subset of all bottles whose label has its ith bit equal to 1. For example, S_0 is the set of all bottles with odd labels. Exactly $\lceil \log_2 n \rceil$ of these sets are non-empty. Similarly, arbitrarily label the first $\lceil \log_2 n \rceil$ Tasters from 0 to $\lceil \log_2 n \rceil - 1$.

```
INIGOMONTOYA($n$):
    $poison \gets 0$        ⟨⟨poisoned bottle's label⟩⟩
    for $i \gets 0$ to $\lceil \log_2 n \rceil - 1$
        Taster $i$ tests subset $S_i$
        if Taster $i$ is mostly dead
            $poison \gets poison + 2^i$
    return $poison$
```

■

Rubric: 2 points. These are not the only correct solutions.

(b) Now suppose there is only one Taster. Prove that $\Omega(n)$ tests are required in the worst case to find the poisoned bottle.

Solution: Without loss of generality, assume the Taster's first name is Westley, and that Westley is lazy: If he discovers that a bottle of wine is safe, he never tests that bottle again. If Westley ever tests more than one previously untested bottle at once and becomes mostly dead, it is impossible to tell which of those previously untested bottles is poisoned. Thus, to identify the poisoned bottle, Westley must test one previously untested bottle at a time. In the worst case, the first $n-1$ tested bottles are free of poison. ■

Rubric: 1 point.

(c) Now suppose there are two Tasters. Describe an algorithm that allows them to find the poisoned bottle using only $O(\sqrt{n})$ tests.

Solution: Label each bottle with a unique pair (i, j) of integers between 1 and $\lceil\sqrt{n}\rceil$. The first Taster learns the first component of the poisoned bottle's label; the second Taster learns the second component.

```
YouKilledMyFather($n$):
    $p \leftarrow 0$
    repeat
        $p \leftarrow p + 1$
        Taster 1 tests all bottles $(p, *)$
    until Taster 1 is mostly dead
    $q \leftarrow 0$
    repeat
        $q \leftarrow q + 1$
        Taster 2 tests bottle $(p, q)$
    until Taster 2 is mostly dead
    return $(p, q)$
```

Each loop ends after at most $\lceil\sqrt{n}\rceil$ iterations, so the entire algorithm uses $O(\sqrt{n})$ tests, as required. ■

Rubric: 3 points. This is not the only solution.

(d) Finally, describe an algorithm to identify the poisoned bottle when there are k tasters. Report the number of tests that your algorithm uses as a function of both n and k.

Solution: Label each bottle with a unique sequence of k integers, each between 1 and $\lceil n^{1/k}\rceil$. Each taster learns one component of the poison bottle's label.

```
PrepareToDie($k, n$):
    for $i$ gets 1 to $k$
        $p[i] \leftarrow 0$
        repeat
            $p[i] \leftarrow p[i] + 1$
            Taster $i$ tests all bottles whose labels start with $p[1..i]$
        until Taster $i$ is mostly dead
    return $p[1..k]$
```

During each iteration of the outer loop, the inner loop ends after at most $\lceil n^{1/k}\rceil$ iterations. Thus, the entire algorithm performs at most $\boldsymbol{O(kn^{1/k})}$ ***tests***. ■

Rubric: 4 points = 3 for algorithm + 1 for time analysis. This is not the only solution.

The simpler time bound $O(n^{1/k})$ is incorrect; k is not a constant! Notice that when $k = \log_2 n$, we have $n^{1/k} = n^{1/\log_2 n} = n^{\log_n 2} = 2$, so $O(kn^{1/k}) = O(\log n)$.

3. Describe and analyze an algorithm to determine the maximum number of ducks that Mariadne can steal.

Solution (graph layering): First let's fix some notation. Let $G = (V, E)$ be the given map of the See-Bull Center. I will describe a more general algorithm that assumes k types of key cards, for some constant k, represented by consecutive integers from 1 to k. (The stated problem has $k = 6$.) Let $[k] = \{1, 2, \ldots, k\}$ denote the set of all key card types. My solution assumes each vertex record v in the input graph has two extra fields:

- v.*duckies* is the number of rubber ducks in room v.
- v.*cards* is a subset of $[k]$ describing the key cards in room v.

Similarly, each edge record e has two extra fields:

- e.*locked* is a boolean indicating whether door w is locked.
- $e.\mathit{key} \in [k]$ indicates which type of key card unlocks door e (if it's locked).

To begin the algorithm, we construct a new *directed* graph $G' = (V', E')$ as follows:

- $V' = V \times 2^{[k]}$. (For any set X, the notation 2^X denotes the power set of X.) Each vertex (v, K) indicates that Mariadne is located at room v and holds exactly the key cards in the subset $K \subseteq [k]$.
- E' contains every directed edge $(v, K) \rightarrow (w, L)$ satisfying the following conditions:
  - $vw \in E$ — There's a door between v and w.
  - $vw.\mathit{locked} = \textsc{False}$ or $vw.\mathit{key} \in K$ — Mariadne can open that door.
  - $L = K \cup v.\mathit{cards}$ — Mariadne adds any new key cards to her collection.

Altogether G' contains at most $2^k V = O(V)$ vertices and $2^{k+1} E = O(E)$ directed edges. (Remember: k is a constant!) We can construct G' by brute force in $O(V + E)$ time.

After we construct G', we simulate Mariadne making two passes through the building. During the first pass, she collects as many different key cards as possible and then returns to the entrance. During the second pass, she collects as many ducks as possible, using the key cards collected in the first pass. In more detail:

- In the first phase, we mark every vertex (v, K) of G' that is reachable from the start vertex $(s, \varnothing)$ using whatever-first search. Let $\overline{K}$ be the largest set such that $(s, \overline{K})$ is marked, or equivalently, the union of all subsets K such that (v, K) is marked for at least one vertex $v \in V$.
- In the second phase, we unmark every vertex of G', mark every vertex $(v, \overline{K})$ that is reachable from $(s, \overline{K})$ using whatever-first search, and then return the sum of v.*duckies* over all marked vertices $(v, \overline{K})$.

  (Alternatively, we modify G by removing all edges e such that $e.\mathit{lock} \neq \varnothing$ and $e.\mathit{lock} \notin \overline{K}$, mark every vertex of G that is reachable from the start vertex s, and return the sum of v.*duckies* over all marked vertices v.)

Each of these phases, and therefore the entire algorithm, runs in ***$O(V + E)$ time***. ■

Solution (keep it simple): I'll use the same input assumptions as the previous solution. Mariadne's algorithm consists of $k+1$ phases, each of which performs a whatever-first search of a subgraph of G, starting at s.

For each index $1 \le i \le k$, let $K_{i-1} \subseteq [k]$ denote the subset of keys that Mariadne has collected during the first i phases; in particular, $K_0 = \varnothing$. Let $G_i = (V, E_i)$ be the subgraph of G defined by setting

$$E_i = \left\{ e \in E \;\middle|\; e.locked = \textsc{False} \;\text{ or }\; e.key \in K_{i-1} \right\}.$$

Finally, in the ith phase, Mariadne marks every vertex that is reachable from s in G_i using a whatever-first search, and then defines K_i to be the union of $v.cards$ over all marked vertices v. (Alternatively, every time Mariadne finds a new key card, she could abort the current search and start the next phase from wherever she happens to be.) Finally, we return the sum of $v.duckies$ over all vertices v that are marked at the end of phase $k+1$.

There are only k types of key cards, so there are at most k initial phases in which Mariadne finds new key cards. In other words, for some index $i \le k$, we must have $K_i = K_j$ for all $j > i$. In particular, after the kth phase, Mariadne as found all the key cards that she is ever going to find. Thus, during phase $k+1$, Mariadne reaches all the rubber ducks she is ever going to reach!

The algorithm consists of $k+1$ traversals of subgraphs of G, so the overall running time is $O(k(V+E)) = O(V+E)$. As a function of k, this is exponentially faster than the previous solution, but k is a constant, so the improvement is also only a constant.

In fact, this algorithm does not require Mariadne to know *anything* about the distributions of rubber ducks, or the distribution of key cards, or which card opens which doors, or even how many types of key cards there are. All she has to do is systematically explore the building, taking every duck and every key card she finds, and trying every key card in every door, until she has performed a complete traversal of the building without finding anything new. The running time of this blind search is still $O(k(V+E))$, where k is the number of key-card types. ■

Solution (modify WFS): I'll use the same input conventions as the previous solution. I'll use a modification of whatever-first search that maintains $k+1$ bags of vertices instead of a single bag. (Here, as in Jeff's textbook, a "bag" is any data structure that stores a set of items and supports two operations, each in $O(1)$ time: Insert one item (given by the user), and Remove one item (chosen by the data structure). Stacks and queue are both bags.)

```
BohBohDohDeeYo($V, E, s$):
  ⟨⟨Intialize everything⟩⟩
  $mycards \gets \varnothing$        ⟨⟨Which cards have we found?⟩⟩
  $myducks \gets 0$        ⟨⟨How many ducks have we found?⟩⟩
  unmark all vertices
  for $i \gets 0$ to $k$
      $Bag[i] \gets$ empty bag
  Insert($Bag[0], s$)

  repeat forever:
      ⟨⟨Remove a vertex from any non-empty accessible bag⟩⟩
      $v \gets$ None
      if $Bag[0]$ is not empty
          $v \gets$ Remove($Bag[0]$)
      for all $k \in mycards$
          if $v =$ None and $Bag[k]$ is not empty
              $v \gets$ Remove($Bag[k]$)

      ⟨⟨If all accessible bags are empty, we're done⟩⟩
      if $v =$ None
          return $myducks$

      ⟨⟨Really explore the space (if we haven't already)⟩⟩
      if $v$ is unmarked
          mark $v$
          $myducks \gets myducks + v.ducks$
          $mycards \gets mycards \cup v.cards$
          for all neighbors $w$ of $v$
              if $vw$ is unlocked        $(\star)$
                  Insert($Bag[0], w$)
              else
                  Insert($Bag[vw.key], w$)
```

Even in the worst case, this algorithm marks each vertex at most once, and therefore considers each edge (in line $(\star)$) at most once. Thus, the total number of Insertions into the bags is at most $E+1$. It follows that the main loop halts after at most $E+1$ iterations. It also follows that the total time spent inside the final for-loop, over the entire execution of the algorithm, is $O(E)$.

In each iteration of the main loop (except the last), we remove one vertex from some non-empty bag for which Mariadne has a key card. The stated implementation requires $O(k)$ time in the worst case to find an empty bag. Except for the final for loop, which we've already accounted for, the rest of the iteration of the main loop requires only $O(1)$ time. We conclude that the overall algorithm runs in ***$O(kE) = O(E)$ time*** (because k is a constant!)

With more effort, we can reduce the time to remove one vertex from an accessible bag (the lines in blue) from $O(k)$ to $O(1)$. We maintain the indices of non-empty accessible bags in a master dictionary (really just another bag).

- Whenever we INSERT a node into any bag, if the set *mycards* contains the index of that bag, we also INSERT that index into the master dictionary.
- Whenever we REMOVE a node from any bag, if that bag is now empty, we REMOVE its index from the master dictionary.
- Whenever we find a new key card, if the corresponding bag is non-empty, we INSERT its index into the master dictionary.
- Finally, to remove one vertex from an accessible bag, we REMOVE a bag index i from the master dictionary, then REMOVE one vertex from $Bag[i]$, and finally, if $Bag[i]$ is not empty, reINSERT i into the master dictionary.

With these modifications, the algorithm runs in O(E) time *even if k is not constant*. ■

Rubric: 10 points = 3 for building the correct graph + 3 for solving the correct problem on that graph + 2 for using the right algorithm for that problem + 2 for time analysis. These are not the only correct solutions.