CSE 102
Homework Assignment 7
Solutions

1. Let m be an integer in the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and consider the following problem: determine m by asking 3-way questions, i.e. questions with at most 3 possible answers. For instance, one could ask which of 3 specific subsets m belongs to.

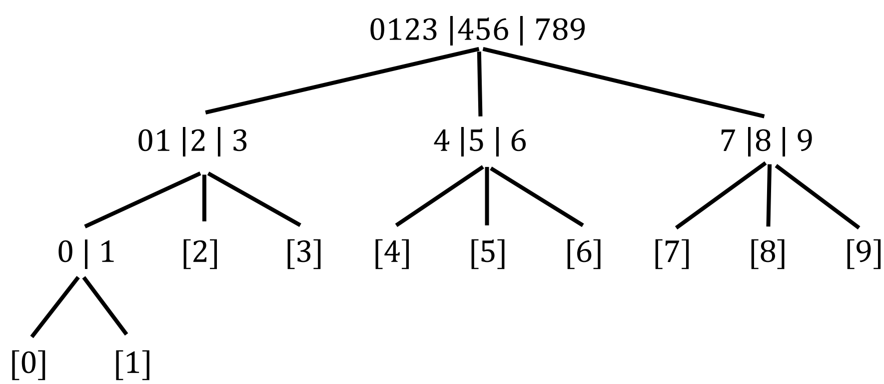
   a. Give a decision tree argument showing that at least 3 such questions are necessary in worst case. In other words, prove that no correct algorithm can solve this problem by asking only 2 questions in worst case.

   Proof:
   Any algorithm that solves this problem can be represented by ternary decision tree, since each question can have at most 3 answers. Since there are 10 possible verdicts, the height of such a tree must satisfy $h \geq \lceil \log_3(10) \rceil = 3$. Therefore no such algorithm can ask less than 3 questions in worst case. ■

   b. Design an algorithm that will solve this problem by asking 3 such questions in worst case. Express your algorithm as a decision tree.

   Solution:
   Each internal node below represents a question asking whether m belongs to one of 3 possible subsets of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. For instance $0123 \,|456\,|\,789$ represents the question: "does m belong to $\{0, 1, 2, 3\}$, to $\{4, 5, 6\}$, or to $\{7, 8, 9\}$?" Verdicts are placed in brackets "[ ]".

   <u>Decision tree:</u>



2. **Bar Weighing Problem**
   Assume we are given 12 gold bars numbered 1 to 12 where 11 bars are pure gold and one is counterfeit: either gold-plated lead (which is heavier than gold), or gold-plated tin (lighter than gold). The problem is to find the counterfeit bar and what metal it is made of using only a balance scale.

Any number of bars can be placed on each side of the scale, and each use of the scale produces one of three outcomes: either the left side is heavier, or the two sides are the same weight, or the right side is heavier.

a. Give a decision tree lower bound for the (worst case) number of weighing's that must be performed by any algorithm solving this problem.

Theorem
Any algorithm that solves this problem must perform at least 3 weighing's in worst case.

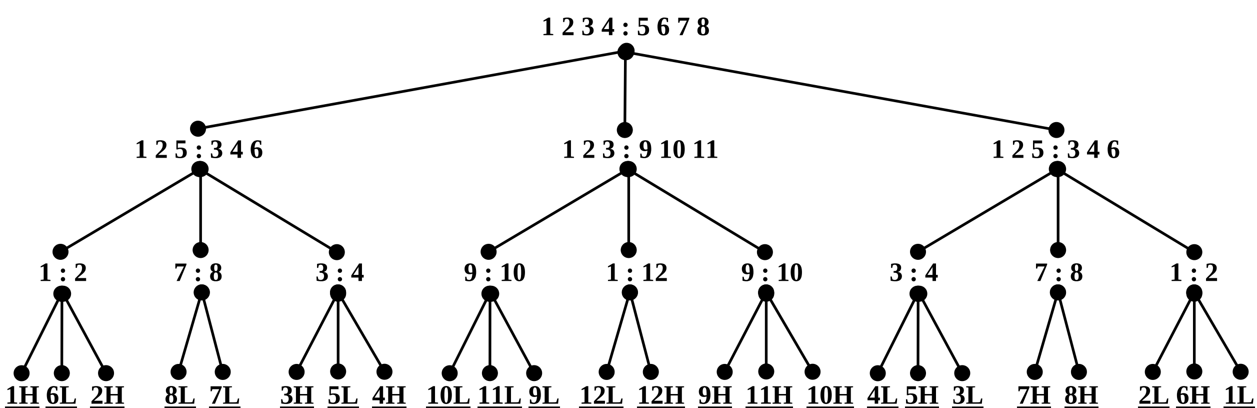
Proof: Since there are 3 possible outcomes to each weighing, the decision tree for this problem is a ternary tree. Observe that there are 24 distinct verdicts, which may be represented as: $\{1L, 2L, 3L, \dots, 12L, 1H, 2H, 3H, \dots, 12H\}$. For instance, $8L$ is the conclusion that bar 8 is light (i.e. tin) and all other bars are genuine, while $11H$ represents the conclusion that bar 11 is heavy (lead) and all others are pure gold.

The height h of a decision tree representing any algorithm for this problem must therefore satisfy $h \geq \lceil \log_3(24) \rceil = 3$. Therefore at least 3 weighing's must be performed (in worst case) to identify the counterfeit bar. ■

b. Design an algorithm that solves this problem with (worst case) number of weighing's equal to the lower bound you found in (a). Present your algorithm by drawing a decision tree, rather than pseudo-code.

Solution:
The following decision tree represents one algorithm (among many) that identifies the counterfeit bar in exactly 3 weighing's.

Each internal node represents a particular set of bars on the left and right side of the balance. For instance, a node labled `abc:def` would mean to place bars `a`, `b` and `c` on the left side of the balance, and place bars `d`, `e` and `f` on the right. The left child is taken if the balance tilts left, indicating that the combination `abc` is heavier than `def`. Similarly the right child is taken if the balance tilts right, and the middle child is taken if the balance does not move, which implies the bar combinations `abc` and `def` are equal in weight.



c. Alter the problem slightly to allow the possibility that all 12 bars are pure gold. Thus, there is one additional possible verdict: "all gold". Make a minor change to your algorithm in part (b) so that it gives a correct answer to this more general problem.

Solution:
Observe that if all bars are pure gold, then all weighing's (of equal numbers of bars) will balance. Therefore, by adding a single leaf to the above tree labeled AllGold as the middle child of the 1:12 node, we obtain an algorithm that solves this problem in the more general setting.

3. **Water Jug Problem** (Problem 8-4: page 206 of CLRS 3rd edition)
Suppose that you are given *n* red and *n* blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.

It is your task to find a grouping of the jugs into pairs of red and blue jugs that hold the same volume of water. To do so, you may perform the following operation: pick a pair of jugs, one red, one blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you if the two jugs hold the same amount of water, and if not, which one holds more water. Assume that such a comparison takes one unit of time. Your goal is to find an algorithm that solves this problem. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe an algorithm that uses $\Theta(n^2)$ comparisons (in worst case) to group the jugs into pairs.

Solution:
Call the red jugs $R_1, R_2, \dots, R_n$ and the blue jugs $B_1, B_2, \dots, B_n$. Begin by comparing R_1 to each of the *n* blue jugs until a match is found. At most $n-1$ comparisons are necessary to determine a match, since if no match has been found during $n-1$ comparisons, the last blue jug must match, and no additional comparison is necessary. Next compare R_2 to the $n-1$ unmatched blue jugs. This time at most $n-2$ comparisons are necessary to determine a match. Next compare R_3 to at most $n-3$ blue jugs. In general, one compares R_i to at most $n-i$ blue jugs. The last red jug R_n doesn't need to be compared to any blue jugs since there is only one possibility left at that point. Once this process is complete, a matching of red and blue jugs has been attained. The total number of comparisons performed is at most

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

■

b. Prove a lower bound of $\lceil \log_3(n!) \rceil$ for the worst case, and $\log_3(n!)$ for the average case number of comparisons to be performed by any algorithm that solves this problem.

Proof:
Each comparison described above has 3 possible outcomes, and hence any algorithm solving this problem is represented by a ternary tree. There is a total of $n!$ ways to match the n red jugs to the n blue jugs, so this problem has $n!$ possible verdicts. Any algorithm solving this problem can be represented by a decision tree whose height satisfies $h \geq \lceil \log_3(n!) \rceil$ and whose average height satisfies $a \geq \log_3(n!)$. These quantities therefore serve as lower bounds on the worst case and average case number of comparisons, respectively. ■

4. Show that at least $\binom{n}{2}$ "adjacency" questions are necessary to determine whether a graph G on n vertices is acyclic. (Hint: use the following adversary strategy. Answer yes to any edge probe, unless that answer would prove the existence of a cycle.)

Proof: First observe that the result is trivial when $n = 2$ since in that case, there is only one potential edge to probe. We may therefore assume without loss of generality that $n \geq 3$. Consider any algorithm for this problem, and run it against the following adversary simulating an input graph G with n vertices. The daemon's strategy is to answer yes to any edge probe, unless that answer would prove the existence of a cycle. More precisely, the daemon maintains two edge sets A and B, where initially B is empty and A contains all $\binom{n}{2}$ edges in K_n. He then performs the following algorithm whenever the algorithm probes an edge e.

```
Probe(e)
1.  if B + e is acyclic
2.      B = B + e
3.      return Yes
4.  else
5.      A = A − e
6.      return No
```

Observe that at all times we have $B \subseteq A$, and the set $A - B$ consists of exactly those edges in K_n not yet been probed. Also note, both A and B are consistent with the daemon's entire sequence of answers, since when the answer Yes is given, the edge is added to B and not removed from A, while if No is given, the edge is removed from A and not added to B. The following invariants are maintained over any sequence of edge probes:

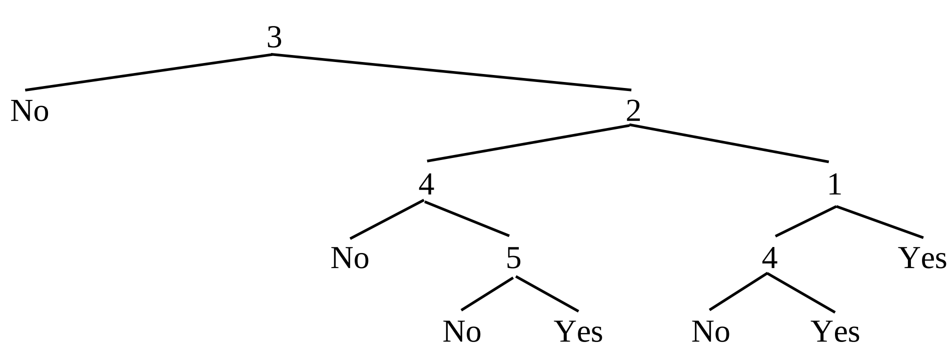
(a) The subgraph B is acyclic. This is obvious from the daemon's strategy.
(b) If B is disconnected, then A contains all edges of K_n that join distinct connected components of B. **Proof:** Let e be an edge joining two components of B. Thus, if e were added to B, it would not create a cycle in B. Such an edge could not have been removed from A, hence $e \in A$.
(c) The subgraph A is connected. **Proof:** Assume, to get a contradiction, that at some point an edge e is probed and removed from A, causing it to become disconnected. At the time of its removal, e does not form a cycle with the edges of A (since otherwise it's removal would not disconnect A.) Since $B \subseteq A$, e does not form a cycle with the edges of B either. But then e should have been added to B rather than removed from A. This contradiction shows that no such edge e exists, and therefore A is connected.
(d) If $A \neq B$, then A contains a cycle. **Proof:** Assume, to get a contradiction, that A is acyclic. Then being connected, A is a tree. Since $B \subseteq A$ and $A \neq B$, we see B must be disconnected. (Another property of trees: removing an edge from a tree results in a disconnected graph.) Let $e \in A - B$. Then e joins two components of B. Since A is acyclic, it can contain no other edges joining the same two components. But since $n \geq 3$, at least one other edge e' joining those two components exists in K_n. By (b) above, e' also belongs to A, contradicting our earlier observation. This contradiction shows that A must contain a cycle.

Now suppose the algorithm halts and returns a verdict (acyclic/not acyclic) after probing fewer than $\binom{n}{2}$ edges. Then some edge of K_n was not probed, hence $A \neq B$. Remark (d) implies that A is not acyclic, and by (a) B is acyclic. Since the algorithm cannot distinguish between A and B, it cannot be correct. If the algorithm says G contains a cycle, then the daemon can claim $G = B$, while if the algorithm says G is acyclic, the daemon claims $G = A$. Thus, any correct algorithm solving this problem must probe all $\binom{n}{2}$ edges of K_n. ■

5. Let $b = x_1x_2x_3x_4x_5$ be a bit string of length 5, i.e. $x_i \in \{0, 1\}$ for $1 \leq i \leq 5$. Consider the following problem. Determine whether or not b contains the substring 111. Restrict attention to those algorithms whose only operation is to peek at a bit. Obviously 5 peeks are sufficient. A decision tree argument provides the (useless) fact that at least one peek is necessary.

   a. Design an algorithm that solves the problem using only 4 peeks in worst case. Express your algorithm as a decision tree.

   Solution:
   The following decision tree is just one of many correct solutions. Each internal node represents the index of the bit being peeked, while the leaves indicate verdicts. A branch to the left signifies that the peek saw a 0, while a branch to the right indicates the peek detected a 1.



b. Use an adversary argument to show that 4 peeks are necessary in general.

Proof:
Consider any algorithm purporting to solve this problem, and start it on an unspecified bit string b of length 5. The daemon's strategy is as follows. The first time one of the bits $\{b_2, b_4\}$ are peeked, answer 0, the second time a bit in this set is peeked, answer 1; any time one of the bits $\{b_1, b_3, b_5\}$ is peeked, answer 1. To keep track, the daemon maintains two bit strings *x* and *y*, where initially $x = 11111$ and $y = 00000$. Each time the answer 0 is given, the corresponding bit in *x* is flipped to 0, and if the answer 1 is given, the corresponding bit in *y* is flipped to 1. We may assume the algorithm never peeks at the same bit twice (otherwise the daemon just repeats his previous answer.) Expressing the daemon's strategy in pseudo-code we have:

```
Peek(i)
if i ∈ {1, 3, 5}
    y_i = 1 and answer 1
else if i == 2
    if x_4 == 0          //  b_4 was already peeked and answered 0
        y_2 = 1 and answer 1
    else                 //  this is our first peek at {b_2, b_4}
        x_2 = 0 and answer 0
else  // i == 4
    if x_2 == 0          //  b_2 was already peeked and answered 0
        y_4 = 1 and answer 1
    else                 //  this is our first peek at {b_2, b_4}
        x_4 = 0 and answer 0
```

Note that after any set of bit peeks, both strings *x* and *y* are consistent with the daemon's sequence of answers: if 0 was returned for the i^{th} bit, then x_i was flipped to 0 and y_i remained 0, while if 1 was returned for the i^{th} bit, then y_i was flipped to 1 and x_i remained 1. Notice also that any sequence of bit peeks results in at most one 0 being returned, and hence *x* can only be one of the strings 11111, 10111 or 11101, all of which contain 111.

Now suppose the algorithm halts and returns an answer (yes/no) after performing at most 3 peeks. In this case, *y* cannot contain the substring 111. (The only way *y* can contain three 1's is if bits 2 and 4 were never peeked. But then *y* is 10101, which does not contain 111.) We conclude that the algorithm cannot be correct, since after only three peeks, there will necessarily exist a bit string consistent with the daemon's sequence of answers, but which contradicts the algorithm's verdict. Therefore, any correct algorithm for this problem must perform at least 4 bit peeks in worst case. ■