Solutions for Homework Assignment #4

Answer to Question 1.

Let $J = \{i_1, i_2, \dots, i_k\}$ be the set of k indices such that $A[i_1] = A[i_2] = \dots = A[i_k] = x$ (if $k = 0$ then $J = \emptyset$).

a. Let i be the index picked by the algorithm in the first iteration of the algorithm's loop. Since i was picked uniformly at random from the set of n indices $I = \{1, 2, \dots, n\}$, for each $j \in I$, $\mathbf{Pr}[i = j] = \frac{1}{n}$.
The algorithm returns TRUE in the first iteration of the repeat loop if and only if $i \in J$.
If $k > 0$ then the probability that this occurs is:

$$\begin{aligned}
\mathbf{Pr}[i \in J] &= \mathbf{Pr}[(i = i_1) \vee (i = i_2) \vee \ldots \vee (i = i_k)] \\
&= \mathbf{Pr}[i = i_1] + \mathbf{Pr}[i = i_2] + \cdots + \mathbf{Pr}[i = i_k] \\
&= \frac{1}{n} + \frac{1}{n} + \ldots + \frac{1}{n} \\
&= \frac{k}{n}
\end{aligned}$$

Note that the above also holds when $k = 0$, i.e., when $J = \emptyset$.

b. The probability that this algorithm returns TRUE is $1 - \left(\frac{n-k}{n}\right)^r$. To see why, note that:
The algorithm returns FALSE iff it *fails* to find x in every one of the r iterations of the **repeat** loop.
Let j_ℓ be the index picked by the algorithm in iteration ℓ of the **repeat** loop, where $1 \le \ell \le r$.
The algorithm returns FALSE iff for every ℓ, $1 \le \ell \le r$, $j_l \notin J$.
From Part **a.**, it is clear that for every ℓ, $1 \le \ell \le r$: $\mathbf{Pr}[j_\ell \notin J] = 1 - \frac{k}{n} = \frac{n-k}{n}$. Therefore:

$$\begin{aligned}
\mathbf{Pr}[\text{the algorithm returns FALSE}] &= \mathbf{Pr}[(j_1 \notin J) \wedge (j_2 \notin J) \wedge \ldots \wedge (j_r \notin J)] \\
&= \mathbf{Pr}[j_1 \notin J] \cdot \mathbf{Pr}[j_2 \notin J] \cdot \cdots \cdot \mathbf{Pr}[j_r \notin J] \\
&= \frac{n-k}{n} \cdot \frac{n-k}{n} \cdot \ldots \cdot \frac{n-k}{n} \\
&= \left(\frac{n-k}{n}\right)^r
\end{aligned}$$

Thus, $\mathbf{Pr}[\text{the algorithm returns TRUE}] = 1 - \left(\frac{n-k}{n}\right)^r$.

c. We can think of each iteration of the **repeat** loop as a *Bernoulli trial* (see Section C.4, page 1201, of our CLRS textbook), which is an experiment with two possible outcomes:

1. *success*, which occurs with probability p, and
2. *failure*, which occurs with probability $1 - p$.

In our case each iteration of the **repeat** loop has two possible outcomes:

1. *success*, i.e., x is found in this iteration, which occurs with probability $p = \frac{k}{n}$, and
2. *failure*, i.e., x is not found in this iteration, which occurs with probability $1 - p = 1 - \frac{k}{n}$.

So the (possibly infinite) sequence **repeat** loop iterations is a sequence of *mutually independent* Bernoulli trials, each with the *same* probability $p = \frac{k}{n}$ for success.

Suppose $k = 0$ (this means that x does not appear at all in the array A). In this case, for each iteration of the **repeat** loop, the probability of success (i.e., of finding x) is $p = 0$; and so the loop is executed ∞ many times.

Now let $0 < k \leq n$. Let the random variable X be the number of trials till the first success occurs. In our case, X represents the number of **repeat** loop iterations that the algorithm executes (because it exits this loop in the first iteration where it finds x). So the ***expected*** number of iterations that the algorithm executes is $\mathbf{E}[X]$.

Note that X has values in the range $\{1, 2, \ldots, \}$, and we have for every $\ell \geq 1$: $\mathbf{Pr}[X = \ell] = (1-p)^{\ell-1}p$ (because this is the probability of $\ell - 1$ consecutive failures followed by a success). This probability distribution is called a *geometric distribution*.

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{\ell=1}^{\infty} \ell \cdot \mathbf{Pr}[X = \ell] \\
&= \sum_{\ell=1}^{\infty} \ell(1-p)^{\ell-1}p \\
&= \frac{1}{p} \qquad \text{(see identity C.32 page 1203 of CLRS)} \\
&= \frac{n}{k} \qquad \text{(because } p = \tfrac{k}{n}\text{)}
\end{aligned}$$

So the ***expected*** number of iterations that the algorithm executes is $\mathbf{E}[X] = \frac{n}{k}$.

Answer to Question 2.

Let L be list a of constraints, and let $L_=$ be the sublist of constraints in L that are equality constraints. For example, if L is

$$x_5 = x_1, x_1 = x_6, x_4 = x_5, x_1 = x_3, x_2 \neq x_6, x_2 = x_7, x_3 \neq x_4$$

then $L_=$ is

$$x_5 = x_1, x_1 = x_6, x_4 = x_5, x_1 = x_3, x_2 = x_7$$

Note that $L_=$ represents a collection $\mathcal{C}_=$ of sets of variables, where each set contains the variables that are *equal* to each other (by transitivity). Here, $\mathcal{C}_=$ consists of the two sets $\{x_1, x_3, x_4, x_5, x_6\}$ and $\{x_2, x_7\}$.

Note that:

1. For every set in the collection $\mathcal{C}_=$, all the variables in that set should be assigned the *same* integer.
2. If L contains some inequality $x_i \neq x_j$, then x_i and x_j should be assigned *different* integers, but this assignment is possible if and only if x_i and x_j are in different sets of $\mathcal{C}_=$.

From the above, the high-level idea of the algorithm is as follows:

1. Use the *equality* constraints in L to build the sets of variables that are *equal to each other*.
2. For each *inequality* constraint $x_i \neq x_j$ in L, determine whether x_i and x_j are in the *same* set; if they are, then output NIL and stop.
3. For each set, assign the same integer to all the variables in this set; different sets must get different integers. Output this assignment.

One way to implement this algorithm efficiently is to use the disjoint-sets UNION-FIND data structure because with this data structure it is easy to: (1) construct the collection of disjoint sets $\mathcal{C}_=$, and (2) check whether the variables of each inequality constraint $x_i \neq x_j$ is in the same set of $\mathcal{C}_=$ or not. More precisely:

1. Start from n singleton sets representing each of the n variables $x_1, x_2, \ldots, x_n$.
2. Scan the list L. For each equality constraint $x_i = x_j$ in L, if x_i and x_j are in different sets then we *merge* these two sets using a UNION operation.
3. Re-scan the list L. For each inequality constraint $x_i \neq x_j$ in L, use two FIND operations to determine whether x_i and x_j are in the *same* set; if they are, then output NIL and stop.
4. For each of the disjoint sets obtained by the end of step (2), assign the same integer to all the variables in this set (ensure that each set has a different number). This can be done in the following way:

   For each variable x_i, $1 \leq i \leq n$, use the FIND operation to find its set representative x_j, and assign the integer j to x_i. Note that we do n additional FIND operations in this step.

The pseudocode for the algorithm outlined above is as follows (where i represents variable x_i, for $1 \leq i \leq n$):

```
SatisfyingAssignment(n, ConstraintsList)
for i = 1 to n
    MakeSet(i)              // variable x_i is represented by i
for each equality constraint x_i = x_j in ConstraintsList
    u = Find(i)
    v = Find(j)
    if u ≠ v
        Union(u, v)
for each inequality constraint x_i ≠ x_j in ConstraintsList
    if Find(i) = Find(j)
        return NIL
A[1..n] = []               // A is the output array. A[i] will store the integer assigned to x_i.
for i = 1 to n
    u = Find(i)             // the representative of the set that contains i is the integer u
    A[i] = u
return A
```

Complexity Analysis. For efficiency it is best to use the *forest implementation* of the disjoint-sets data structure with the weighted union (WU) and path compression (PC) heuristics. Recall that $m \geq n$. Note that the above algorithm performs $O(n)$ MAKESETS, $O(n)$ UNIONS, and $O(m)$ FINDS. Thus, the worst-case time complexity of this algorithm is $O(m \log^* n)$.