CSC263, Assignment #4

Brendan Neal[1], Filip Tomin[2]

[1]Section L0201 Student number: 1001160236

[2]Section L0201 Student number: 1001329984

1. The only way that $x = i$ can be true is that the algorithm changes x to i with probability p_i on iteration i, and then for every iteration after i, the algorithm does not change x. Since x is changed on the i-th iteration, we are not concerned with x's previous value or any of the previous probabilities for that matter. Therefore, the probability of x staying equal to i is equal to $(p_i) * (1 - p_{i+1}) * (1 - p_{i+2}) * \cdots * (1 - p_n)$. Thus, $\Pr[x = i] = (p_i) * (1 - p_{i+1}) * (1 - p_{i+2}) * \cdots * (1 - p_n)$

   (a) Since this question specifies that $\forall i \in \{2, 3, \ldots, n\}, p_i = \frac{1}{2}$, it is true that $\forall i \in \{2, 3, \ldots, n\}, 1 - p_i = \frac{1}{2}$. Thus, $\Pr[x = i] = (p_i) * (1 - p_{i+1}) * (1 - p_{i+2}) * \cdots * (1 - p_n) = \Pr[x = i] = (\frac{1}{2}) * (\frac{1}{2}) * (\frac{1}{2}) * \cdots * (\frac{1}{2})$. From the i^{th} index to the n^{th} index there are $n - i$ elements, plus one more for the actual i^{th} element. There is one edge case however, $i = 1$ - since x is initialized as 1, we can ignore the $+1$ in $n - i + 1$ because the chance of $x = 1$ is initially 1, which means that the first probability in the expansion for $\Pr[x = 1]$ is 1. Thus,

   $$\Pr[x = i] = \begin{cases} (\frac{1}{2})^{n-i} \text{ if } i = 1 \\ (\frac{1}{2})^{n-i+1} \text{ if } i \geq 2 \end{cases}$$

   (b) This question specifies that $\forall i, 1 \leq i \leq n, \Pr[x = i] = \frac{1}{n}$. Since we specified in introduction to this question that $\Pr[x = i] = (p_i) * (1 - p_{i+1}) * \cdots * (1 - p_n)$, it must be true that $\frac{1}{n} = (p_i) * (1 - p_{i+1}) * (1 - p_{i+2}) * \cdots * (1 - p_n)$. We postulate that $\forall i, 1 \leq i \leq n$ $(p_i = \frac{1}{i} \iff \Pr[x = i] = \frac{1}{n})$, and we will prove this using strong induction below:

   Predicates and Variables:
   Define $P(k)$ to be true when the logical statement
   $(\forall i, (n - k + 1) \leq i \leq n, p_i = \frac{1}{i} \implies \Pr[x = n - k + 1] = \frac{1}{n})$ evaluates to true for an input k.

   Base Case:
   Consider the case where $k = 1$.

Since x's value cannot change after the n-th iteration of the algorithm,
$\Pr[x = n] = \frac{1}{n}$.
By definition of $\Pr[x = n]$, $\Pr[x = n] = p_n$.
Thus, $\Pr[x = n] = p_n = \frac{1}{n}$
Since $(\forall i, n \leq i \leq n,\ p_i = \frac{1}{i} \implies \Pr[x = n] = \frac{1}{n})$, P(1) must hold.

Induction Hypothesis:
Assume that for $\forall j \in \mathbb{N}, 1 < j < n$, $P(j)$ holds.

Induction Step:
Consider the case where k = n.
By the definition of $\Pr[x = n - k + 1] = \Pr[x = n - n + 1] = \Pr[x = 1] = (1 - p_2) * (1 - p_3) * \cdots * (1 - p_n)$.
As specified by the question:
$\Pr[x = 1] = \frac{1}{n} = (1 - p_2) * (1 - p_3) * \cdots * (1 - p_{n-1}) * (1 - p_n)$.
By the induction hypothesis:
$\Pr[x = 1] = (1 - p_2) * (1 - p_3) * \cdots * (1 - p_{n-1}) * (1 - p_n)$
$= (1 - \frac{1}{2}) * (1 - \frac{1}{3}) * \cdots * (1 - \frac{1}{n-1}) * (1 - \frac{1}{n})$
$= (\frac{2-1}{2}) * (\frac{3-1}{3}) * \cdots * (\frac{n-1-1}{n-1}) * (\frac{n-1}{n})$
$= (\frac{1}{2}) * (\frac{2}{3}) * \cdots * (\frac{n-2}{n-1}) * (\frac{n-1}{n})$

Notice that product of the numerators is $(n-1)!$ and the product of the denominators is $n!$
Thus, $\Pr[x = 1] = \frac{(n-1)!}{n!}$
Dividing the factorials, we get: $\Pr[x = 1] = \frac{(n-1)!}{n(n-1)!} = \frac{1}{n}$.
Since $(\forall i, 1 \leq i \leq n,\ p_i = \frac{1}{i} \implies \Pr[x = 1] = \frac{1}{n})$, $P(n)$ must hold.

Conclusion
Since it has been shown that $(P(1) \wedge \forall j \in \mathbb{N}, 1 < j < n, P(j)) \implies P(n)$, it is true by strong induction that $\forall i, 1 \leq i \leq n\ (p_i = \frac{1}{i} \iff \Pr[x = i] = \frac{1}{n})$

2. (a) We will have two variables v_1 and v_2. We will then call FlipBiasedCoin twice, once for v_1 and once for v_2. We will set v_1 to be equal to the value obtained from the first call of FlipBiasedCoin and v_2 to be equal to the second. If v_1 and v_2 are equal, we will repeat this process again. If v_2 and v_2 are not equal, we stop and return the value of v_1.

This algorithm outputs 1 or 0 with equal probability $(\frac{1}{2})$ by limiting the sample space of possible outcomes. Without limitation, the sample space of (v_1, v_2) is equal to $\{(1,1),(0,0),(1,0),(0,1)\}$. Since the probability of FlipBiasedCoin returning 1 is p and the probability of it returning 0 is $1-p$, the respective probabilities of the sample space are: $\{(p*p),((1-p)*(1-p)),(p*(1-p)),((1-p)*p)\} =$

$\{p^2, (1-p)^2, p-p^2, p-p^2\}$. Since the algorithm flips the biased coins until the results are different, it limits the sample space to $\{(1,0),(0,1)\}$.

Since the algorithm is guaranteed to stop when $(v_1, v_2) \in \{(1,0),(0,1)\}$, the total of the probabilities of the events in this sample space must be equal to 1. Since the probabilities of $(1,0)$ or $(0,1)$ occurring are equal and there are only two different outcomes in the sample space, the probability that the algorithm produces either $(0,1)$ or $(1,0)$ is equal to $\frac{1}{2}$.

According to the algorithm's rules (that it returns the value of v_1 when it stops), the probability that it outputs 0 is equal to the probability that $(v_1, v_2) = (0,1)$ and the probability that the algorithm outputs 1 is equal to $(v_1, v_2) = (1,0)$. Since we know that both these probabilities have value $\frac{1}{2}$, we can say that the algorithm outputs 1 with probability $\frac{1}{2}$ and 0 with probability $\frac{1}{2}$.

(b) We will provide some pseudo-code for this algorithm to make its run-time easier to analyze:

```
UnbiasedRNG():
    v1 = 0
    v2 = 0
    while (v1 == v2):
        v1 = FlipBiasedCoin()
        v2 = FlipBiasedCoin()
    return v1
```

Notice that:
The algorithm needs to execute two constant-time steps to set itself up (2 assignment statements)
Each iteration of the while loop executes 3 constant-time steps (1 while loop check, 2 coin flips)
A failure in the while-loop condition executes 2 constant-time steps (1 while loop check, 1 return statement)

Let E be the expected number of steps that the algorithm executes.
Let B be the number of steps the algorithm executes on its setup.
Let F be the number of steps the algorithm executes when the while-loop condition is true.
Let S be the number of steps the algorithm executes when the while-loop condition is false.
Let x be the probability that the algorithm gets $(1,1)$ or $(0,0)$.

The number of steps that the algorithm executes for i flips consisting of $i-1$ flips

that return doubles and one flip that returns different values is $(B+F*(i-1)+S)$. The probability that the algorithm executes this order of i flips is equal to $(x^{i-1} * (1-x)$, since each flip is independent from the next.

Thus, the probability-weighted number of steps that the algorithm executes when it stops on the i-th iteration is $(x^{i-1} * (1-x) * (B + F * (i-1) + S)$.

In the absolute worst case, this algorithm will run forever (if it keeps flipping the same values), giving an upper bound of inf on the number of iterations it must execute.

Thus, the expected number of steps that the algorithm needs to execute is equal to the probability-weighted number of steps summed from $i = 1$ to ∞. Therefore:

$$E = \sum_{i=1}^{\infty}(x^{i-1} * (1-x) * (B + F * (i-1) + S)$$

From the above, we know that B, F, S are equal to $2, 3, 2$, respectively, so:

$$E = \sum_{i=1}^{\infty}(x^{i-1} * (1-x) * (2 + 3 * (i-1) + 2))$$
$$= (1-x) * \sum_{i=1}^{\infty}(x^{i-1} * (3i+1))$$
$$= \tfrac{1-x}{x}(\sum_{i=1}^{\infty}(x^i * (3i+1)))$$
$$= \tfrac{1-x}{x}(3\sum_{i=1}^{\infty} i * x^i + \sum_{i=1}^{\infty} x^i)$$

After calculating the value of the two power series, we get:

$\sum_{i=1}^{\infty} i * x^i = \frac{x}{(x-1)^2}$ and $\sum_{i=1}^{\infty} x^i = \frac{-x}{x-1}$

$$E = \tfrac{1-x}{x}(\tfrac{3x}{(x-1)^2} - \tfrac{x}{x-1}) = (1-x)*(\tfrac{3}{(x-1)^2} - \tfrac{1}{x-1}) = \tfrac{1-x}{x-1}(\tfrac{3}{x-1} - 1) = -(\tfrac{3-(x-1)}{x-1}) = \tfrac{4-x}{1-x}$$

The probability that the algorithm gets either $(1,1)$ or $(0,0)$ is equal to:
$\Pr[(1,1)] + \Pr[(0,0)] = (\Pr[1] * \Pr[1]) + (\Pr[0] * \Pr[0]) = (p*p) + ((1-p)*(1-p)) = (p^2) + (p^2 - 2p + 1) = 2p^2 - 2p + 1$.

By our above reasoning, we know that x is equal to $(2p^2 - 2p + 1)$. Thus:
$E = \frac{4-x}{1-x} = \frac{4-(2p^2-2p+1)}{1-(2p^2-2p+1)} = \frac{3+2p-2p^2}{2(p-p^2)}$

3. For this question, we will be using a forest implementation of the disjoint set ADT, using the weighted union and path compression rules. According to the lecture where we were taught this implementation of disjoint sets, the worst case cost of doing a

sequence of $n-1$ unions and m finds in this implementation is $O(n+m\log^* n)$. Therefore, in order to maintain this upper bound, our algorithm can only add a constant number of steps.

For our algorithm to keep track of the number of nodes at every root, we will be maintaining an array R where each element is a tuple. The first element of the tuple corresponds to the vertex in V at the same index as R, and the second element will be the total number of nodes attached to that vertex or 0 if it is not a root node. For example, if every node was a singleton, $R_i = (V_i, 1)$, where i is the index of both R_i in R and V_i in V. We will be assuming that for each element in V, the index is also the value at that index, such that $R_i = V_i = i$. This array will only take a constant amount of steps to maintain, keeping our worst-case run time stable alongside regular disjoint set operations like $Makeset(x)$ and $Union(x, y)$.

Our algorithm will first call $Makeset(X)$ for all $x \in V$, such that each vertex in V will become a singleton and at the same time we will be populating R for each X we encounter in V. Since appending to R will add a constant amount of steps to $Makeset(X)$, the run-time of $Makeset(X)$ and our algorithm is unaffected.

Before we begin looping through E, we can input $C[0]$ to equal 1, since we can only possibly have singletons when no edges exist yet. Each element in E corresponds to an edge between two vertices in V, so we can loop through E, and for each element call $Union(V_i, V_j)$ where V_i and V_j correspond to the two values in an element of E, as defined in the question. When $Union(V_i, V_j)$ is called, we can access R_i and R_j to find the root nodes of the trees we are merging by simply getting $R[i]$ and $R[j]$ respectively, in constant time, by definition of R. Whichever node becomes the new tree's root node will have their tuple's second value increase by the other node's second value (thus adding both tree's number of nodes together) and the other node's second value will become 0 (since it is no longer a root node). We do not delete the element in R to preserve the indices of all the nodes after it and maintain constant find time.

Since we don't need to output which node has the largest number of nodes, nor can C ever decrease since we never destroy edges, we only need to compare the new value we obtained when adding our unionized nodes' size with the last value we input into C. If it is larger, we place the new value into C, if it is not, then the last value of C is still the largest, and we place that value. Finding the right R tuples by index, doing the arithmetic to change their size values, and the comparison and input of the next C value all take a constant amount of time, thus attaching these steps to $Union(x, y)$ does not affect the run time.

Once we finish looping through E, we should then have a completed and correct C

array which can then be returned at the end of the algorithm. Since every modification made to both $Union(x, y)$ and $Makeset(x)$ only added a constant number of steps, the worst-case runtime is maintained and is $O(n + m \log^* n)$.