Problem 1: Let \( t = |S| \) denote the actual number of “increment” commands received by the device. Let \( X \) denote the RV that indicates the count at the end of the sequence \( S \). Further, let \( x_i \) denote the indicator RV associated with the \( i \)th “increment” command, \( 1 \leq i \leq t \); thus, \( x_i = 1 \) indicates that the counter correctly carried out the increment. Since the counter is correctly initialized to 0, we have \( X = \sum_{i=1}^{t} x_i \), and \( E[X] \) gives the expected count.

By linearity of expectation, \( E[X] = \sum_{i=1}^{t} E[x_i] \). Now, for any \( i \), \( E[x_i] = \Pr\{x_i = 1\} = 1/2 \) (given). Therefore, \( E[X] = t/2 \). Since the given strategy reports twice the count, the expected value of the count returned by the strategy = \( 2 \times (t/2) = t = |S| \).

Problem 2: Let \( m = |E| \) denote the number of edges in the graph and let \( e_1, e_2, \ldots, e_m \) denote the edges themselves. Let \( y_i \) denote the indicator RV associated with edge \( e_i \), \( 1 \leq i \leq m \); thus, \( y_i = 1 \) indicates that edge \( e_i \) is well colored. Let \( Y \) denote the RV that gives the number of edges that are well colored by the uniform random coloring scheme. Thus, \( Y = \sum_{i=1}^{m} y_i \). By linearity of expectation, \( E[Y] = \sum_{i=1}^{m} E[y_i] \). For any \( i \), we compute \( E[y_i] \) as follows.

Since \( y_i \) is an indicator RV, \( E[y_i] = \Pr\{y_i = 1\} \). To compute this probability, let the edge \( e_i \) join two nodes \( u \) and \( v \). Since we can color \( u \) in 3 ways and \( v \) in 3 ways, the total number of combinations of colors to these two nodes is \( 3 \times 3 = 9 \). Of these 9 combinations of colors to \( u \) and \( v \), only 3 combinations (i.e., those in which both \( u \) and \( v \) assigned the same color) cause the edge to be not well colored. Therefore, 6 of the 9 combinations cause edge \( e_i \) to be well colored. Since all of the 9 color combinations are equally likely, we have \( \Pr\{y_i = 1\} = 6/9 = 2/3 \). In other words, \( E[y_i] = 2/3 \).

Hence \( E[Y] = \sum_{i=1}^{m} E[y_i] = 2m/3 \). In any coloring, the maximum number \( e^* \) of edges that can be well colored is at most \( m \), the total number of edges. Therefore, \( E[Y] \geq (2/3)e^* \).

Problem 3:

Part (a): For any \( d \geq 2 \) and \( n \geq 1 \), the height of a \( d \)-ary heap with \( n \) nodes is equal to \( \lfloor \log_d [n(d-1)] \rfloor \).

Proof: We will use the following (easy to verify) fact in the proof.

Fact 1: Let \( k \) and \( d \) be positive integers and let \( r \) be nonnegative integer such that \( d^r \leq k < d^{r+1} \). Then, \( r = \lfloor \log_d k \rfloor \).
If the heap has height = 0, the number of nodes in the heap is just 1. In this case, \[\lfloor \log_d (d - 1) \rfloor = 0\] and so the formula for the height is correct.

So, assume that we have a \(d\)-ary heap of height \(h \geq 1\). Let \(n\) denote the number of nodes in the heap. Suppose we number the levels of the heap using integers 0 through \(h\), where the root is at level 0, the children of the root are at level 1, the children of level 1 nodes are at level 2, etc. For each of the levels 0 through \(h - 1\), the number of nodes in level \(i\) is \(d^i\), since all these levels are complete. Thus, the total number of nodes in levels 0 through \(h - 1\) is \(\sum_{i=0}^{h-1} d^i = (d^h - 1)/(d - 1)\).

Level \(h\) contains at least 1 node and at most \(d^h\) nodes. Therefore, the total number of nodes \(n\) in the heap satisfies the following inequality:

\[
1 + \frac{(d^h - 1)}{(d - 1)} \leq n \leq \frac{d^h}{(d - 1)}. \tag{1}
\]

From Inequality (1) in conjunction with the given condition that \(d \geq 2\), it is easy to see that

\[
d^h \leq n(d - 1) < d^{h+1}. \tag{2}
\]

From Inequality (2) and Fact 1, we have \(h = \lfloor \log_d [n(d - 1)] \rfloor\).

**Part (b):** The algorithm for \textsc{Extract-Max} for a \(d\)-ary heap is similar to that of a binary heap. We need to develop the corresponding \textsc{Heapify} algorithm and use it in performing \textsc{Extract-Max}.

When a \(d\)-ary heap is stored in an array \(A[1..n]\), the children of the node at index \(i \geq 1\) are at indices \((i - 1)d + 2, (i - 1)d + 3, \ldots, id + 1\). (If any of these indices is greater than the heap size, then the corresponding child does not exist.) For simplicity, in the following pseudocode, we don’t check whether an index is outside the heap. We also assume that the heap size is \(n\).

**\textsc{Heapify}(A, i)**

1. Let \(c_1, c_2, \ldots, cd\) denote the indices of the children of \(i\).
   
   Find the maximum value among \(A[i], A[c_1], A[c_2], \ldots, A[cd]\). 
   
   Let \(r\) be the index containing the maximum value.

   /* Swap \(A[i]\) with \(A[r]\) and use recursion. */

2. if (\(i \neq r\))
   
   2.1 Exchange \(A[i]\) with \(A[r]\).
   
   2.2 \textsc{Heapify}(A, r).

The running time of \textsc{Heapify} is \(O(d \times \text{height}(i))\) since the function spends \(O(d)\) time in Step 1 and goes down to the next level. Since \(\text{height}(i)\) is at most \(\lfloor \log_d [n(d - 1)] \rfloor\), the time for \textsc{Heapify} is \(O(d \log_d [n(d - 1)])\).
**Extract-Max(A)**

2. Heapify(A,1).
3. return max.

The running time of Extract-Max is dominated by the time for Heapify(A,1) (Step 2 above). In other words, the running time of Extract-Max is $O(d \log_d [n(d - 1)])$.

---

**Problem 4:**

**Main idea:** Suppose we are at row $i$ of the matrix and we know the largest column number $j$ such that $A[i,j]$ is 1. We know that in each row of $A$, all the 1's come before the 0's. Therefore, in subsequent steps, we need to consider only those rows which have 1's in columns greater than $j$.

To start this search, we can scan the first row of the matrix $A$ and find the rightmost (i.e., largest) column index $j$ such that $A[1,j]$ is 1. (Clearly, this can be done in $O(n)$ time.)

**Description of the algorithm:** The algorithm keeps track of the current index of a row with maximum number of 1's in the variable best. Variable $i$ represents the index of the row being considered and $j$ represents the largest column index in row best such that the matrix entry $A[best, j]$ is 1. We also use a temporary variable $k$ as a loop index.

1. Let best = 1.
2. Find the largest index $j$ of row 1 such that $A[1,j]$ = 1.
   - if ($j = n$) /* Row 1 has n 1’s. */
     - print 1 as the answer and stop.
3. for $i$ = 2 to $n$ do
   /* Check if row $i$ has a 1 in some column larger than $j$. */
   3.1. for $k$ = $j+1$ to $n$ do
     - if ($A[i,k] = 0$) break;
   3.2. if ($k = n+1$) /* The no. of 1’s in Row i is n. */
     - then print $i$ as the answer and stop.
   else {
     - if ($k > j+1$) /* Does row $i$ have more than $j$ 1’s? */
       - then { /* Update best and j. */
         - best = $i$; $j = k-1$;
       }
   }
   /* End of for loop for $i$ */
4. Print best as the answer.
Running Time Analysis: Step 1 runs in $O(1)$ time and Step 2 runs in $O(n)$ time.

In Step 3, even though there are two nested loops, the running time of that step is just $O(n)$. This can be seen as follows.

(a) Each execution of Step 3.2 uses only $O(1)$ time. Therefore, through all the iterations of the for loop of Step 3, the total time for Step 3.2 is $O(n)$.

(b) To analyze the time spent by the loop in Step 3.1, note that each iteration of the loop in Step 3.1 merely checks whether $A[i, k]$ is zero. So, the total time spent in Step 3.1 over all the iterations of the loop in Step 3 is bounded by a constant times the number of entries of $A$ examined in Step 3.1. Further, if entry $A[i, k]$ is examined in some step, then the next entry that is examined is either $A[i, k + 1]$ or $A[i + 1, k]$; that is, in the next entry either the row index or the column index increases by 1. Since each of these indices can increase at most $n$ times, the total number of entries examined is at most $2n$. Therefore, the time taken by Step 3.1 is also $O(n)$.

Step 4 runs in $O(1)$ time. Thus, the overall running time of the algorithm is $O(n)$. 