1. Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t whose length is a multiple of 3.

Solution (graph modeling): We construct another directed graph $H = (V', E')$ as follows:

- $V' = V \times \{0, 1, 2\}$
- $E' = \big\{(u, i) \to (v, i + 1 \bmod 3) \;\big|\; u \to v \in E\big\}$

We need to determine if there is a path (or walk) from $(s, 0)$ to $(t, 0)$—or equivalently, if $(t, 0)$ is reachable from $(s, 0)$—in H. We can solve this problem using whatever-first search from $(s, 0)$ in $O(V' + E') = O(V + E)$ time. ■

Solution (dynamic programming (7/10)): We solve the problem using a variant of Bellman-Ford. For any vertex v and any integer ℓ, define $Walk(v, \ell) = \textsc{True}$ if there is a walk from s to v of length ℓ, and $Walk(v, \ell) = \textsc{False}$ otherwise.

We need to compute $\bigvee_{i=0}^{V} Walk(v, 3i)$. Suppose there is a walk from s to t of length $3k$ for some $k > V$. That walk must visit some vertex v at least three times, which means it must include a closed walk from v to v whose length is a multiple of 3. Removing that loop gives us a shorter walk from s to t whose length is a multiple of 3.

The *Walk* function satisfies the following recurrence:

$$Walk(v, \ell) = \begin{cases} \textsc{True} & \text{if } v = s \text{ and } \ell = 0 \\ \textsc{False} & \text{if } v \neq s \text{ and } \ell = 0 \\ \displaystyle\bigwedge_{u \to v} Walk(v, \ell - 1) & \text{otherwise} \end{cases}$$

We can memoize this function into a $V \times (3V + 1)$ array, indexed by vertices v and walk lengths $0 \le \ell \le 3V$. We can fill the array by increasing ℓ in the outer loop, and considering vertices in arbitrary order in the inner loop. The resulting algorithm runs in ***O(VE) time*** — within each iteration of the outer loop, we consider every edge $u \to v$ exactly once, just like Bellman-Ford. ■

Rubric: 10 points: 2 for vertices + 2 for edges + 2 for problem (reachability) + 2 for algorithm + 2 for running time.

A correct algorithm that runs in $O(VE)$ time is worth 7 points (scaled dynamic programming rubric). No proof of correctness is required, but the algorithm must be completely specified. (In particular, only V iterations of the outer loop is not enough.)

2. Describe and analyze an algorithm to decide, given three strings X, Y, and Z, whether Z is a smooth shuffle of X and Y.

Solution: Suppose the input strings are $X[1..m]$, $Y[1..n]$, and $Z[1..n+m]$. (If the length of Z is not the sum of the lengths of X and Y, we can immediately return FALSE.)

To simplify boundary cases, we add sentinel characters $X[-1] = X[0] = \spadesuit$ and $Y[-1] = Y[0] = \heartsuit$, where $\spadesuit$ and $\heartsuit$ are new symbols that do not appear elsewhere in the input strings.

For any indices i and j, we define two auxiliary boolean functions:

- $SmoothX(i,j) =$ TRUE if and only if $Z[1..i+j]$ is a smooth shuffle of $X[1..i]$ and $Y[1..j]$ ***whose last symbol (if any) comes from X***.
- $SmoothY(i,j) =$ TRUE if and only if $Z[1..i+j]$ is a smooth shuffle of $X[1..i]$ and $Y[1..j]$ ***whose last symbol (if any) comes from Y***.

We need to compute $SmoothX(m,n) \vee SmoothY(m,n)$.

These two functions obey the following mutual recurrences:

$$SmoothX(i,j) = \begin{cases} \text{TRUE} & \text{if } i = j = 0 \\ \text{FALSE} & \text{if } X[i] \neq Z[i+j] \\ SmoothY(i-1,j) & \text{if } X[i] = Z[i+j] \\ & \quad \text{and } X[i-1] \neq Z[i+j-1] \\ SmoothY(i-1,j) \vee SmoothY(i-2,j) & \text{if } X[i] = Z[i+j] \\ & \quad \text{and } X[i-1] = Z[i+j-1] \end{cases}$$

$$SmoothY(i,j) = \begin{cases} \text{TRUE} & \text{if } i = j = 0 \\ \text{FALSE} & \text{if } Y[j] \neq Z[i+j] \\ SmoothX(i,j-1) & \text{if } Y[j] = Z[i+j] \\ & \quad \text{and } Y[j-1] \neq Z[i+j-1] \\ SmoothX(i,j-1) \vee SmoothX(i,j-2) & \text{if } Y[j] = Z[i+j] \\ & \quad \text{and } Y[j-1] = Z[i+j-1] \end{cases}$$

We can memoize these functions into two two-dimensional arrays $SmoothX[0..m, 0..n]$ and $SmoothY[0..m, 0..n]$. We can simultaneously fill these two arrays in standard row-major order (increasing i in the outer loop and increasing j in the inner loop) in ***$O(mn)$ time***. ■

Rubric: 10 points: dynamic programming rubric. This is not the only correct solution.
Omitting the English description of the recursive function(s) is a Deadly Sin.

3. (a) Describe an algorithm that simulates a fair coin, using independent rolls of a fair three-sided die as your only source of randomness.

   (b) What is the expected number of die rolls performed by your algorithm in part (a)?

Solution:

```
FAIRCOIN:
  r ← ROLLD3
  if r = 1
      return HEADS
  else if r = 2
      return TAILS
  else
      return FAIRCOIN
```

Let X be the number of die rolls performed by this algorithm. We immediately have

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}[X \mid r=1]\cdot\Pr[r=1] + \mathrm{E}[X \mid r=2]\cdot\Pr[r=2] + \mathrm{E}[X \mid r=3]\cdot\Pr[r=3] \\
&= 1 \cdot \frac{1}{3} + 1 \cdot \frac{1}{3} + (1+E[X]) \cdot \frac{1}{3} \\
&= 1 + \frac{E[X]}{3},
\end{aligned}$$

which implies that The expected number of die rolls is **3/2**. ■

Rubric: 5 points = 3 for part (a) + 2 for part (b). −1 in part (a) for any algorithm needing more than $3/2$ expected rolls. Credit for part (b) requires a correct analysis of the algorithm submitted for part (a). No proof of correctness or derivation of expected rolls is required.

(c) Describe an algorithm that simulates a fair three-sided die, using independent fair coin flips as your only source of randomness.

(d) What is the expected number of coin flips performed by your algorithm in part (c)?

Solution (two flips):

```
RollD3:
  a ← CoinFlip
  b ← CoinFlip
  if a = Heads and b = Heads
      return 1
  else if a = Heads and b = Tails
      return 2
  else if a = Tails and b = Heads
      return 3
  else
      return CoinFlip
```

Let X be the number of coin flips performed by this algorithm. We immediately have

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}[X \mid ab \neq HT] \cdot \Pr[ab \neq HT] + \mathrm{E}[X \mid ab = HT] \cdot \Pr[ab = HT] \\
&= 2 \cdot \frac{3}{4} + (2 + E[X]) \cdot \frac{1}{4} \\
&= 2 + \frac{E[X]}{4},
\end{aligned}$$

which implies that The expected number of die rolls is **8/3**. ■

Solution (three flips (4/5)):

```
RollD3:
  a ← CoinFlip
  b ← CoinFlip
  c ← CoinFlip
  if a = b = c
      return RollD3
  else if b = c
      return 1
  else if a = c
      return 2
  else ⟨⟨a = b⟩⟩
      return 3
```

Let X be the number of coin flips performed by this algorithm. We immediately have

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}[X \mid a = b = c] \cdot \Pr[a = b = c] + \mathrm{E}[X \mid \neg(a = b = c)] \cdot \Pr[\neg(a = b = c)] \\
&= (3 + E[X]) \cdot \frac{1}{4} + 3 \cdot \frac{3}{4} \\
&= 3 + \frac{E[X]}{4},
\end{aligned}$$

which implies that The expected number of die rolls is **4**. ■

Rubric: 5 points = 3 for part (c) + 2 for part (d). −1 for any solution needing more than 8/3 expected flips. Credit for part (d) requires a correct analysis of the algorithm submitted for part (c). No proof of correctness or derivation of expected flips is required.

4. (a) Describe and analyze an algorithm that computes the probability that Dirk wins the game against Death, assuming Dirk plays randomly and Death plays to win.

Solution (tree): Suppose the tree T is explicitly given in the input. For any node v *with even depth*, let $Dirk(v)$ denote the probability that Dirk wins if the game starts at node v. We need to compute $Dirk(root(T))$.

$$Dirk(v) = \begin{cases} 1 & \text{if } v \text{ is a white leaf} \\ 0 & \text{if } v \text{ is a black leaf} \\ \dfrac{1}{2}\min\{Dirk(v.left.left),\ Dirk(v.left.right)\} & \\ \qquad + \dfrac{1}{2}\min\{Dirk(v.right.left),\ Dirk(v.right.right)\} & \text{otherwise} \end{cases}$$

We can evaluate this recurrence by a simple post-order traversal of the tree in $\boldsymbol{O(4^n)}$ ***time***. ■

Solution (leaf intervals): Let $L[1..4^n]$ be the input array of bit representing the colors of the leaves, where $L[j] = 1$ if and only if the jth leaf is white.

Let $\boldsymbol{v(i,\ell)}$ denote the root of the subtree of T whose leaves are stored in the interval $L[i..i+\ell-1]$; this node exists only if ℓ is a power of 2 and $i \bmod \ell = 1$. The children of $v(i,\ell)$ are $v(i,\ell/2)$ and $v(i+\ell/2,\ell/2)$.

For any integer i and ℓ such that ℓ is a power of 4 (sic) and $i \bmod \ell = 1$, let $\boldsymbol{Dirk(i,\ell)}$ denote the probability that Dirk wins if the game starts at node $v(i,\ell)$. We need to compute $Dirk(0,4^n)$.

$$Dirk(i,\ell) = \begin{cases} L[i] & \text{if } \ell = 1 \\ \dfrac{1}{2}\min\begin{Bmatrix} Dirk(i, & \ell/4), \\ Dirk(i+\ell/4, & \ell/4) \end{Bmatrix} & \\ \qquad + \dfrac{1}{2}\min\begin{Bmatrix} Dirk(i+\ell/2, & \ell/4), \\ Dirk(i+3\ell/4, & \ell/4) \end{Bmatrix} & \text{otherwise} \end{cases}$$

Let $T(n)$ denote the time to compute $Dirk(0,4^n)$ by naive recursion. This running time obeys the recurrence $T(n) = O(1) + 4T(n-1)$, which implies that the recursive algorithm runs in $\boldsymbol{O(4^n)}$ ***time***. ■

Solution (implicit tree): Let $L[1..4^n]$ be the input array of bit representing the colors of the leaves, where $L[j] = 1$ if and only if the jth leaf is white.

We use the standard array representation of binary trees (usually applied to binary *heaps*) to represent the tree T. Each node in T is represented by an integer between 1 and $2 \cdot 4^k - 1$. The integer 1 represents the root of T, integers $2k$ and $2k+1$ represent the left and right children of k, and the integer $(4^n - 1) + \ell$ represents the ℓth leaf from the left. The depth of any node k is one less than the number of bits in the binary representation of k.

For any integer k that represents a node with even depth (or equivalently, whose binary representation has odd length), let $Dirk(k)$ denote the probability that Dirk wins if the game is started at that node. We need to compute $Dirk(1)$.

$$Dirk(k) = \begin{cases} L[k-(4^n-1)] & \text{if } k > 4^n - 1 \\ \dfrac{1}{2}\min\left\{Dirk(4k), Dirk(4k+1)\right\} & \\ \qquad + \dfrac{1}{2}\min\left\{Dirk(4k+2), Dirk(4k+3)\right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $Dirk[1..2\cdot 4^k - 1]$, which we can fill in reverse order in $\boldsymbol{O(4^n)}$ ***time***. ■

Rubric: 8 points = 3 for English + 4 for recurrence + 1 for running time, as in the standard dynamic programming rubric. (No additional points for memoization, because memoization doesn't actually help!) No penalty for assuming the tree is given explicitly. These are not the only correct solutions. ***Omitting the English description of the recursive function is a Deadly Sin.***

(b) Describe and analyze an algorithm that computes the probability that Dirk wins the game again Death, assuming both players play randomly.

Solution: Death and Dirk are just choosing a leaf uniformly at random, so the probability that Dirk wins is precisely the fraction of leaves that are white.

Let $L[1..4^n]$ be the input array of bit representing the colors of the leaves, where $L[j] = 1$ if and only if the jth leaf is white.

```
RandomDeathGame(L[1..4^n]):
  sum ← 0
  for i ← 1 to 4^n
      sum ← sum + L[i]
  return sum/4^n
```

The algorithm runs in $O(4^n)$ *time*. ■

Solution: For any node v, let $RDG(v)$ (short for "random Death game") denote the probability that a randomly chosen path downward from v ends at a white leaf. We need to compute $RDG(root(T))$.

$$RDG(v) = \begin{cases} 1 & \text{if } v \text{ is a white leaf} \\ 0 & \text{if } v \text{ is a black leaf} \\ \dfrac{RDG(v.left) + RDG(v.right)}{2} & \text{otherwise} \end{cases}$$

We can evaluate this recurrence using a standard post-order traversal of the tree in $O(4^n)$ *time*. ■

Rubric: 2 points.