1. For any positive integer \( n \), the \( n \)th \textbf{Fibonacci string} \( F_n \) is defined recursively as follows, where \( x \cdot y \) denotes the concatenation of strings \( x \) and \( y \):

\[
\begin{align*}
F_1 & := 0 \\
F_2 & := 1 \\
F_n & := F_{n-1} \cdot F_{n-2} \quad \text{for all } n \geq 3
\end{align*}
\]

For example, \( F_3 = 10 \) and \( F_4 = 101 \).

(a) What is \( F_8 \)?

\textbf{Solution:} \( 101101011010110101 \)

\textbf{Rubric:} 2 points:

(b) \textbf{Prove} that every Fibonacci string except \( F_1 \) starts with 1.

\textbf{Solution:} Let \( n \) be an arbitrary positive integer greater than 1. Assume that \( F_k \) starts with 1 for every positive integer \( k \) such that \( 2 \leq k < n \). There are two cases to consider.

- If \( n = 2 \), then \( F_n \) starts with 1 by definition.
- If \( n > 2 \), then \( F_n = F_{n-1} \cdot F_{n-2} \). The definition of \( \cdot \) implies that \( F_n \) starts with the first symbol of \( F_{n-1} \), and the inductive hypothesis implies that the first symbol of \( F_{n-1} \) is 1.

In both cases, \( F_n \) starts with 1.

\textbf{Rubric:} 4 points: standard induction rubric (scaled)

(c) \textbf{Prove} that no Fibonacci string contains the substring \texttt{00}.

\textbf{Solution:} Let \( n \) be an arbitrary positive integer. Assume for all positive integers \( k < n \) that \( F_k \) does not contain the substring \texttt{00}. There are two cases to consider.

- If \( n = 1 \), then \( F_n = 0 \) does not contain the substring \texttt{00}.
- If \( n = 2 \), then \( F_n = 1 \) does not contain the substring \texttt{00}.
- If \( n = 3 \), then \( F_n = 10 \) does not contain the substring \texttt{00}.
- If \( n \geq 4 \), then \( F_n = F_{n-1} \cdot F_{n-2} \). Thus, any substring of \( F_n \) either is a substring of \( F_{n-1} \) or \( F_{n-2} \), or contains the first symbol in \( F_{n-2} \).
  - The inductive hypothesis implies that \( F_{n-1} \) and \( F_{n-2} \) do not contain the substring \texttt{00}.
  - Part (b) implies that \( F_{n-2} \) begins with 1 (because \( n-2 \geq 2 \)), so the substring \texttt{00} cannot contain the first symbol of \( F_{n-2} \).

In all cases, we conclude that \( F_n \) does not contain the substring \texttt{00}.

\textbf{Rubric:} 4 points: standard induction rubric (scaled)
2. Design and analyze an algorithm that computes the maximum total score you can achieve by selectively pulling all-nighters, given the arrays \( \text{Score}[1..n] \) and \( \text{Bonus}[1..n] \) as input. If you submit the \( i \)th assignment immediately after pulling \( k \) consecutive all-nighters, your score for that assignment will be \( (\text{Score}[i] + \text{Bonus}[i])/2^{k-1} \).

**Solution:** For any integers \( k \) and \( i \), let \( \text{MaxScore}(i, k) \) be the maximum score total score I can get on assignments \( i..n \), assuming I have just pulled \( k \) consecutive all-nighters. We need to compute \( \text{MaxScore}(1, 0) \). This function obeys the following recurrence:

\[
\text{MaxScore}(i, k) = \begin{cases} 
0 & \text{if } i > n \\
\max \left\{ \text{MaxScore}(i + 1, k + 1) + (\text{Score}[i] + \text{Bonus}[i])/2^k \right\} & \text{otherwise}
\end{cases}
\]

We can memoize this function into a two-dimensional array \( \text{MaxScore}[1..n, 0..n] \) indexed by \( i \) and \( k \). We can fill this array by decreasing \( i \) in the outer loop, and by increasing \( k \) in the inner loop, in \( O(n^2) \) time.

**Solution:** To save some space, I’ll write \( S[\cdot] \) and \( B[\cdot] \) everywhere instead of \( \text{Score}[\cdot] \) and \( \text{Bonus}[\cdot] \), respectively.

For any integer \( i \), let \( \text{MaxSc}(i) \) be the maximum score total score I can get on assignments \( i..n \) assuming I do not pull an all-nighter for assignment \( i \). To simplify the algorithm, we add a sentinel 0th assignment with \( S[0] = 0 \); then we need to compute \( \text{MaxSc}(0) \). This function obeys the following recurrence.

\[
\text{MaxSc}(i) = \begin{cases} 
0 & \text{if } i > n \\
S[i] + \max_{\ell=0}^{n-i} \left\{ \sum_{j=1}^{\ell} S[i+j] + B[i+j] + \text{MaxSc}(i + \ell + 1) \right\} & \text{otherwise}
\end{cases}
\]

We can memoize this function into a one-dimensional array \( \text{MaxSc}[0..n] \). We can fill this array from right to left (decreasing \( i \)). A naive implementation runs in \( O(n^3) \) time, because we have three variables \( i, \ell, \) and \( j \) on the right side of the recurrence, but we can do better. For any integers \( i \) and \( \ell \), define

\[
\text{Sum}(i, \ell) := \sum_{j=1}^{\ell} S[i+j] + B[i+j] / 2^{j-1}
\]

so that \( \text{MaxSc}(i) = S[i] + \max_{\ell} (\text{Sum}(i, \ell) + \text{MaxSc}(i + \ell + 1)) \). Then we have

\[
\text{Sum}(i, \ell) = \text{Sum}(i, \ell - 1) + S[i + \ell] + B[i + \ell] / 2^{\ell-1}
\]

We can memoize this function into a new two-dimensional array; we can fill each row of this array from right to left (decreasing \( \ell \)), and we can consider the rows in any order we like. Thus we can compute \( \text{Sum}(i, \ell) \) for all \( i \) and \( \ell \) in \( O(n^2) \) time. After this preprocessing step, we can compute \( \text{MaxSc}(i) \) for all \( i \) in \( O(n^2) \) time.

**Rubric:** 10 points: standard dynamic programming rubric. Omitting the English description earned a 3-point penalty instead of an automatic zero. These are not the only correct solutions.
3. The following algorithm finds the smallest element in an unsorted array. The subroutine Shuffle randomly permutes the input array $A$; every permutation of $A$ is equally likely. Assume no two elements of $A$ are equal.

```plaintext
RANDOMMIN$(A[1..n])$
\begin{align*}
\text{min} & \leftarrow \infty \\
\text{SHUFFLE}(A) \\
\text{for } i \leftarrow 1 \text{ to } n \\
& \quad \text{if } A[i] < \text{min} \\
& \quad \text{min} \leftarrow A[i] \\
\text{return } \text{min}
\end{align*}
```

(a) In the worst case, how many times does RandomMin execute line $(\star)$?

**Solution:** $n$  

**Rubric:** 2 points: all or nothing

(b) For each index $i$, let $X_i = 1$ if line $(\star)$ is executed in the $j$th iteration of the for loop, and let $X_i = 0$ otherwise. What is $\Pr[X_i = 1]$?

**Solution:** $X_i = 1$ if and only if $A[i]$ is the smallest element in $A[1..i]$. Since every permutation of $A[1..i]$ is equally likely, $\Pr[X_i = 1] = 1/i$.  

**Rubric:** 2 points: all or nothing

(c) What is the exact expected number of executions of line $(\star)$?

**Solution:** $E[\sum_{i=1}^n X_i] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n \Pr[X_i = 1] = \sum_{i=1}^n 1/i = H_n$.  

**Rubric:** 3 points  
- 2 points for "$\sum_{i=1}^n 1/i$" or "$O(\log n)$" or "$\ln n$" or other additive error  
- 1 point for "$\log n$" or other multiplicative error.

(d) **Prove** that line $(\star)$ is executed $O(\log n)$ times with high probability, assuming the variables $X_i$ are mutually independent.

**Solution:** Let $X = \sum_i X_i$. Apply Chernoff’s bound with $\delta = e - 1$ and $\mu = E[X] = H_n$:

$$
\Pr[X > e H_n] < \left( \frac{e^\delta}{(1 + \delta)^{1+\delta}} \right) ^\mu = \left( \frac{e^{e-1}}{e^e} \right) ^{H_n} = e^{-H_n} < e^{-\ln(n+1)} = \frac{1}{n+1}.
$$

The second inequality follows from the estimate $\ln(n+1) < H_n$. We conclude that line $(\star)$ is executed at most $e H_n$ times with high probability.  

**Rubric:** 3 points. The particular choice of $\delta$ doesn’t matter. No penalty for implicitly assuming $\mu = \ln n$. 


(e) [Extra credit] **Prove** that the variables $X_i$ are mutually independent.

**Solution:** The behavior of the algorithm depends only on the **ranks** of the elements of $A$, not their particular values. So without loss of generality, assume that $A$ is a permutation of the integers $1, 2, \ldots, n$. Calling `Shuffle(A)` ensures that $A$ stores a **random** permutation of the integers $1, 2, \ldots, n$. There are $n!$ such permutations.

Let $R = (R_1, R_2, \ldots, R_n)$ be the random integer vector where for each index $i$,

$$R_i = \# \{ h \mid 1 \leq h \leq i \text{ and } A[h] \leq A[i] \}.$$

In other words, $R_i$ is the rank of $A[i]$ in the prefix $A[1..i]$.

By definition, the vector $R$ is determined entirely by the permutation stored in $A$; conversely, given only $R$, we can recover the permutation $A$. Thus, there is a bijection between the $n!$ permutations of $A$ and the set of $n!$ possible values of $R$. Because $A$ is distributed uniformly at random, we conclude that $R$ is also distributed uniformly at random.

But the uniform distribution of $R$ is just the product of uniform distributions of the individual prefix ranks $R_i$. For all integer vectors $r = (r_1, r_2, \ldots, r_n)$ where $1 \leq r_i \leq i$ for each $i$, we have

$$\Pr[R = r] = \Pr\left[ \bigwedge_{i=1}^{n} R_i = r_i \right] = \frac{1}{n!} = \prod_{i=1}^{n} \frac{1}{i} = \prod_{i=1}^{n} \Pr[R_i = r_i].$$

In other words, the random variables $R_i$ are mutually independent.

Finally, for each index $i$, we have $X_i = 1$ if and only if $R_i = 1$; thus, each variable $X_i$ depends only on the corresponding variable $R_i$. It follows immediately that the variables $X_i$ are mutually independent, as required. ■

**Rubric:** Up to 5 points, recorded separately as extra credit. This is not the only correct proof!
4. Describe and analyze an algorithm to determine Elmo’s expected score, given the initial sequence of \( n \) cards as input. Assume Elmo moves first. Elmo plays greedily; his opponent Daisy plays randomly.

**Solution:** Let \( C[1..n] \) be the input card values. Assume all card values are distinct (since otherwise we don’t know how Elmo plays).

For any indices \( i \) and \( j \), let \( EES(i,j) \) denote Elmo’s Expected Score if Elmo plays first, starting with the cards \( C[i..j] \). We need to compute \( EES(1,n) \). This function obeys the recurrence

\[
EES(i,j) = \begin{cases} 
0 & \text{if } i > j \\
C[i] & \text{if } i = j \\
\frac{EES(i+1,j-1) + EES(i+2,j)}{2} & \text{if } i < j \text{ and } C[i] > C[j] \\
\frac{EES(i+1,j-1) + EES(i,j-2)}{2} & \text{if } i < j \text{ and } C[i] < C[j]
\end{cases}
\]

This recurrence can be memoized into a two-dimensional array \( EES[1..n, 1..n] \), which we index by \( i \) and \( j \). We can fill the array by decreasing \( i \) in the outer loop and increasing \( j \) in the inner loop, in \( O(n^2) \) time.

**Rubric:** 10 points: Standard dynamic programming rubric. Omitting the English description earned a 3-point penalty instead of an automatic zero.