CS/Math 473 ✧ Fall 2022 Midterm 2 Solutions Problem 1	Name:

A line of n passengers is waiting to board the Hogwarts Express, which has n seats, each assigned to one passenger. The first passenger, Professor Dumbledore, has forgotten his assigned seat, so he chooses a seat uniformly at random. When any other passenger boards the train, if Dumbledore is sitting in their assigned seat, Dumbledore moves to a different *unoccupied* seat, again chosen uniformly at random.

(a) What is the exact probability that Dumbledore never moves after choosing his first seat?

Solution: $\boxed{\frac{1}{n}}$

Dumbledore never moves if and only if his first randomly chosen seat is his assigned seat. ■

Rubric: 2 points.

(b) What is the exact probability that the kth passenger has to ask Dumbledore to move? *[Hint: Consider the special cases $k = 2$ and $k = n$, in particular when $n = 3$.]*

Solution: $\boxed{\frac{1}{n-k+2}}$

When the kth passenger boards, passengers 2 through $k-1$ are in their assigned seats, and Dumbledore is equally likely to be in any of the other $n-(k-2)$ seats. ■

Rubric: 3 points.

(c) What is the exact expected number of times Dumbledore changes seats?

Solution: $\mathrm{E}[\#\text{moves}] = \sum_{k=2}^{n} \Pr[\text{Dumbo moves in step } k] = \sum_{k=2}^{n} \frac{1}{n-k+2} = \sum_{j=2}^{n} \frac{1}{j} = \boxed{H_n - 1}$ ■

Rubric: 5 points. 3 points for $\Theta(\log n)$.

CS/Math 473 ✧ Fall 2022 Midterm 2 Solutions Problem 2	Name:

Describe and efficient analyze algorithms for each of the following problems. The input to each problem is a pair of strings $P[1..m]$ and $T[1..n]$.

(a) For each index i, find the longest prefix of P ending at $T[i]$.

Solution: This information is already computed by KnuthMorrisPratt. Add a sentinel character $P[m+1] = \#$ different from every other character in P or T. Compute the failure function for P and then run the main KMP algorithm. At each iteration of the main loop, if we find an index j such that $P[j] = T[i]$, set $Pre[i] \leftarrow j$; if we do not find such an index, set $Pre[i] \leftarrow 0$. Finally, return the array $Pre[1..n]$. ■

Solution: Compute the KMP failure function for the string $P\#T\#$, where $\#$ is a sentinel character different from every other symbol in P or T. Then for each index $1 \le i \le n$, set $LP[i] \leftarrow fail[i+m]+1$. Finally, return the array $Pre[1..n]$. ■

Rubric: 3 points.

(b) For each index i, find the longest suffix of P starting at $T[i]$.

Solution: $\text{LongestSuffixes}(P,T) = rev(\text{LongestPrefixes}(rev(P), rev(T))$. ■

Rubric: 3 points.

(c) Determine whether P is an almost-substring of T.

This subproblem was removed from the exam; everybody received full credit.

Non-solution: The following algorithm runs in $O(m+n)$ time.

```
AlmostSubstring(P[1..m], T[1..n]):
  Pre[1..n] ← LongestPrefixes(P, T)
  Suf[1..n] ← LongestSuffixes(P, T)
  ⟨⟨Look for substring with one deletion⟩⟩
  for i ← 1 to n−1
      if Pre[i] + Suf[i+1] = m−1
          return True
  ⟨⟨Look for substring with at most one replacement⟩⟩
  for i ← 1 to n−2
      if Pre[i] + Suf[i+2] = m−1
          return True
  ⟨⟨Look for substring with one insertion⟩⟩
  for i ← 1 to n−2
      if Pre[i] + Suf[i+2] = m
          return True
  return False
```

This was the solution we had in mind, but it doesn't work. It's not enough to find *longest* prefixes and suffixes; we need to find prefixes and suffixes *of the right lengths*. For example, the pattern $P = \texttt{AAAAAA}$ is an almost-substring of the text $T = \texttt{AAAABAAAA}$, but the intended algorithm would return False. ♣

Solution (preprocessing via dynamic programming): Let's first consider only deletions: We want to find a substring of T that can also be obtained from P by deleting one character. The other two operations are similar. For each index i, we want to find a prefix of P that ends at $T[i]$ and a suffix of P that starts at $T[i+1]$ *whose lengths sum to exactly* $m-1$.

From parts (a) and (b) we can compute the lengths of the *longest* prefix of P that ends at $T[i]$ and the *longest* suffix of P that starts at $T[i+1]$, but these are not necessarily the prefix and suffix we want. Fortunately, the failure functions of P and its reversal actually encode *all* prefixes and suffixes of P that end at any point in T.

Recall that a *border* of a string is any proper prefix that is also a suffix. For any index i, let $Lborder[i] = fail(i+1)-1$ denote the length of the longest border of the prefix $P[1..i]$. Similarly, let $Rborder[j] = fair(j+1)-1$ be the length of the longest border of the suffix $P[m-j+1..m]$. (Here *fair* is the failure function of the reversal of P.) We can compute the arrays $Lborder[1..m]$ and $Rborder[1..m]$ in $O(m)$ time.

Now let $Subsring(l, r) =$ True if the string containing the first l symbols of P followed by the last r symbols of P contains P with one symbol deleted, and False otherwise. This function satisfies the following recurrence:

$$Subsring(l,r) = \begin{cases} \text{False} & \text{if } l + r < m-1 \\ \text{True} & \text{if } l + r = m-1 \\ Subsring(Lborder(l), r) \vee Subsring(l, Rborder(r)) & \text{otherwise} \end{cases}$$

We can memoize this recurrence into an $m \times m$ array, in standard row-major order, in $O(m^2)$ time.

We can similarly compute an array and $Subshtring[1..m, 1..m]$ to help search for P with one symbol replaced and P with one symbol inserted.

```
AlmostSubstring(P[1..m], T[1..n]):
  Pre[1..n] ← LongestPrefixes(P, T)
  Suf[1..n] ← LongestSuffixes(P, T)
  precompute arrays Subsring and Subshtring
  ⟨⟨Look for substring with one deletion⟩⟩
  for i ← 1 to n − 1
      if Subtring[Pre[i], Suf[i + 1]]
          return True
  ⟨⟨Look for substring with at most one replacement⟩⟩
  for i ← 1 to n − 2
      if Subtring[Pre[i], Suf[i + 2]]    ⟨⟨Yes i + 2, not i + 1⟩⟩
          return True
  ⟨⟨Look for substring with one insertion⟩⟩
  for i ← 1 to n − 2
      if Subshtring[Pre[i], Suf[i + 1]]
          return True
  return False
```

The entire algorithm runs in $\boldsymbol{O(m^2 + n)}$ ***time***.

I don't know if it's possible to reduce the $O(m^2)$ term to $O(m)$, but I wouldn't be surprised by some magic with suffix trees/arrays. ■

CS/Math 473 ✧ Fall 2022 Midterm 2 Solutions Problem 4	Name:

(a) Describe and analyze an efficient algorithm that either assigns each ghost a distinct house that they can haunt, or correctly reports that such an assignment is impossible.

Solution: We build a bipartite graph $G = (L \sqcup R, E)$, where L is the set of n ghosts, R is the set of m houses, and $ij \in E$ if and only if ghost i can haunt house j. Then we compute a maximum matching in the graph, as described in class, in $O(VE) = \mathbf{O((n+m)mn) = O(m^3)}$ ***time***. (If $n > m$, we can return FALSE immediately, so it's safe to assume otherwise.) Finally, if this matching has n edges, we return TRUE; otherwise, we return FALSE. ■

Rubric: 5 points = 2 for setting up the graph + 2 for maximum matching + 1 for running time as a function of m and n.

(b) Suppose every ghost is assigned to a distinct house that they cannot haunt. Describe and analyze an efficient algorithm to compute an exchange that results in a valid assignment of ghosts to houses. Assume $m = n$.

Solution (alternating cycle): We set up the same bipartite graph G as in part (a). Beetlejuice's assignment corresponds to a set of edges that are *missing* from G; call this set of non-edges B.

Compute a maximum matching M in G, in $O(VE) = O(n^3)$ time, as described in class.

Now consider the graph $M \cup B$. Every vertex in G' is incident to exactly one edge in M and exactly one edge in B. Thus, every vertex of $M \cup B$ has degree 2, which implies that $M \cup B$ consists entirely of disjoint simple cycles, which alternate between M and B.

Direct the vertices in B from ghost to house, and direct the vertices in M from house to ghost. Finally, for each ghost i, let $GiveTo[i]$ be the ghost vertex reached from vertex i by following two directed edges (first in B and second in M).

The overall algorithm runs in $\mathbf{O(n^3)}$ ***time***.

Without the assumption that $m = n$, we need to modify G by first throwing away any house vertex that does not appear in Beetlejuice's matching. The rest of the algorithm is unchanged; the modified algorithm runs in $O(mn + n^3)$ time. ■

Solution (matching in a smaller graph (when $m \neq n$)): We build a different bipartite graph $G = (L \sqcup R, E)$, where L and R both correspond to the set of n ghosts, and $ij \in E$ if and only if ghost i has been assigned a house that ghost j can haunt. We can construct this graph in $O(mn)$ time. We compute a maximum matching M in G in $O(VE) = O(n^3)$ time. Finally, for each edge $ij \in M$, we assign $GiveTo[i] \leftarrow j$. If the matching is perfect, every ghost gives their house to another ghost than can haunt it, so the exchange is valid; otherwise, no valid exchange is possible. The overall algorithm runs in $\mathbf{O(mn + n^3)}$ ***time***. ■

Rubric: 5 points = 2 for building $M \cup B$ + 2 for pointer-jumping + 1 for running time as a function of n. +1 extra credit if the algorithm is still correct when $m \neq n$.

CS/Math 473 ✧ Fall 2022 Midterm 2 Solutions Problem 5	Name:

(a) Describe an algorithm to simulate one roll of a fair 20-sided die using independent rolls of a fair 6-sided die *and no other source of randomness*.

(b) What is the *exact* expected number of 6-sided-die rolls executed by your algorithm?

(c) Derive an upper bound on the probability that your algorithm requires more than N rolls. Express your answer as a function of N.

(d) Estimate the smallest number N such that the probability that your algorithm requires more than N rolls is less than δ. Express your answer as a function of δ.

Solution (simple rejection sampling): Generate an integer z uniformly between 1 and 36 using two die rolls. If $z \le 20$, then return z; otherwise, start over.

```
RollD20():
  x ← Random(6)
  y ← Random(6)
  z ← 6(x − 1) + y
  ⟨⟨z is uniform in 1..36⟩⟩
  if z ≤ 20
      return z
  else
      RollD20()
```

Each trial uses two die rolls and succeeds with probability $20/36 = 5/9$. So the expected number of die rolls satisfies the equation

$$X = 2 + \frac{4}{9}X \implies X = \boxed{\frac{18}{5} = 3.6}.$$

Let P_N denote the probability that we need more than N rolls; we immediately have

$$P_N = \boxed{\left(\frac{4}{9}\right)^{\lfloor N/2 \rfloor}} = \begin{cases} (\frac{2}{3})^N & \text{if } N \text{ is even} \\ (\frac{2}{3})^{N-1} & \text{if } N \text{ is odd} \end{cases}$$

So to succeed in at most N rolls with probability at least $1 - \delta$, it suffices to set

$$\delta = \left(\frac{2}{3}\right)^N \implies N = \log_{2/3} \delta = \boxed{\log_{3/2} \frac{1}{\delta}} \approx 2.4663 \ln(1/\delta) = \Theta\left(\log \frac{1}{\delta}\right)$$

■

Rubric: 10 points = 4 for part (a) + 2 for part (b) + 2 for part (c) + 2 for part (d). +1 extra credit for an algorithm where the answer to part (b) is less than 3. Answers to (b), (c), and (d) must match the algorithm in part (a) to receive full credit. Only the expressions in boxes are required for full credit.

Solution (factored rejection sampling): Generate an integer x uniformly between 1 and 4 and another integer y uniformly between 1 and 5.

```
RollD20():
  repeat
    x ← Random(6)
  until x ≤ 4
  repeat
    y ← Random(6)
  until y ≤ 5
  return 5(x − 1) + y
```

Let X be the expected number of rolls in the first loop, and let Y be the expected number of rolls in the second loop. We immediately have

$$X = 1 + \frac{X}{3} \implies X = \frac{3}{2} \qquad\qquad Y = 1 + \frac{Y}{6} \implies Y = \frac{6}{5}$$

So the total expected number of rolls is $3/2 + 6/5 = \boxed{27/10 = 2.7}$.

Let P_N denote the probability that we need more than N rolls. If we need more than N rolls, then we need more than $N/2$ rolls in at least one of the two loops. It follows that

$$\boxed{P_N \le \frac{1}{3^{N/2}} + \frac{1}{6^{N/2}}}$$

So to succeed in at most N rolls with probability at least $1 - \delta$, it suffices to set

$$\delta = \frac{1}{3^{N/2}} \implies N = \boxed{2\log_3\left(\frac{1}{\delta}\right)} \approx 1.8204 \ln(1/\delta) = \Theta\left(\log \frac{1}{\delta}\right)$$

■

Solution (Optimal! $4 \cdot 20 = 5 \cdot 16 = 80$):

```
RollD20():
  z ← 6 · (Random(6) − 1) + Random(6)     ⟨⟨z is uniform in 1..36⟩⟩
  if z ≤ 20
      return z
  r ← z − 20                              ⟨⟨r is uniform in 1..16⟩⟩
  repeat forever:
      x ← Random(6)
      if x ≤ 5
          y ← 5(r − 1) + x                ⟨⟨y is uniform in 1..80⟩⟩
          return (y mod 20) + 1
```

Let X be the expected total number of rolls executed by this algorithm, and let Y be the expected number of rolls in the main repeat-forever loop. We immediately have

$$Y = 1 + \frac{Y}{6} \implies Y = \frac{6}{5}.$$

The main loop is executed with probability $16/36 = 4/9$, so

$$X = 2 + \frac{4}{9}Y = \boxed{\frac{38}{15} \approx 2.53333}.$$

Let P_N denote the probability that we need more than N rolls; we immediately have

$$\boxed{P_N = \begin{cases} 1 & \text{if } N = 1 \\ \frac{4}{9}\left(\frac{1}{6}\right)^{N-2} & \text{otherwise} \end{cases}}$$

Alternatively, we can observe that for any $N > 1$, there are 6^N possible outcomes for the first N rolls, and exactly 16 of those outcomes do not produce output: the 16 possible rolls of the first two dice that don't immediately terminate, followed by $N-2$ 6s. Thus,

$$\boxed{P_N = \begin{cases} 1 & \text{if } N = 1 \\ \frac{16}{6^N} & \text{otherwise} \end{cases}}$$

This algorithm is in fact optimal! For any $N > 1$, exactly $6^N \bmod 20 = 16$ outcomes from the first N rolls do not produce output, which is the least possible.

So to succeed in at most N rolls with probability at least $1 - \delta$, it suffices to set

$$\delta = \frac{16}{6^N} \implies N = \boxed{\log_6\left(\frac{16}{\delta}\right)} \approx 0.5581 \ln N + 1.5474 = \Theta\left(\log \frac{1}{\delta}\right)$$

■