1. (a) Prove that the following algorithm is not correct. [Hint: There is a one-line proof!]

\[
\text{RandomPermutation}(n):
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad \pi[i] \leftarrow i \\
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad \text{swap } \pi[i] \leftarrow \pi[\text{Random}(n)]
\end{align*}
\]

**Solution:** Here’s the one-line “proof” — \(3^3/3! = 27/6 = 9/2\) is not an integer!

Here are the output probabilities for each permutation when \(n = 3\):

\[
\begin{align*}
\Pr[1 \ 2 \ 3] &= 4/27 & \Pr[1 \ 3 \ 2] &= 5/27 & \Pr[2 \ 1 \ 3] &= 5/27 \\
\Pr[2 \ 3 \ 1] &= 5/27 & \Pr[3 \ 1 \ 2] &= 4/27 & \Pr[3 \ 2 \ 1] &= 4/27
\end{align*}
\]

More generally, there are exactly \(n^n\) possible outcomes for this algorithm, because there are \(n\) possible return values for each of the \(n\) calls to \(\text{Random}\). Each of these outcomes occurs with probability \(1/n^n\). If each permutation is equally likely, then exactly \(n^n/n!\) different outcomes lead to each permutation. But for any \(n \geq 3\), the number \(n^n/n!\) is not an integer—\(n!\) is divisible by \(n-1\), but \(n^n\) is not—so we have a contradiction. For any \(n \geq 3\), some permutations must be more likely than others. ■

(b) Consider the following implementation of \(\text{RandomPermutation}\).

\[
\text{RandomPermutation}(n):
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad \pi[i] \leftarrow \text{Null} \\
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad j \leftarrow \text{Random}(n) \\
&\quad \text{while } (\pi[j] \neq \text{Null}) \\
&\quad\quad j \leftarrow \text{Random}(n) \\
&\quad\quad \pi[j] \leftarrow i \\
&\text{return } \pi
\end{align*}
\]

Prove that this algorithm is correct and analyze its expected running time.

**Solution:** First let’s prove the algorithm correct. The first three lines inside the main loop choose an index \(j\) uniformly at random among the indices where \(\pi[j] = \text{Null}\). Thus, for purposes of correctness, we can rewrite the algorithm as follows:

\[
\text{RandomPermutation}(n):
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad \pi[i] \leftarrow \text{Null} \\
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad j \leftarrow \text{random index such that } \pi[j] = \text{Null} \\
&\quad \pi[j] \leftarrow i \\
&\text{return } \pi
\end{align*}
\]

For any permutation \(\sigma\) of \(\{1, 2, \ldots, n\}\), the algorithm outputs \(\sigma\) if and only if \(j = \sigma^{-1}(i)\) in the \(i\)th iteration of the main loop, for all \(i\). Thus, every permutation has non-zero probability of being the output.

The line in red can be rewritten further without changing the algorithm’s behavior.
RandomPermutation(n):
    for i ← 1 to n
        π[i] ← Null
    for i ← 1 to n
        k ← Random(n − i + 1)
        j ← the kth smallest index such that π[j] = Null
        π[j] ← i
    return π

There exactly \(n − i + 1\) possible outcomes for the \(i\)th iteration of the main loop, and therefore exactly \(n!\) possible outcomes for the entire algorithm. For any fixed permutation \(σ\), at least one of the \(n!\) outcomes produces \(σ\). But there are \(n!\) permutations, so each one must be produced by exactly one execution path.

To compute the expected running time, we refer back to the original formulation of the algorithm. For each index \(i\), let \(T_i\) denote the number of times that we call Random\((n)\) in the \(i\)th iteration of the main loop. Up to constant factors, the expected running time of the algorithm is \(\sum_i E[T_i]\).

Fix an index \(i\) and consider the \(i\)th iteration of the main loop. Each call to Random\((n)\) succeeds (meaning \(π[j] = Null\)) with probability \(1 − (i−1)/n\) and fails with probability \((i−1)/n\). If the call succeeds, the iteration ends; otherwise, we start the \(i\)th iteration over from scratch (or equivalently, we recurse). So linearity of expectation gives us

\[
E[T_i] = 1 + \frac{i−1}{n} E[T_i], \quad \text{which implies} \quad E[T_i] = \frac{n}{n−i+1}.
\]

It follows that the overall expected running time is

\[
\sum_{i=1}^{n} E[T_i] = \sum_{i=1}^{n} \frac{n}{n−i+1} = n \sum_{k=1}^{n} \frac{1}{k} = nH_n = \Theta(n \log n).
\]

(In the second step, we substitute \(k = n−i+1\).)

Rubric: This is not the only correct proof of correctness.

(c) Describe and analyze an implementation of RandomPermutation that runs in expected worst-case time \(O(n)\).

Solution: We modify the algorithm from part (a) in two different ways, which are essentially equivalent. The algorithm on the left, which was first proposed by Richard Durstenfeld in 1964, is usually called the Fisher-Yates shuffle or the Knuth shuffle.¹

¹Knuth popularized this algorithm in his seminal The Art of Computer Programming. The first edition of TAOCP attributed the algorithm to Lincoln Moses and Robert Oakford in 1963 and to Durstenfeld in 1964; second and later editions attribute the algorithm to Ronald Fisher and Frank Yates in 1938 (“in ordinary language”) and Durstenfeld (“in computer language”), and ignored Moses and Oakford.

But I believe credit for the algorithm should go solely to Durstenfeld. Fisher and Yates proposed a different and slower algorithm, similar to the algorithm at the top of this page, that runs in \(O(n^2)\) expected time. Moses and Oakford proposed a version of this algorithm where Random\((i)\) is implemener by repeatedly looking up the next number in a table of random integers between 1 and \(N\) (for some fixed \(N ≥ n\)) until the result is at most \(i\). The resulting algorithm, which runs in \(O(N \log n)\) expected time, is arguably closer to the coupon-collector algorithm from part (b) than to “Fisher-Yates”. Durstenfeld proposed his shuffle algorithm exactly in its modern linear-time form, in a 1/4-page article that appears to be his only scientific publication.

I have no idea who first realized that the algorithm on the right is equivalent.
These algorithms both clearly run in $O(n)$ worst-case time. The correctness of the first algorithm follows inductively from the observation that $\pi[n]$ is distributed uniformly at random, and the rest of the array is randomly permuted by the recursion fairy. The second algorithm is just the first algorithm run backward; the inverse of a random permutation is still a random permutation. Alternatively, both algorithms have exactly $n!$ different execution paths and output any fixed permutation with non-zero probability, so by the pigeonhole principle, each output permutation must be equally likely.

Solution: Let’s use radix sort!

The analysis of hashing collisions in the lecture notes implies that the elements of $X$ are distinct with high probability. Because each element of $X$ can be considered a three-digit number in base $n$, radix sort runs in $O(n)$ time. Once $X$ is sorted, we can easily check for duplicates in $O(n)$ time. Thus, the overall running time is $O(n)$ with high probability.

Solution: We modify the algorithm in part (b). Informally, we “hash” the integers 1 through $n$ into a table of size $2n$, and then compact the result down to an array of size $n$. 
Correctness follows from the usual counting argument. There are exactly \( (2n)!/n! \) possible contents for the array \( H \) after the second for-loop—\( n! \) permutations times \( \binom{2n}{n} \) choices of occupied cells. On the other hand, if we replace the while loop with “\( j \leftarrow \text{a random empty index} \)”, then the second for-loop has exactly \( (2n)!/n! \) possible outcomes. Thus, each possible array \( H \) is equally likely, which implies that each possible output permutation is equally likely.

This algorithm runs in \( O(n) \) expected time. Specifically, let \( X_i \) denote the number of calls to \( \text{RANDOM} \) during the \( i \)th iteration of the outer loop; the running time of the algorithm is clearly \( \Theta(n) + \Theta(\sum_i X_i) \). During the \( i \)th iteration of the outer loop, exactly \( i - 1 \) of the slots in the output array are already occupied. Thus, with probability \( (i - 1)/2n \), the while loop iterates more than once. Linearity of expectation implies the recurrence
\[
E[X_i] = 1 + \frac{i - 1}{2n} E[X_i],
\]
whose solution is \( E[X_i] = 2n/(2n - i + 1) \leq 2 \). Thus, the total expected number of calls to \( \text{RANDOM} \) is at most \( 2n \). There’s nothing special about 2 here; any constant bigger than 1 will do.

Rubric: These are not the only valid proofs of correctness for these algorithms. There are many other correct algorithms.
2. (a) Prove that any deterministic algorithm that computes the value of the root of a majority tree must examine every leaf.

**Solution:** We can use the following adversary strategy to assigns values to vertices of the tree (both leaves and internal nodes) as slowly as possible. Whenever the algorithm asks for the value of a leaf $\ell$, the adversary executes the following recursive procedure:

```
QUERY(\ell):
  if no sibling of $\ell$ has value 0
    value(\ell) ← 0
  else if no sibling of $\ell$ has value 1
    value(\ell) ← 1
  else if $\ell$ is the root
    value(\ell) ← 0
  else
    value(\ell) ← QUERY(parent(\ell))
  return value(\ell)
```

(Yep, it’s another algorithm.) This procedure assigns a value to any node only when values have been determined for all three of its children, and therefore (inductively) all of its descendants, and therefore in particular all of its leaves. On the other hand, if a value has not yet been assigned to a node, then at most one of its children has value 1 and at most one of its children has value 0, so (again inductively) the value could be either 0 or 1, depending on the values of the unassigned leaves.

(b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some explicit constant $c < 3$.

**Solution:** To determine the value of a node $v$, recursively evaluate two of its children, chosen uniformly at random. If and only if those two children has different values, recursively evaluate the third child. If all three children have the same value, the algorithm recurses only twice. Otherwise, there is a $1/3$ probability that the two children with the same value will be chosen first, in which case the algorithm will recurse only twice. Thus, the expected number of recursive calls is at most $8/3$, which implies that the expected running time $T(n)$ for a tree of depth $n$ obeys the recurrence

$$T(n) \leq \frac{8}{3} T(n-1).$$

We conclude that the expected running time is $O((8/3)^n)$.  

■
3. (a) Prove that for any heap-ordered binary trees $Q_1$ and $Q_2$ (not just those constructed by the operations listed above), the expected running time of $\text{Meld}(Q_1, Q_2)$ is $O(\log n)$, where $n = |Q_1| + |Q_2|$.

**Solution (exact):** Consider merging two arbitrary heap-ordered binary trees $Q_1$ and $Q_2$. The execution of the $\text{Meld}$ algorithm traces random paths downward from the roots of $Q_1$ and $Q_2$. For example, the path in $Q_1$ is extended downward, with equal probability either left or right, when the current node in $Q_1$ has lower priority than the current node in $Q_2$. Thus, the running time of $\text{Meld}(Q_1, Q_2)$ is at most $P(Q_1) + P(Q_2)$, where $P(T)$ denotes the length of a random root-to-leaf path in binary tree $T$.

To complete the analysis, we need to prove that $E[P(Q_1)] + E[P(Q_2)] = O(\log n)$, where $n$ is the total number of nodes in both trees. It suffices to prove that for every binary tree $T$ with $n$ nodes, we have $P(T) = O(\log n)$.

So let $P(n)$ denote the maximum, over all trees $T$ with $n$ nodes, of $E[P(T)]$. We claim that $P(n) \leq \log(n + 1)$. If $n = 0$, this claim is vacuously true, because there are no trees with at most 0 nodes. Otherwise, let $T$ be an arbitrary binary tree with $n$ nodes. Suppose the left and right subtrees of $T$ contain $\ell$ and $r$ nodes, respectively; observe that $\ell + r + 1 = n$. The inductive hypothesis implies that $E[P(\ell)] \leq \log(\ell + 1)$ and $E[P(r)] \leq \log(r + 1)$. Thus, the expected length of a random root-to-leaf path in $T$ is at most

\[
1 + \frac{1}{2} (P(\ell) + P(r)) \leq 1 + \frac{1}{2} (\log(\ell + 1) + \log(r + 1))
\]

\[
= \log\left(2\sqrt{\ell + 1}(r + 1)\right)
\]

\[
\leq \log\left((\ell + 1) + (r + 1)\right)
\]

\[
= \log(n + 1).
\]

The second-to-last step uses the inequality $2\sqrt{xy} \leq x + y$, which follows immediately from the fact that $(\sqrt{x} - \sqrt{y})^2 \geq 0$. This inequality is actually tight when $\ell = r = (n - 1)/2$. Somewhat counter-intuitively, it follows by induction that $E[P(T)]$ is maximized when $T$ is a perfectly balanced binary tree!

**Solution (up to constant factors):** Again, let $P(n) = \max_{|T| \leq n} E[P(T)]$, where the maximum is taken over all trees with at most $n$ nodes. As in the previous solution, it suffices to prove that $P(n) = O(\log n)$.

In any tree with at most $n$ nodes, at least one of the two subtrees of the root has at most $n/2$ nodes. In other words, after visiting the root, the random walk moves to a tree with at most $n/2$ nodes with probability 1/2, and to a tree with (crudely) at most $n$ nodes with probability 1/2. Thus, we have the recurrence

\[
P(n) \leq 1 + \frac{1}{2} P(n/2) + \frac{1}{2} P(n),
\]

or equivalently, $P(n) \leq 2 + P(n/2)$. The base case $P(1) = 1$ implies that $P(n) \leq 2[\log n] + 1 = O(\log n)$. Intuitively, we are bounding $P(n)$ by the expected number of independent fair coin flips required to get $\log n$ heads.

**Rubric:** Either solution is fine for full credit.
(b) Prove that Meld\((Q_1, Q_2)\) runs in \(O(\log n)\) time with high probability.

**Solution:** It suffices to prove that for any binary tree \(T\) with at most \(n\) nodes, we have \(P(T) = O(\log n)\) with high probability.

We build on our crude approximation of the expected root-to-leaf path length in terms of fair independent coin flips. The probability that \(4\lg n\) coin flips include less than \(\lg n\) heads is

\[
\frac{1}{2^{4\lg n}} \sum_{i=0}^{\lg n-1} \binom{4\lg n}{i} < \frac{1}{n^4} \left(\frac{4\lg n}{\lg n}\right)^{\lg n} < \frac{(4e)^{\lg n}}{n^4} = \frac{n^{2+\lg e} \lg n}{n^4} < \frac{1}{n}.
\]

The second inequality in this sequence uses the identity \(\binom{nk}{k} \leq (ea)^k\), which follows from Stirling's approximation. It follows that a random root-to-leaf path in an \(n\)-node binary tree has length at most \(4\lg n\) with high probability. ■

(c) Show that each of the other meldable priority queue operations can be implemented with at most one call to Meld and \(O(1)\) additional time.

**Solution:**

- **FindMin**\((Q)\): Return \(key(Q)\)
- **DeleteMin**\((Q)\): Return **Meld**\((left(Q), right(Q))\)
- **Insert**\((Q, x)\): Create a new one-node priority queue \(Q_x\) containing \(x\), and then return **Meld**\((Q, Q_x)\).
- **DecreaseKey**\((Q, x, y)\): Let \(Q_x\) be the node containing \(x\). Detach \(Q_x\) from its parent, set \(key(Q_x) \leftarrow y\), and return **Meld**\((Q, Q_x)\).
- **Delete**\((Q, x)\): Let \(Q_x\) be the the node containing \(x\). Replace \(Q_x\) with **DeleteMin**\((Q_x)\) (which calls **Meld**\((left(Q_x), right(Q_x))\)) and then return \(Q\). ■