1. (a) ***Prove*** that the following algorithm is ***not*** correct. *[Hint: Consider the case $n = 3$.]*

```
RandomPermutation(n):
  for i ← 1 to n
      π[i] ← i
  for i ← 1 to n
      swap π[i] ↔ π[Random(n)]
  return π
```

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 $n = 3$:

$$\Pr[1\ 2\ 3] = 4/27 \qquad \Pr[1\ 3\ 2] = 5/27 \qquad \Pr[2\ 1\ 3] = 5/27$$
$$\Pr[2\ 3\ 1] = 5/27 \qquad \Pr[3\ 1\ 2] = 4/27 \qquad \Pr[3\ 2\ 1] = 4/27$$

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 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. Thus, for any $n \geq 3$, some permutations must be more likely than others. ■

(b) Describe and analyze a correct RandomPermutation algorithm that runs in $O(n)$ expected time.

Solution: There are *lots* of correct solutions for this problem. Here are just a few.

```
RANDOMPERMUTATION(n):
  for i ← 1 to n
      π[i] ← i
  for i ← n down to 1
      j ← RANDOM(i)
      swap π[i] ↔ π[j]
  return π
```

```
RANDOMPERMUTATION(n):
  for i ← 1 to n
      π[i] ← i
  for i ← 1 to n
      j ← RANDOM(i)
      swap π[i] ↔ π[j]
  return π
```

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.

```
RANDOMPERMUTATION2(n):
  for j ← 1 to 2n
      H[j] ← EMPTY

  for i ← 1 to n
      j ← RANDOM(2n)
      while (H[j] ≠ EMPTY)
          j ← RANDOM(2n)
      H[j] ← i

  i ← 1
  for j ← 1 to 2n
      if H[j] ≠ EMPTY
          π[i] ← H[j]
          i ← i + 1
  return π
```

This algorithm runs in $O(n)$ expected time. Specifically, let X_i denote the number of calls to RANDOM during the ith iteration of the outer loop; the running time of the algorithm is clearly $\Theta(n) + \Theta(\sum_i X_i)$. During the ith 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. Since the random numbers produced by RANDOM are identically distributed, we have a recurrence

$$\mathrm{E}[X_i] \;=\; 1 + \frac{i-1}{2n}\,\mathrm{E}[X_i],$$

whose solution is $\mathrm{E}[X_i] = 2n/(2n-i+1) \le 2$. Thus, the total expected number of calls to RANDOM is at most $2n$, so the algorithm's expected running time is $\Theta(n)$. There's nothing special about the number 2 here; any constant bigger than 1 will do.

```
RandomPermutation3(n):
  for i ← 1 to n
      π[i] ← i
      X[i] ← Random(n^3)
  Sort X using base-n radix sort and permute π to match
  if any two elements of X are equal
      return RandomPermutation3(n)    ⟨⟨Try again recursively!⟩⟩
  else
      return π
```

This algorithm exploits the balls-and-bins analysis we did in class, which 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. ■

2. Describe an efficient 2-approximation algorithm for the candy-boxing problem. ***Prove*** that the approximation ratio of your algorithm is 2.

Solution: Here is the algorithm. Intuitively, we pack as much into the first box as possible, and then recurse on the remaining pieces of candy.

```
PACKCANDY(W[1..n]):
  sort W in decreasing order (*)
  b ← 1
  wtb ← 0
  for i ← 1 to n
      put candy i in box b
      wtb ← wtb + W[i]
      if w > L
          close box b
          b ← b + 1
          wtb ← 0
  return b − 1
```

Call a piece of candy *heavy* if it weighs more than L ounces, and *light* otherwise. Call a box *full* if it contains at least L ounces of candy.

Let H be the number of heavy pieces, and let OPT_L be the optimal solution for just the light pieces. If a box contains a heavy piece and something else, we can move the something else to a new empty box without decreasing the number of full boxes. Thus, there is an optimal packing of all the candy in which every heavy piece lies in its own box. In other words, $OPT = H + OPT_L$.

Let W_L denote the total weight of the light pieces; we clearly have $OPT_L \le W_L/L$.

Because we sorted the weights in the first line, PACKCANDY puts every heavy piece into its own box. Moreover, in the packing computed by PACKCANDY, any box containing light pieces has total weight less than $2L$. Thus, the total number of full boxes computed by PACKCANDY is at most $H + W_L/2L \ge H + OPT_L/2 > OPT/2$. ■

3. (a) Describe an efficient randomized approximation algorithm for Maximum-k-Cut, and ***prove*** that its expected approximation ratio is ~~at most~~ $(k-1)/k$.

Solution: Clearly OPT is at most the total weight of *all* edges.

Any edge e crosses the partition with probability $1-1/k$. Thus, by linearity of expectation, the total weight of all edges crossing the partition is exactly $1-1/k$ times the weight of all edges, which is at least $(1-1/k)OPT$. ■

(b) Now suppose we want to minimize the sum of the weights of edges that do *not* cross the partition. What expected approximation ratio does your algorithm from part (a) achieve for this new problem? ***Prove*** your answer is correct.

Solution: The expected approximation ratio is ***infinite*** in the worst case.

Let $k = 2$, and consider a graph with one edge of weight 1. The optimal solution has cost 0, but our algorithm computes a solution with expected cost $1/2$. ■

4. (a) ***Prove*** that the greedy change algorithm uses at most one coin of each denomination, when the denominations are powers of 2.

Solution: The greedy change algorithm computes the binary representation of the input number x; every bit in this representation is equal to 0 or 1.

We can also prove this claim by induction. The base case $x = 0$ is trivial. Suppose $2^k \le x < 2^{k-1}$. Then $x - 2^k < 2^k$, so the greedy algorithm uses exactly one 2^k-bit coin. The inductive hypothesis implies that the greedy algorithm uses at most one coin of each denomination less than 2^k to make $x - 2^k$ bits. ■

(b) ***Prove*** that the greedy change algorithm always returns the minimum number of coins with a given value, when the denominations are powers of 2.

Solution: Consider an optimal pile of coins with total value x, where $2^k \le x \le 2^{k+1}$. If the pile includes a 2^k-bit coin, then by the inductive hypothesis, the remaining coins are an optimal solution for $x - 2^k$. Otherwise, there must be at least two coins of the same value, because $\sum_{i=0}^{k-1} 2^i = 2^k - 1 < x$. We can replace two coins with the same value 2^i with a single coin of value 2^{i+1}, but this contradicts the optimality of the given pile of coins. ■

5. Let π be a random permutation of $\{1, 2, \ldots, n\}$, and let k be an arbitrary integer between 1 and n.

   (a) ***Prove*** that the probability that the number 1 lies in a cycle of length k in π is precisely $1/n$.

   Solution: Let $x_0 = 1$, and for any integer $k > 0$, let $x_k = \pi(x^{k-1})$. Let D_k denote the probability that integers $1, x_1, x_2, \ldots, x_{k-1}$ are all distinct, and let C_k denote the probability that 1 lies in a cycle of length k. Trivially $D_1 = 1$ and $C_1 = \Pr[\pi(1) = 1] = 1/n$.

   Suppose the numbers $1, x_1, x_2, \ldots, x_{k-1}$ are all distinct. Then x_k cannot equal x_j for any $2 \le j \le k-1$, but each of the other $n-k+1$ values between 1 and n is equally likely. Either $x_k = 1$ or $x_k \neq 1$. If $x_k = 1$, then 1 lies in a cycle of length k; this event happens with probability $1/(n-k+1)$. Otherwise, the numbers $1, x_1, x_2, \ldots, x_{k-1}$ are all distinct. Thus, we have the recurrences

   $$C_k = D_{k-1} \cdot \frac{1}{n-k+1} \qquad D_k = D_{k-1} \cdot \frac{n-k}{n-k+1}.$$

   Expanding the latter recurrence gives us

   $$D_k = \frac{n-k}{n-k+1} \cdot \frac{n-k-1}{n-k} \cdots \frac{n-1}{n} = \frac{n-k}{n}$$

   and therefore

   $$C_k = D_{k-1} \cdot \frac{1}{n-k+1} = \frac{n-k+1}{n} \cdot \frac{1}{n-k+1} = \frac{1}{n}$$

   ■

   (b) What is the *exact* expected length of the cycle in π that contains the number 1?

   Solution: Every cycle length between 1 and n is equally likely, so the expected cycle length is precisely $\boldsymbol{(n+1)/2}$. ■

   (c) What is the *exact* expected number of cycles of length k in π?

   Solution: Each element of $\{1, 2, \ldots, n\}$ is in a cycle of length k with probability $1/n$, so the expected number of elements in cycles of length k is 1. Thus, the expected number of cycles of length k is exactly $\boldsymbol{1/k}$. ■

   (d) What is the *exact* expected number of cycles in π?

   Solution: Let X_k denote the number of cycles of length k, and let $X = \sum_k X_k$ be the number of cycles overall. Then we have

   $$\mathrm{E}[X] = \sum_{k=1}^{n} \mathrm{E}[X_k] = \sum_{k=1}^{n} \frac{1}{k} = \boldsymbol{H_k}.$$

   ■