Problems marked with **E** are graded on effort, which means that they are graded subjectively on the perceived effort you put into them, rather than on correctness. For bonus questions, we will not provide any insight during office hours or Piazza, and we do not guarantee anything about the difficulty of these questions. We strongly encourage you to typeset your solutions in LaTeX.

If you collaborated with someone, you must state their names. You must write your own solution and may not look at any other student's write-up.

The following may be useful for this homework. The *geometric distribution* gives the probability that in a sequence of independent trials, each with success probability p, the first success occurs on the kth trial.

$$\Pr[\text{The } k\text{th trial is the first success}] = p(1-p)^{k-1}.$$

Using this, we can find the expected trial number of the *first* success,

$$\text{Ex}[\text{trial number of the first success}] = \sum_{k=1}^{\infty} kp(1-p)^{k-1} = \frac{1}{p}.$$

Proving the above equality takes a bit of work (but not too much). As an exercise try to prove it, but you may use it as a fact in your homework solutions.

0. If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

1. A group of n students, all of whom have distinct heights, line up in a single-file line uniformly at random to get a group picture taken. If a student has any students in front of them who is taller than them, then they will not be seen in the picture. For this reason, every student files one complaint to the photographer for each taller student who is in front of them since each one of these students would individually block the original student from being seen. Compute the expected number of complaints that the photographer will receive.

Solution: Let X_{ij} be an indicator random variable where:

$$X_{ij} = \begin{cases} 1 & \text{if } i < j \text{ and the student in position } i \text{ is taller than student in position } j \\ 0 & \text{otherwise.} \end{cases}$$

Note that for convention, we assume position 1 to be the front of the line and position n to be the end of the line. The number of complaints that will be filed is exactly equal to the number of X_{ij}s that equal 1.

For any $i < j$, is it equally likely that the student in position i is taller than the student in position j as it is that the student in position i is shorter than the student in position j. One explanation for this is that the students lined up uniformly at random. Another explanation is that for any ordering of students for which the student in position i is taller than the student in position j, there is exactly one other ordering for which the student in position i is shorter than the student in position j and all other students are in the exact same position (just swap these two students). Now we compute:

$$\begin{aligned}
\text{Ex}\left[\sum_{1\le i<j\le n} X_{ij}\right] &= \sum_{1\le i<j\le n} \text{Ex}[X_{ij}] \\
&= \sum_{1\le i<j\le n} \frac{1}{2} \\
&= \binom{n}{2} \cdot \frac{1}{2} \\
&= \frac{n\cdot(n-1)}{4}
\end{aligned}$$

2. In order to generate random integers, operating systems often use real-world events as a source of randomness (e.g., mouse movements, atmospheric noise, thermal fluctuations, even the radioactive decay of certain isotopes). Unfortunately, these events may not be uniformly random, so they can cause bias in the integers we generate from them. This presents a problem since most randomized algorithms rely on access to unbiased, uniformly random integers. Generating uniformly random integers from a stream of arbitrarily biased integers is sometimes called *decorrelation*. Decorrelation is especially important to fields such as signal processing, data compression, and cryptography (`https://en.wikipedia.org/wiki/Decorrelation`).

   (a) Complete line 6 and line 7 in the following Las Vegas algorithm. GET-UNBIASED-INT returns an integer uniformly at random from $\{1, 2, ..., n\}$ using calls to GET-BIASED-INT as a source of potentially biased but independent integers.
   GET-BIASED-INT is a function that returns an integer in $\{1, 2, ..., n\}$.
   For each $1 \le i \le n$, the probability of GET-BIASED-INT returning i is p_i, with each $p_i > 0$ and where $\sum_{i=1}^{n} p_i = 1$.

```
1: function GET-UNBIASED-INT(n)
2:     A = [ ]
3:     loop
4:         for k from 1 to n do
5:             Add GET-BIASED-INT(n) to the back of A
6:         if _____ then
7:             return _____
8:         else
9:             A = [ ]
```

Solution:

```
1: function GET-UNBIASED-INT(n)
2:     A = [ ]
3:     loop
4:         for k from 1 to n do
5:             Add GET-BIASED-INT(n) to the back of A
6:         if Each a ∈ A is distinct then
7:             return A[0]
8:         else
9:             A = [ ]
```

(b) Argue why the proposed algorithm outputs each integer in $\{1, 2, ..., n\}$ with equal probability.

Solution: The algorithm will only return once A is a permutation of the numbers between 1 and n. Each of the $n!$ such permutations has identical probability of being generated. Therefore, the probability that a given $1 \leq i \leq n$ is returned from the algorithm is equal to the probability that i is the first number in the permutation. Because there are exactly $(n-1)!$ permutations beginning with i for each i, then each element in $\{1, 2, ..., n\}$ has equal probability of being returned.

(c) Assuming that each call to GET-BIASED-INT takes $O(1)$ time, determine the expected runtime of GET-UNBIASED-INT as a function of the probability assignment, $P = (p_1, p_2, \ldots, p_n)$.

Solution: Each iteration of the loop will make n calls to GET-BIASED-INT which takes $O(1)$ time for each call, thus taking $O(n)$ time in the for loop. Also, it takes $O(n)$ time to check if the elements inside A are distinct if implemented using a hash table. Therefore, it takes a combined $O(n) + O(n) = O(n)$ time for each iteration of the loop. Alternatively, checking that all elements are distinct could be achieved by first sorting the elements in the array, then iterating over the array to check if any consecutive elements are equal. This would take $O(n \cdot log(n)) + O(n) = O(n \cdot log(n))$ time using merge sort or quicksort, but for the rest of this solution, we will assume a hash table implementation. Now we must determine the expected number of times that the main loop will run until the function returns.

In order for the function to return, each of the n elements in A must be distinct. Each individual permutation of the n numbers will occur with probability $\prod_{i=1}^{n} p_i$ because for each integer i, the probability of GET-BIASED-INT returning i is p_i. Furthermore, there are $n!$ permutations. Therefore, the probability that an arbitrary run of the main loop causes the function to return is $n! \cdot \prod_{i=1}^{n} p_i$.

Finally, by the given formula for Ex[trial number of the first success], the expected number of iterations of the main loop is $\frac{1}{n! \cdot \prod_{i=1}^{n} p_i}$ and each iteration takes $O(n)$ time. Therefore, the expected overall runtime is $O(n \cdot \frac{1}{n! \cdot \prod_{i=1}^{n} p_i})$.

(d) *Optional bonus:* GET-BIASED-BIT is also a source of potentially biased but independent outputs and is a function that returns either 0 or 1. Again, the probability of each output is fixed, but unknown. Using calls to GET-BIASED-BIT, construct a Las Vegas algorithm, GET-WEIGHTED-BIT, that takes in two natural number inputs, x and y, and outputs 0 with probability $\frac{x}{x+y}$ and 1 with probability $\frac{y}{x+y}$.
Hint: Think how you could accomplish this on small inputs such as 2 and 3.

Solution: Expanding on the central idea from part (a), it is necessary to find multiple different output sequences of GET-BIASED-BIT function calls that each have equal probability. This is useful because although the probability of either 0 or 1 being returned from a single call to GET-BIASED-BIT is unknown, the probability of different bit sequences will be the same as long as there are an equal number of 0s and 1s in the sequence. For example, the probability of seeing the sequence 0011 is equally likely as seeing 1001 when calling GET-BIASED-BIT 4 times.

With this in mind, after the ratio $x : y$ is fully simplified to $x' : y'$, we could then randomly select from among $x' + y'$ bit strings of equal length and equal numbers of 0s and 1s and return 0 for the first x' of them and 1 for the next y' of them. For any particular bit string length, n, the number of bit strings with the same number of 0s, say k, is maximized when $k = \frac{n}{2}$ when n is even, or $k = \frac{n+1}{2}$ when n is odd. For example, if $n = 4$, then $\binom{4}{k}$ (i.e. the number of bit strings of length 4 with k 0s) is maximized when $k = 2$. From these observations, we reach the following algorithm.

```
1:  function GET-WEIGHTED-BIT(x, y)
2:      Set x' : y' equal to the fully simplified ratio x : y
3:      n = smallest n such that (n choose ⌈n/2⌉) ≥ x' + y'
4:      A = [ ]
5:      loop
6:          for i from 1 to n do
7:              Add GET-BIASED-BIT() to the back of A
8:          if A is one of the first x' bit sequences with ⌈n/2⌉ 0s then
9:              return 0
10:         else if A is one of the x' + 1^th, ..., x' + y'^th sequences with ⌈n/2⌉ 0s then
11:             return 1
12:         else
13:             A = [ ]
```

As an example, if the algorithm was given inputs $x = 2$ and $y = 3$, then $x' = 2$, $y' = 3$, and $n = 4$ since $\binom{4}{2} = 6 \geq 2 + 3$. Next, we continue computing random bit sequences of length 4 until we get one of the sequences $\{0011, 0101, 0110, 1001, 1010\}$ and output 0 if the sequence is any of $\{0011, 0101\}$ and output 1 if the sequence is any of $\{0110, 1001, 1010\}$. In this way, the interpretation of 'first' in the algorithm is with respect to the natural ordering of the binary numbers associated with each binary string.

3**E** (a) Lecture 19 covered Fermat's primality testing. Run Fermat's primality test on each of the following numbers, using the provided list of numbers as the "random" choices of a. (You might not always need to use the entire list before being able to output a decision.)

i. Test 45 with $a = 14, 32, 7$.

ii. Test 73 with $a = 38, 15, 21$.

iii. Test 561 with $a = 50, 164, 403$.

Note: Feel free to use a computing tool to make the calculations easier, e.g., `https://www.mtholyoke.edu/courses/quenell/s2003/ma139/js/powermod.html`

Solution:

(i) $14^{44} \equiv 16 \pmod{45}$ so we conclude that 45 is (definitely) composite.

(ii) $38^{72} \equiv 1 \pmod{73}$, so 73 may be prime
$15^{72} \equiv 1 \pmod{73}$, so 73 may be prime
$21^{72} \equiv 1 \pmod{73}$, so 73 may be prime
We (correctly!) declare that 73 is prime.

(iii) $50^{560} \equiv 1 \pmod{561}$, so 561 may be prime
$164^{560} \equiv 1 \pmod{561}$, so 561 may be prime
$403^{560} \equiv 1 \pmod{561}$, so 561 may be prime
We (incorrectly!) declare that 561 is prime.

(b) Suppose we get a result that a number is prime, but want to be at least 95% sure that our result is correct. Also, assume that for a given composite number, a single iteration of Fermat's primality test has probability at least $\frac{1}{2}$ (over the choice of a) of declaring that the number is composite. How many independent tests on the number must be conducted until we have met our desired degree of confidence?

Solution: In order to be at least 95% confident, there must be at most a 5% chance of error in the result. Given that each iteration of Fermat's primality test is independent, for a given composite number, there is at most a $(\frac{1}{2})^n$ chance that the number is composite after n iterations. We can then solve for n to find that the minimum n for which the probability of error is lower than 5% is 5.

4. The randomized algorithm Randomized-Quicksort (which is Quicksort with pivots chosen at random) has worst-case running time $\Theta(n^2)$, but expected running time $O(n \log n)$ when run on any array of n elements. In practice, well-implemented Quicksort is faster than all the other $O(n \log n)$-time sorting algorithms on "generic" data.

Prove that for any array A of n elements, and any constant $c > 0$,

$$\lim_{n \to \infty} \Pr[\text{Randomized-Quicksort on input } A \text{ takes at least } cn^2 \text{ steps}] = 0.$$

That is, Randomized-Quicksort is very unlikely to run in $\Omega(n^2)$ time.

Hint: The number of steps a randomized algorithm takes is a random variable. You do not need to know anything about how Quicksort or Randomized-Quicksort work; just use the above facts and Markov's inequality.

Solution: We want to show that for any constant $c > 0$, the probability that Randomized-Quicksort takes at least cn^2 steps approaches 0 as $n \to \infty$. Fix an arbitrary constant $c > 0$. For any fixed array A of length n, let T be a random variable denoting the number of steps the algorithm takes. We know that Randomized-Quicksort takes $O(n \log n)$ steps in expectation, i.e., $\text{Ex}(T) \leq Cn \log n$ for some constant C (for all large enough n). Then we use Markov's inequality (which is valid because T is non-negative):

$$\Pr[T \geq cn^2] \leq \frac{\text{Ex}[T]}{cn^2} \leq \frac{Cn \log n}{cn^2} = \frac{C}{c}\frac{\log n}{n}.$$

Notice that C/c is a constant. So, taking $n \to \infty$ yields

$$\lim_{n\to\infty} \Pr[T \geq cn^2] \leq \lim_{n\to\infty} \frac{C}{c}\frac{\log n}{n} = 0.$$

By definition of probability, we have $\lim_{n\to\infty} \Pr[T \geq cn^2] \geq 0$, so combining that with the above yields

$$\lim_{n\to\infty} \Pr[T \geq cn^2] = 0$$

as desired.

5. Here is a randomized algorithm for producing a cut in a graph $G = (V, E)$:

```
1. Initialize S to be the empty set.
2. For each vertex v in V:
   Put v into S with probability 1/2.
3. Output (S, T = V \ S).
```

Recall that, for an undirected graph, $E(S, T)$ is the set of edges "crossing" the cut (S, T), i.e., those that have one endpoint in S and the other endpoint in T. The size of the cut is $|E(S, T)|$.

(a) Prove that the above algorithm is a $\frac{1}{2}$-approximation *in expectation* for Max-Cut on undirected graphs. That is, the expected size of the output cut is at least half the size of a maximum cut.

Hint: use linearity of expectation to show that in expectation, half the edges of the graph are in $E(S, T)$.

Solution: Let $E = \{e_1, e_2, \ldots, e_m\}$ where $m = |E|$, and suppose without loss of generality that there are no self-loop edges. (This is without loss of generality because such edges cannot cross any cut, so to get a $\frac{1}{2}$ approximation it suffices to cut at least half of the non-self-loop edges.) For each $i = 1, \ldots, m$, let E_i be an indicator random variable that is 1 if $e_i \in E(S, T)$ and 0 otherwise. Consider an arbitrary edge $e_i = (u, v)$. Then because E_i is an indicator variable,

$$\text{Ex}[E_i] = \Pr[E_i = 1]$$

$$= \Pr[u \in S \wedge v \in T] + \Pr[u \in T \wedge v \in S].$$

Because u and v are placed in their respective sets *independently* (they are distinct vertices), the above is

$$\begin{aligned} &= \Pr[u \in S]\Pr[v \in T] + \Pr[u \in T]\Pr[v \in S] \\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \\ &= \frac{1}{2}. \end{aligned}$$

Now we can analyze the expected size of the cut. By definition of the E_i and linearity of expectation,

$$\mathrm{Ex}[|E(S,T)|] = \mathrm{Ex}\Big[\sum_{i=1}^{m} E_i\Big] = \sum_{i=1}^{m} \mathrm{Ex}[E_i] = \sum_{i=1}^{m} \frac{1}{2} = \frac{m}{2}.$$

Since no cut can be larger than the total number of edges, we know that $\mathrm{OPT} \leq m$. Thus,

$$\mathrm{Ex}[|E(S,T)|] = \frac{|E|}{2} \geq \frac{\mathrm{OPT}}{2}.$$

Therefore, this algorithm gives a $\frac{1}{2}$-approximation, in expectation, to undirected Max-Cut.

(b) Argue that, for any undirected graph $G = (V, E)$, there exists a cut of size at least $\frac{|E|}{2}$.

Solution: Fix a graph $G = (V, E)$. The cut obtained by running the above algorithm on G is of size $\frac{|E|}{2}$ in expectation. In other words, in the space of all possible cuts the algorithm could output when run on G, the average size is $\frac{|E|}{2}$. The largest cut the algorithm could output cannot be smaller than the average cut, so the probability that the algorithm outputs a cut of size at least $\frac{|E|}{2}$ is nonzero. Thus, such a cut must exist in G.

6. Recall that a skip list is a linked list with multiple levels. When an item is added, the following procedure is used to potentially place the item in multiple levels:

```
1. Find where to insert the element at the lowest level.
2. Insert the item at the current level.
3. Flip a fair coin:
   i. If the result is heads, move up one level and continue from step 2.
  ii. Otherwise stop.
```

Suppose we add n items to an initially empty skip list (and delete none of them). Let random variable Z_i be the number of items at level i (where level 0 is the lowest level), and let random variable h be the *height* of the skip list, i.e., the highest non-empty level.

(a) Show that the expected value of Z_i is $\text{Ex}[Z_i] = n/2^i$.

Solution: Let X_{ij} be an indicator random variable where:

$$X_{ij} = \begin{cases} 1 & \text{if element } j \text{ appears on level } i \\ 0 & \text{otherwise.} \end{cases}$$

Then $Z_i = \sum_{j=1}^{n} X_{ij}$, because the summation counts the number of elements appearing on level i. So by linearity of expectation we have

$$\begin{aligned} \text{Ex}[Z_i] &= \text{Ex}\Big[\sum_{j=1}^{n} X_{ij}\Big] \\ &= \sum_{j=1}^{n} \text{Ex}[X_{ij}] \\ &= \sum_{j=1}^{n} \Pr[X_{ij} = 1]. \end{aligned}$$

In order for the jth element to be on level i, when inserting the element we must have flipped i heads in a row (the remaining coin flip(s) are irrelevant). This occurs with probability $1/2^i$, so $\Pr[X_{ij} = 1] = 1/2^i$. Putting everything together, we have

$$\text{Ex}[Z_i] = \sum_{j=1}^{n} \frac{1}{2^i} = \frac{n}{2^i}.$$

(b) *Optional bonus:* Show that the expected value of the height is $\text{Ex}[h] = O(\log n)$.

Solution: Let X_{ij} be defined as in (a) and let Y_i be a random variable where

$$Y_i = \begin{cases} 1 & \text{if level } i \text{ is nonempty} \\ 0 & \text{otherwise.} \end{cases}$$

Notice that

$$h = \sum_{i=1}^{\infty} Y_i,$$

and so by linearity of expectation

$$\text{Ex}[h] = \sum_{i=1}^{\infty} \text{Ex}[Y_i].$$

Now,

$$\text{Ex}[Y_i] = \Pr[Y_i = 1]$$

$$= \Pr[(X_{i1} = 1) \vee \ldots \vee (X_{in} = 1)]$$

which by the union bound is

$$\begin{aligned} &\leq \sum_{j=1}^{n} \Pr[X_{ij} = 1] \\ &= \sum_{j=1}^{n} \frac{1}{2^i} \\ &= \frac{n}{2^i}. \end{aligned}$$

Because $\text{Ex}[Y_i] \leq 1$, we now have two upper bounds on $\text{Ex}[Y_i]$. Notice that 1 is the tighter bound when $2^i < n$ and $\frac{n}{2^i}$ is tightest once $2^i \geq n$. So, we break the expression for $\text{Ex}[h]$ into two parts, using a tightest bound for each part:

$$\begin{aligned} \text{Ex}[h] &= \sum_{i=1}^{\infty} \text{Ex}[Y_i] \\ &= \sum_{i=1}^{\lfloor \log n \rfloor} \text{Ex}[Y_i] + \sum_{i=\lfloor \log n \rfloor + 1}^{\infty} \text{Ex}[Y_i] \\ &\leq \sum_{i=1}^{\lfloor \log n \rfloor} 1 + \sum_{i=\lfloor \log n \rfloor + 1}^{\infty} \frac{n}{2^i} \\ &\leq \lfloor \log n \rfloor + \sum_{j=0}^{\infty} \frac{1}{2^j} \\ &= \lfloor \log n \rfloor + 2 \\ &= O(\log n). \end{aligned}$$

(c) Just like with a balanced binary search tree, we want the height of a skip list to be small. Although the height is expected to be small (see the previous part), there is a chance that it can end up being much larger than its expectation. Here you will show that the probability of this happening becomes small as n grows large. Specifically, prove that

$$\lim_{n \to \infty} \Pr[h \geq (\log n)^3] = 0.$$

You may use part (b) for your answer, regardless of whether you solved (b).

Solution: Using $\text{Ex}[h] = O(\log n)$ and Markov's inequality (which is valid because h is non-negative), we have

$$\Pr[h \geq (\log n)^3] \leq \frac{\text{Ex}[h]}{(\log n)^3} \leq \frac{c \log n}{(\log n)^3} = \frac{c}{(\log n)^2}$$

for some constant c. As n goes to infinity, this quantity approaches 0 because $(\log n)^2$ grows without bound.

(d) *Optional bonus:* Prove part (c) *without* using part (b).

Solution: Here is an argument that does not rely on part (b), and gives a much stronger bound on the probability. Consider the probability that some particular element appears on level $(\log n)^3$. In order for this to occur, we must flip heads $(\log n)^3$ times in a row when inserting that element; this occurs with probability $1/2^{(\log n)^3} = 1/n^{(\log n)^2}$. By the union bound, the probability that *at least one* of the n elements appears on level $\log^3 n$ is therefore at most $n/n^{(\log n)^2} = 1/n^{(\log n)^2-1}$. This quantity approaches zero (very quickly) as n goes to infinity.