1. **Some random hotel.** (2 points) There's a hotel with 100 rooms, each belonging to one of 100 guests. After an evening soiree, all of the guests (rather inebriated) randomly selects a room to sleep in for that night. Multiple guests might end up in the same room.

   (a) (1 point) What is the expected number of guests that end up returning to their own hotel room?

   [We are expecting a number. Please show your work.]

   (b) (1 point) What is the expected number of guests that end up in a room with exactly one other person?

   [We are expecting a mathematical expression like $(6)(8)$ **or a number like** 48**. Please show your work.]**

SOLUTION:

(a) Let

$$X_i = \begin{cases} 1 & \text{if guest } i \text{ returns to their own room} \\ 0 & \text{otherwise} \end{cases}$$

The quantity of interest is

$$E[X] = E[\sum_{i=1}^{100} X_i] = \sum_{i=1}^{100} E[X_i]$$

by linearity of expectation.

$$E[X_i] = P(\text{guest } i \text{ returns to their own hotel room})$$

This is $1/100$, so $E[X] = \sum_{i=1}^{100} E[X_i] = \sum_{i=1}^{100}(1/100) = 1$.

(b) Here it is easier to count by rooms than by guests. We must find the expected number of rooms that have exactly 2 guests (and then multiply that number by 2). The probability that room j has 2 guests is

$$\binom{100}{2} \cdot \left(\frac{1}{100}\right)^2 \cdot \left(\frac{99}{100}\right)^{98}$$

By the logic of part (a), the expected number of rooms that have 2 guests is

$$\binom{100}{2} \cdot \frac{1}{100} \cdot \left(\frac{99}{100}\right)^{98}$$

which is

$$\frac{99}{2} \cdot \left(\frac{99}{100}\right)^{98}$$

So the expected number of guests that are in a room with exactly two people is

$$99 \cdot \left(\frac{99}{100}\right)^{98}$$

2. **An exhausting set of hash functions.** (6 points) In this problem, we'll investigate the definition of universal hash functions. Let $\mathcal{U}$ denote a universe of size M, and let n be the number of buckets in a hash table.

   (a) (3 points) Let $\mathcal{H}$ be the family of all possible functions mapping from $\mathcal{U}$ to $\{1, ..., n\}$. Prove that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h randomly chosen from $\mathcal{H}$,

   $$\Pr[h(x_i) = h(x_j)] = \frac{1}{n}.$$

   This shows that $\mathcal{H}$ is a universal hash family, and moreover that we have *equality* in the definition of the universal hash family property, not just a $\leq$ relationship.

   [We are expecting: a careful and rigorous proof, though it should not need to be more than one or two paragraphs. You should be especially careful about what is random and what is not.]

   (b) (3 points) There also exist hash families such that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h randomly drawn from the family, $\Pr[h(x_i) = h(x_j)] < \frac{1}{n}$. (Notice that the inequality here is strict!) We will now explore one such family. Consider $\mathcal{H}' = \mathcal{H} \setminus \{h_1\}$, where h_1 is the function defined by

   $$h_1(x_i) = 1 \qquad \text{for all} \qquad x_i \in \mathcal{U}.$$

   That is, $\mathcal{H}'$ is the family of all functions from $\mathcal{U}$ to $\{1, ..., n\}$ *except* for the function h_1 which sends all elements of $\mathcal{U}$ to 1.

   Prove that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h drawn randomly from $\mathcal{H}'$, we have

   $$\Pr[h(x_i) = h(x_j)] < \frac{1}{n}.$$

   [We are expecting: a clear and rigorous proof. As above, this needn't be more than a paragraph of two, but you should be very careful about what is random and what is not.]

   (c) (1 point extra credit) Give a hash family $\mathcal{H}$ so that

   $$\max_{x \neq y \in \mathcal{U}} \Pr[h(x) = h(y)]$$

   is as small as possible (and ideally prove that it is as small as possible). What is this probability? Above, the probability is over the choice of a uniformly random $h \in \mathcal{H}$.

   [We are expecting: we aren't expecting anything, this is a bonus problem.]

Solution:

(a) We give several solutions below.

Solution 1:
We can break up the probability into the sum of conditional probabilities based on the value of $h(x_i)$. That is, $\Pr[h(x_i) = h(x_j)] = \sum_{k=1}^{n} \Pr[h(x_i) = k] \cdot \Pr[h(x_i) = h(x_j) | h(x_i) = k]$. But what is $\Pr[h(x_i) = h(x_j) | h(x_i) = k]$ exactly? It is in fact just $\Pr[h(x_j) = k]$, since once we know that $h(x_i) = k$, the only way for $h(x_j)$ to equal $h(x_i)$ is if $h(x_j) = k$. Then $\Pr[h(x_i) = h(x_j)] = \sum_{k=1}^{n} \Pr[h(x_i) = k] \cdot \Pr[h(x_j) = k]$.
Now, because $\mathcal{H}$ consists of every possible function from $\mathcal{U}$ to $\{1, ..., n\}$, we know that there are equally many functions mapping x_i to every possible value in $\{1, ..., n\}$. Then $\Pr[h(x_i) = k]$ is exactly $1/n$, and similarly for $\Pr[h(x_j) = k]$. This tells us that $\Pr[h(x_i) = h(y_i)] = \sum_{k=1}^{n} \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n}$, since we are summing $(\frac{1}{n})^2$ up n times. This proves that indeed the probability of collision is indeed exactly $1/n$, as desired.

Solution 2:
The number of hash functions so that $h(x_i) = h(x_j) = k$ is n^{M-2} for each $k = 1, \ldots, n$, because there are $M - 2$ remaining elements of $\mathcal{U}$ that can be assigned. So the total number of hash functions so that $h(x_i) = h(x_j)$ is exactly n^{M-1}. Thus the probability that $h(x_i) = h(x_j)$ is exactly

$$\frac{n^{M-1}}{|\mathcal{H}|} = \frac{n^{M-1}}{n^M} = \frac{1}{n}.$$

Note: We will also accept correct solutions that start from the fact that a uniformly random function $h : \mathcal{U} \to \{1, \ldots, n\}$ has that $\{h(x_i) \mid x_i \in \mathcal{U}\}$ is a collection M of i.i.d. uniformly random variables in $\{1, \ldots, n\}$.

(b) One way to see that this is the case is to notice that, by removing h_1 from the set of possible functions, we are making sure that every pair of elements is strictly less likely to collide. That is, in part (a), the probability we got was based on the existence of this function in the set, and removing the function must make the probability of collision strictly lower.
We can also (and should, to get full credit!) prove this more formally. Fix $x_i, x_j \in \mathcal{U}$. We will count the number of $h \in \mathcal{H}$ that have $h(x_i) = h(x_j)$.
First, observe that for all $k \in \{2, \ldots, n\}$, the number of h so that $h(x_i) = h(x_j) = k$ is exactly n^{M-2}. That is, for each of the remaining $M - 2$ items of $\mathcal{U}$, we have n choices each. The fact that $\mathcal{H}'$ is missing h_1 does not affect this computation, since $h(x_i) = k \neq 1$, so $h \neq h_1$ in this case.
On the other hand, for $k = 1$, the number of h so that $h(x_i) = h(x_j) = 1$ is exactly $n^{M-2} - 1$. The reasoning is exactly the same as before, except we throw h_1 out of the count.

Then, the probability

$$\begin{aligned}\Pr h(x_i) = h(x_j) &= \frac{\text{number of } h \in \mathcal{H}' \text{ so that } h(x_i) = h(x_j)}{|\mathcal{H}'|} \\ &= \frac{n^{M-2} - 1}{n^M - 1} + \sum_{k \neq 1} \frac{n^{M-2}}{n^M - 1} \\ &= \frac{n^{M-1} - 1}{n^M - 1} \\ &< \frac{1}{n}.\end{aligned}$$

(c) (Bonus problem). [Note: this solution is less rigorous than normal because the problem is a bonus problem. Please ask if you want more details on any step!]

Let H be any hash family, and draw h from H. Suppose that $n|M$ for simplicity. First, we compute

$$\begin{aligned}\sum_{x,y \in \mathcal{U}} \Pr(h(x) = h(y)) &= E \sum_{x,y \in \mathcal{U}} \mathbf{1}(h(x) = h(y)) \\ &= E \sum_{j=1}^{n} |B_{j,h}|^2\end{aligned}$$

where

$$B_{j,h} = \{x \in \mathcal{U} \mid h(x) = j\},$$

and where the probability and expectation are both over h. Breaking up the first sum into pairs x, y where $x = y$ and pairs where they are not equal, we have

$$M + \sum_{x \neq y} \Pr(h(x) = h(y)) = E \sum_{j=1}^{n} |B_{j,h}|^2,$$

hence

$$\max_{x \neq y} \Pr(h(x) = h(y)) \geq \frac{E\left[\sum_{j=1}^{n} |B_{j,h}|^2\right] - M}{M(M-1)}.$$

Now, the sum in the right hand side is minimized when all of the buckets $B_{j,h}$ are the same size; that is, size M/n. (This follows from Cauchy-Shwartz). Plugging this in, we compute

$$\max_{x \neq y} \Pr(h(x) = h(y)) \geq \frac{M/n - 1}{M - 1}.$$

The reasoning above also shows that this is tight: if we choose $\mathcal{H}$ to be the set of functions that are "balanced" (that is, $\mathcal{H}$ is the set of all functions $h : U \to \{1, \ldots, n\}$ so that $|B_{j,h}| = M/n$ for all j), then all of the inequalities above are actually equalities.

3. Suppose that on your computer you have stored n password-protected files, each with a unique password. You've written down all of these n passwords, but you do not know which password unlocks which file. You've put these files into an array F and their passwords into an array P in an arbitrary order (so $P[i]$ does not necessarily unlock $F[i]$). If you test password $P[i]$ on file $F[j]$, one of three things will happen:

   1) $P[i]$ unlocks $F[j]$
   2) The computer tells you that $P[i]$ is lexicographically smaller than $F[j]$'s true password
   3) The computer tells you that $P[i]$ is lexicographically greater than $F[j]$'s true password

   You **cannot** test whether a password is lexicographically smaller or greater than another password, and you **cannot** test whether a file's password is lexicographically smaller or greater than another file's password.

   (a) (3pts) Design an randomized algorithm to match each file to its password, which runs in expected runtime $O(n \log(n))$. [**We are expecting: pseudocode and/or an English description of an algorithm.**]

   (b) (2pts) Explain why your algorithm is correct. [**We are expecting: an informal argument (a paragraph or so) about why your algorithm is correct, which is enough to convince the reader/grader. You may also submit a formal proof if you prefer.**]

   (c) (2pts) Analyze the running time of your algorithm, and show that it runs in expected runtime $O(n \log(n))$. [**We are expecting: a formal analysis of the runtime.**]

   SOLUTION.

   (a) We first give an english description and then the pseudocode. In English, the algorithm is very much like quicksort. For the pivot, we pick a random file $F[i]$. Now, for the partition step, we test $F[i]$ against each $P[j]$. For those $P[j]$ which are lexicographically less than $F[i]$'s password, we put them in an array P_{less}, and for those lexicographically greater, we put them in an array $P_{greater}$. We will also eventually find $F[i]$'s correct password, say it's $P[i^*]$, and we will label this P_{match}.

   Next, we do the same thing to the files, using $P[i^*]$ to partition them: If $F[j]$'s password is lexicographically less than $P[i^*]$, we will put it in F_{less}, and so on.

   Then we match $F[i]$ to $P[i*]$, and recurse on our arrays (F_{less}, P_{less}) and $(F_{greater}, P_{greater})$. We stop when the arrays are empty.

   The following pseudocode captures the idea described above and meets the desired runtime but does not perform in-place pivoting; it is space inefficient.

```
def unlock_files(F, P):
    Let n = length of F (which equals length of P)
    if n == 0:
        return
    Let i = random integer from 1 to n
```

```
    Let P_less = [], P_greater = [], P_match = null
    for j = start to end:
        result = test_file(F[i], P[j])
        if result == 'unlock':
            P_match = P[j]
        else if result == 'P[j] is smaller':
            P_less.append(P[j])
        else if result == 'P[j] is greater':
            P_greater.append(P[j])
    Let F_less = [], F_greater = []
    for j = start to end, skipping i:
        result = test_file(F[j], P_match)
        if result == 'P_match is smaller':
            F_greater.append(F[j])
        else if result == 'P_match is greater':
            F_less.append(F[j])
    register_match(F[i], P_match)
    unlock_files(F_less, P_less)
    unlock_files(F_greater, P_greater)

def main():
   unlock_files(F, P)
```

(b) We verify correctness by induction.

Inductive hypothesis: We use the inductive hypothesis that at the return of each recursive call to our algorithm, the files in F are matched to the passwords in P for arrays of size less than n.

Initialization: Empty arrays are already matched, so the base case holds trivially.

Maintenance: For arrays of size n, $P[i^*]$ matches $F[i]$. We recurse on (F_{less}, P_{less}) and $(F_{greater}, P_{greater})$. F_{less} has size less than n and consists of all files in F that have passwords lexicographically smaller than $F[i]$, and P_{less} contains all passwords in P that are lexicographically smaller than $P[i^*]$, and so there is a matching between the files in F_{less} and the passwords in P_{less}. The other side follows similar logic. Both the subproblems (F_{less}, P_{less}) and $(F_{greater}, P_{greater})$ have lengths less than n, so by the inductive hypothesis, our algorithm matches those files and passwords. Finally, the algorithm matches $F[i]$ with $P[i^*]$, so all n files and passwords are matched.

Conclusion: By induction, the inductive hypothesis holds for all n, hence the files in F are correctly matched to the passwords in P. Thus our algorithm is correct.

(c) To analyze this, we may do an analysis similar to the one we did in class for Quick-Sort. We outline the approach here. Let P^* denote the array of passwords in lexicographic order, and let F^* denote the files sorted to match. Let $X_{i,j}$ be an indicator random variable which is equal to 1 if file $F^*[i]$ is ever compared against password $P^*[j]$, and 0 otherwise. Because these comparisons dominate the runtime

(and because each pair $F^*[i]$ and $P^*[j]$ is compared at most once), it suffices to count them to get a $O()$ bound on the runtime.

Thus, the expected running time is O(E[number of comparisons]), which is

$$E\left[\sum_{i\neq j} X_{i,j}\right] = \sum_{i<j} EX_{i,j} + \sum_{j<i} EX_{i,j}.$$

First, focus on $EX_{i,j}$ for $i < j$. Just as in class, $EX_{i,j}$ is the probability that $F^*[i]$ and $P^*[j]$ are ever compared. This happens if and only if $F[i]$ is picked as a pivot first out of all of the files $F^*[i], F^*[i+1], \ldots, F^*[j]$. Indeed, if some other file $F^*[k]$ for $i < k \leq j$ in that range were picked first, then $P^*[j]$ would be moved into `P_greater`, since it is lexicographically larger than $P^*[k]$ which would be chosen as `P_match`; $F^*[i]$ would be moved into `F_less` for the same reason. But then $F^*[i]$ and $P^*[j]$ would end up in separate recursive calls and would never be compared. Thus, for $i < j$,

$$EX_{i,j} = \frac{1}{j-i+1},$$

because there is only one choice ($P^*[i]$) out of all $j-i+1$ possible pivot choices that would cause $P^*[j]$ and $F^*[i]$ to be compared. Similarly, for $i > j$, we have

$$EX_{i,j} = \frac{1}{i-j+1}.$$

Just as in class, when we add these up, we see that the expected running time is $O(n\log(n))$.

4. **Bloom filters.** (5 points) Hash functions are extremely good at what they do. Unsurprisingly, there are many fancier data structures that can be built on top of them. In this problem we will motivate and explore the idea of a "Bloom Filter", which is one example of a fancier structure built on top of hash functions. (Feel free to Google around for resources on Bloom Filters if you are inspired.)

   Suppose you are hired by someone to make a plagiarism detection software for internal use so as to avoid any potentially embarrassing allegations of plagiarism. Specifically, your goal is to make a lightweight (i.e. fast, and relatively low-memory) piece of software that will take a sentence and output one of the following messages: 1) "potentially problematic, please rewrite", or 2) "fresh like an ocean breeze." Suppose your goal is the following: if the input sentence is something that you have already seen, you output "potentially problematic" (with probability 1), and if the input is something new, you want to output "fresh" with probability at least 0.99 (its alright if you have a few false-alarms).

   (a) (1 point) First, you decide to use a hash table. You will make a hash table that maps a piece of text to a bucket, then scrape the web for all English sentences, and hash each one to your table. Given a new sentence, you will check to see if it hashes to an empty bucket—if so, you will output option "fresh" otherwise you will output "potential plagiarism." Suppose there are 1 Billion unique sentences online—how many buckets will your hash table need to have to have the desired functionality?
   [We are expecting a number and one to two sentences of justification.]

(b) (2 points) You decide that is a little too much space usage, and consider the following approach: you choose 10 hash functions, $h_1, \ldots, h_{10}$ that each map sentences to the numbers 1 though 10 billion. You initialize an array A of 10 billion bits, initially set to 0. For each sentence s that you encounter, you compute $h_1(s), h_2(s), \ldots, h_{10}(s)$, and set the corresponding indices of A to be 1 (namely you set $A[h_1(s)] := 1, A[h_2(s)] := 1, \ldots$). Argue that after processing the 1 Billion unique sentences, you expect a $(1 - 1/(10 \text{ billion}))^{10\text{billion}} \approx 0.37$ fraction of the elements to be 0.

For this part, feel free to assume that the h_i are "idealized" hash functions that map each key s to a uniformly random bucket.

[We are expecting a paragraph with your argument.]

(c) (2 points) Now, given a sentence s, to check if it might be plagiarized, you compute the 10 hashes of s, and check if $A[h_1(s)] = A[h_2(s)] = \ldots = A[h_{10}(s)] = 1$. If so, you output "potential problem," otherwise you output "fresh." Prove that if s is actually in your set of 1 Billion sentences, that you will output "potential problem" with probability 1, and that if s is *not* in your set of 1 Billion sentences, you will output "fresh" with probability $\approx 1 - (1 - 0.37)^{10} \approx 0.99$. [Note: again, feel free to assume that the hash functions are random, and that the claim of the previous part holds, namely that after processing the 1 Billion sentences, there are 3.7 Billion indices in the array A with value 0.]

[We are expecting informal mathematical justifications for each of the bounds.]

(d) (0 points, food for thought) Is there a better tradeoff between the number of hash functions (10), and the size of A (10 Billion)? Specifically, is there a number of hash functions, t, such that with an array A of size less than 10 Billion you could still achieve a similar probability of correctly identifying "fresh" sentences?

Solution

(a) Given a new sentence s, we output "fresh" if s hashes to an empty bucket (i.e. if s hashes to a different bucket than the 1 billion sentences). The probability that s hashes to the same bucket as a given sentence is $1/n$, so the probability that s hashes to a different bucket than a given sentence is $1 - \frac{1}{n}$.

Therefore, the probability that s hashes to a different bucket than the 1 billion sentences is $(1 - \frac{1}{n})^{1 \text{ billion}}$. Solving for n yields that n should be at least 100 billion.

(b) First of all, let i be an integer between 1 and 10 billion. Then, the probability that a given h_j $(1 \leq j \leq 10)$ hashes a given sentence to i, is: $\frac{1}{10 \text{ billion}}$.

Therefore, the probability that a given h_j hashes a given sentence to another value than i, is: $1 - \frac{1}{10 \text{ billion}}$.

So, the probability that a given h_j hashes ALL 1 billion sentences to another value than i, is: $(1 - \frac{1}{10 \text{ billion}})^{1\text{billion}}$..

And the probability that ALL $h_1, h_2 \ldots h_{10}$ hash ALL 1 billion sentences to another value than i is: $((1 - \frac{1}{10 \text{ billion}})^{1\text{billion}})^{10} = (1 - \frac{1}{10 \text{ billion}})^{10 \text{ billion}} \approx 1/e \approx 0.37$.

Each element $A[i]$ is equal to 0 if and only if ALL $h_1, h_2, h_3 \ldots h_{10}$ map ALL 1 billion sentences to another value than i. By linearity of expectation, the expected

fraction of 0s in A is simply the probability that a given element of A is equal to 0: $(1 - \frac{1}{10 \text{ billion}})^{10 \text{ billion}}$.

(c) If a sentence s is in our set of 1 Billion sentences, the array A already contains 1s at positions $h_1(s), h_2(s) \ldots h_{10}(s)$. So we will output "potentially problematic" with probability 1. So, if s is a new sentence, we output "fresh" when the following property is NOT satisfied.

- $A[h_1(s)] = 1$ and $A[h_2(s)] = 1 \ldots$ and $A[h_{10}(s)] = 1$.

Since part (b) tells us that $Pr(A[h_i(s)] = 1) = 1 - 0.37$, we an see that $Pr(A[h_1(s)] = 1$ and $A[h_2(s)] = 1 \ldots$ and $A[h_{10}(s)] = 1) = (1 - 0.37)^{10}$.

Therefore, Pr(we output "fresh" given that s is a new sentence) $= 1 - (1 - 0.37)^{10} \approx 0.99$.

5. **Colinear points.** You are given n distinct ordered pairs of integers $(x_1, y_1), (x_2, y_2), ..., (x_3, y_3)$, where for all i, j, $x_i \neq x_j$ and $y_i \neq y_j$. Recall two points uniquely define a line $y = mx + b$, with slope m and intercept b. We say a set of points S is **collinear** if they all fall on the same line; that is, for all $(x_i, y_i) \in S$, $y_i = mx_i + b$ for some fixed m and b. We want you to find the maximum cardinality subset of the given points A which are collinear. Assume that given two points, you can compute the corresponding m and b for the line passing through them in constant time, and you can compare two slopes or two intercepts in constant time. Your algorithms should not use any form of hash table.

(a) (2 points) Design an algorithm to find a maximum cardinality set of collinear points in $O(n^2 \log n)$ time. If there are several maximal sets, your algorithm can output any such set. Briefly justify the runtime of the algorithm.

[We are expecting a description (pseudocode optional) of your algorithm and a brief justification of its runtime.]

(b) (4 points) It is not known whether we can solve this collinear points problem in under $O(n^2)$ time. But suppose we know that our maximum cardinality set of collinear points consists of exactly n/k points for some constant k. Design a randomized algorithm that reports the points in some maximum cardinality set in expected time $O(n)$. (Hint: your running time may also be expressed as $O(k^2 n)$). Briefly justify the correctness and runtime of the algorithm.

[We are expecting a description (pseudocode optional) of your algorithm and a brief justification of its runtime.]

(c) (1 point) Is your algorithm from (b) guaranteed to terminate?

[We are expecting a Yes/No.]

Solution

(a) The pseudocode is as follows.

```
For all pairs of points x,y:
    compute their slope and intercepts (m, b)
Sort the pairs by (m, b)
```

```
find the longest set of identical set of (m, b)
Return the list of points in the longest set of pairs
```

Runtime: There are $O(n^2)$ pairs of points. For a given pair of points, we can compute the slope and the intercept (m, b) in $O(1)$ time. Also, since we can compare (m, b) in $O(1)$ time, sorting the (m, b) pairs gives $O(n^2 \log n^2) = O(n^2 \log n)$ time.

(b) The pseudocode is as follows.

```
while true:
    Sample two points uniformly at random and compute (m, b)
    For all other points:
        count how many satisfy y = mx + b
    If the total number of points on this line is n/k:
        return this set
```

Expected runtime: Modeling the expected value of t trials until a single success of both points being chosen from the correct max collinear set is equivalent to modelling a geometric distribution. The probability of choosing two points in the max collinear set from A is $\frac{n}{k}/n \cdot \frac{n}{k}/n = \frac{1}{k^2}$. Thus the expected value of t, by the geometric distribution, is k^2. Additionally, for each trial, the algorithm loops through all other $n - 2$ points in A, and performs constant work (calculating a slope) on each point. Thus, the expected runtime of this algorithm is $O(k^2 \cdot n)$, or $O(n)$.

(c) No, it is not guaranteed to terminate, since it does not guarantee picking the pair of points that is in the maximum set.

6. **How is the course so far?** (0.5 points extra credit)

Please complete the poll at `https://goo.gl/forms/zD2ovjb3akeZdSKC3`.