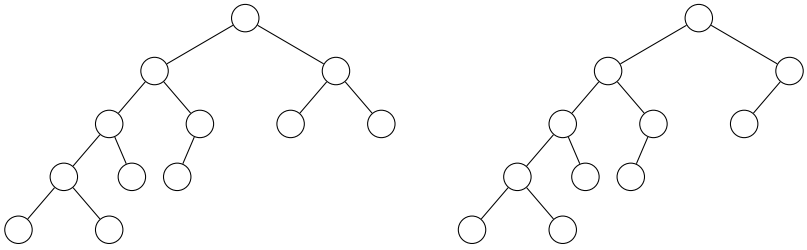
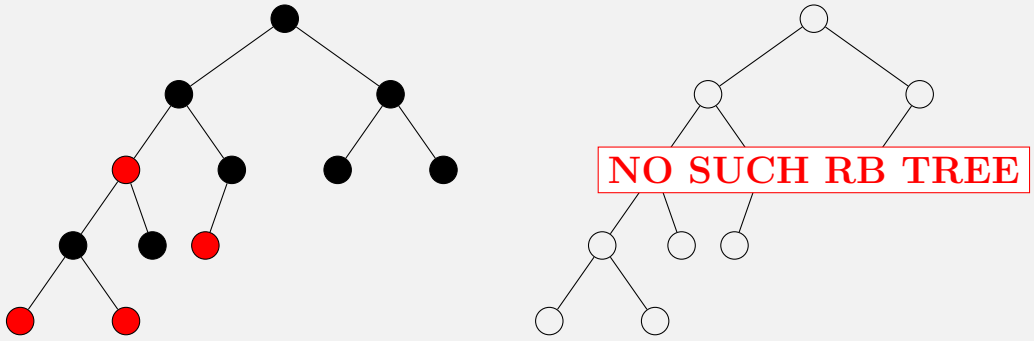
Red-Black Trees

For each of the following examples, if the nodes can be colored red or black to make a legitimate red-black tree, then give such a coloring. If not, then say that they cannot.



SOLUTION:



Randomly Built BSTs

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\log n)$. A *randomly built binary search tree* with n nodes is one that arises from inserting the n keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely.

Let $d(x, T)$ be the depth of node x in a binary tree T (The depth of the root is 0). Then, the average depth of a node in a binary tree T with n nodes is

$$\frac{1}{n}\sum_{x \in T} d(x, T).$$

1. Let the **total path length** $P(T)$ of a binary tree T be defined as the sum of the depths of all nodes in T, so the average depth of a node in T with n nodes is equal to $\frac{1}{n}P(T)$. Show

that $P(T) = P(T_L) + P(T_R) + n - 1$, where T_L and T_R are the left and right subtrees of T, respectively.

SOLUTION: Let $r(T)$ denote the root of tree T. Note the depth of node x in T is equal to the length of the path from $r(T)$ to x. Hence, $P(T) = \sum_{x \in T} d(x, T)$. For each node x in T_L, the path from $r(T)$ to x consists of the edge $(r(T), r(T_L))$ and the path from $r(T_L)$ to x. The same reasoning applies for nodes x in T_R. Equivalently, we have

$$d(x,T) = \begin{cases} 0, & \text{if } x = r(T) \\ 1 + d(x, T_L), & \text{if } x \in T_L \\ 1 + d(x, T_R), & \text{if } x \in T_R \end{cases}$$

Then,

$$\begin{aligned} \sum_{x \in T} d(x,T) &= d(r(T),T) + \sum_{x \in T_L} d(x,T) + \sum_{x \in T_R} d(x,T) \\ &= 0 + \sum_{x \in T_L} [1 + d(x,T_L)] + \sum_{x \in T_R} [1 + d(x,T_R)] \\ &= |T_L| + |T_R| + \sum_{x \in T_L} d(x,T_L) + \sum_{x \in T_R} d(x,T_R) \\ &= n - 1 + P(T_L) + P(T_R). \end{aligned}$$

It follows that $P(T) = P(T_L) + P(T_R) + n - 1$.

Problem Solving Notes:

(a) *Read and Interpret:* We are proving an expression defining $P(T)$ for a BST that refers to the left and right subtrees of a given tree T. As we learned in lecture, BSTs have unique properties that favor recursive definitions like the one seen above.

(b) *Information Needed:* How can we define each possible path in the tree with respect to the root node? After doing so, what unique property of BSTs can we apply here to rewrite that path in terms of the left or right subtree?

(c) *Solution Plan:* Write out the formulas for path lengths given the three possible options of the path ending at the root node, the path ending somewhere in the left subtree, and the path ending somewhere in the right subtree. Combine these formulas and rearrange the terms to prove the above statement.

2. Let $P(n)$ be the expected total path length of a randomly built binary search tree with n nodes. Show that $P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n - 1)$.

SOLUTION: Let T be a randomly built binary search tree with n nodes. Without loss of generality, we assume the n keys are {1, ..., n}. By definition, $P(n) = \mathbb{E}_{\mathbb{T}}[P(T)]$. Then, $P(n) = \mathbb{E}_{\mathbb{T}}[P(T_L) + P(T_R) + n - 1] = n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)]$, where T_L and T_R are

the left and right subtrees of T, respectively. Note

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n} \mathbb{E}_{\mathbb{T}}[P(T_L)|r(T) = i] \cdot Pr(r(T) = i).$$

Since each element is equally likely to be the root of T, $\Pr(r(T) = i) = \frac{1}{n}$ for all i. Conditioned on the event that element i is the root, T_L is a randomly built binary search tree on the first $i - 1$ elements. To see this, assume we picked element i to be the root. From the point of view of the left subtree, elements $1, ..., i - 1$ are inserted into the subtree in a random order, since these elements are inserted into T in a random order and subsequently go into T_L in the same relative order. Hence, $\mathbb{E}_{\mathbb{T}}[P(T_L)|r(T) = i] = P(i - 1)$. Putting these together, we get

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n} \frac{1}{n} P(i - 1).$$

Similarly, we get $\mathbb{E}_{\mathbb{T}}[P(T_R)] = \sum_{i=1}^{n} \frac{1}{n} P(n - i)$. Then,

$$\begin{aligned} P(n) &= n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)] \\ &= n - 1 + \frac{1}{n} \sum_{i=1}^{n} [P(i - 1) + P(n - i)] \\ &= n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} [P(i) + P(n - i - 1)], \end{aligned}$$

where we changed the indexing of the summation in the last equality.

Problem Solving Notes:

(a) *Read and Interpret:* The question above asks us to prove that the expected value of a path length through a randomly built binary tree T can be defined recursively. What are the input values to those recursive calls, and what do they suggest to us about how the left and right subtrees of T used to complete this proof?

(b) *Information Needed:* The formula for expected value indicates to us that we want to find the expectation of the expression we derived in question 1. How can we use the proprieties of expected value to rewrite this expression in terms of the left and right subtrees? After doing so, what probabilities are we using to calculate our expected value?

(c) *Solution Plan:* Redefine $P(n)$ in terms of summed probability-value products, differentiating between the recursive definitions for the two subtrees.

3. Show that $P(n) = O(n \log n)$. You may cite a result previously proven in the context of other topics covered in class.

SOLUTION: This is the same recurrence that appears in the analysis of Quicksort.

4. Design a sorting algorithm based on randomly building a binary search tree. Show that its (expected) running time is $O(n \log n)$. Assume that a random permutation of n keys can be generated in time $O(n)$.

SOLUTION: The algorithm is

(a) Construct a randomly built binary search tree T by inserting given elements in a random order, then

(b) Do the in-order traversal on T to get a sorted list.

Note that step (b) can be done in $O(n)$ time. We argue that step (a) takes $O(n \log n)$ time in expectation. We observe that given the final state of tree T, we can compute the amount of work spent to construct T. To insert a node x at depth d, we traversed exactly the path from the root to the parent of x, at depth $d-1$, to insert it. Hence, we can upper bound the total work done to construct T by $O(P(T))$.

From Question 3, we know that $P(T) = O(n \log n)$ in expectation. It follows that step (a) takes $O(n \log n)$ time in expectation. Overall, the algorithm runs in $O(n \log n)$ in expectation.

Problem Solving Notes:

(a) *Read and Interpret:* Our input to our sorting algorithm is a BST. Even thought our elements are inserted in random order, BSTs are built in a way such that we can access elements in-order by traversing the tree in a specific way.

(b) *Information Needed:* The big-O for the runtime of our sorting algorithm is based upon both the time needed to build the BST and the time taken to sort the elements.

(c) *Solution Plan:* While the runtime for in-order traversal is explained in lecture, showing that building a tree takes $O(n \log n)$ time requires applying our findings from question 3, which calculate that the big-O of the expected path length of T.

Pattern Matching with Rolling Hash

In the Pattern Matching problem, the input is a *text* string T of length n and a *pattern* string P of length $m < n$. Our goal is to determine if the text has a (consecutive) substring[1] that is exactly equal to the pattern (i.e. $T[i \dots i+m-1] = P$ for some i).

1. Design a simple $O(mn)$-time algorithm for this problem.

[1] In general, *subsequences* are not assumed to be consecutive, but a *substring* is defined as a consecutive subsequence.

SOLUTION: Compare P to each length m substring of T starting from index 0 to $n - m$. Return true if any substring matches exactly, and false otherwise.

This algorithm iterates through $O(n)$ substrings of T, and each check against P takes $O(m)$, making the algorithm $O(mn)$.

2. Can we find a more efficient algorithm using hash functions? One naive way to do this is to hash P and every length-m substring of T. What is the running time of this solution?

SOLUTION: Hashing every length-m substring of T takes $O(m)$ for each substring, with a total of $O(n)$ substrings. This overall is still $O(mn)$.

3. Suppose that we had a universal hash family H_m for length-m strings, where each $h_m \in H_m$ is the sum of hashes of characters (computed using a hash function h) in the string:

$$h_m(s) = h(S[0]) + \cdots + h(S[m-1]). \tag{1}$$

Explain how you would hash all m-length substrings in T using this hash family in $O(n)$ time.

(Hint: the idea is to improve over your naive algorithm by **reusing your work***.)*

SOLUTION: Each time we hash the next substring, subtract the hash of the character that was removed and add the hash of the character that was added. This takes $O(1)$ for each substring, so the overall runtime becomes $O(n)$.

4. Unfortunately, a family of "additive" functions like the one in the previous item cannot be universal. Prove it.

SOLUTION: Consider a 2 character string S, and another 2 character string S' with the characters of S in reverse order. For any $h_m \in H_m$ we have $P(h_m(S) = h_m(S')) = 1$ and $S' \neq S$.

5. The trick is to have a hash function that looks almost like (1): the hash function treats each character of the string is a little differently to circumvent the issue you discovered in the previous part, but they're still related enough that we can use our work. Specifically, we will consider hash functions parameterized by a fixed large prime p, and a random number x from $1, \ldots, p-1$:

$$h_x(S) = \sum_{i=0}^{m-1} S[i] \cdot x^i \pmod{p}.$$

For fixed pair of strings $S \neq S'$, the probability over random choice of x that the hashes are equal is at most m/p, i.e.

$$\Pr[h_x(S) = h_x(S')] \leq m/p.$$

(This follows from the fact that a polynomial of degree $(m-1)$ can have at most m zeros. Do you see why?)

Design a randomized algorithm for solving the pattern matching problem. The algorithm should have worst-case run-time $O(n)$, but may return the wrong answer with small probability (e.g. $< 1/n$).

(Assume that addition, subtraction, multiplication, and division modulo p can be done in $O(1)$ time.)

SOLUTION: Our algorithm uses the same idea from part 3, but applies this polynomial rolling hash function instead. The key insight is that if we have $h_x(T[k \ldots k+m-1])$ then we have

$$h_x(T[k+1 \ldots k+m]) = (h_x(T[k \ldots k+m-1]) - T[k])/x + T[k+m] \cdot x^{m-1} \pmod{p}$$

Algorithm 1: PATTERNMATCH(T, P)

```
p_h ← h_x(p)
for all substrings s_k ∈ T do
    if k = 0 then
        hash ← h_x(s_k)
    else
        hash ← (hash − s_{k−1}[0])/x + s_k[m] · x^{m−1}  (mod p)
    if hash = p_h then
        return True
return False
```

Runtime: Computing the hash for a substring from scratch takes $O(m)$ time. However, we compute the entire hash only for P and the first substring of T. Remaining hashes requires computing x^{m-1}, but we can precompute and store this value. This makes computing hashes for subsequent substrings $O(1)$.

Checking the hash and string equality is $O(m)$ as we must iterate through a m-length string. However, this step is expected to run $O(1)$ times, so this is also $O(1)$. The algorithm iterates over $O(n)$ substrings and each iteration is $O(1)$ in expectation, making the algorithm $O(n)$.

If we pick p to be large enough, eg. $p > mn$, then the probability of returning an incorrect solution will be $\leq \frac{1}{n}$.

Problem Solving Notes:

(a) *Read and Interpret:* We are improving upon the approach we used in part 3 by reducing the upper bound for its runtime. What information could we store during the computation process to eliminate unnecessary work, and how to we make our random choice such our algorithm meets the constraints provided in the problem?

(b) *Information Needed:* We need computing hashes for the majority of hashes, and then checking the hash and string equality to take $O(1)$ time. Therefore, we need to design an algorithm such that the time we take on each substring to be $O(1)$ in expectation, which requires precomputing information that can be used in each of the equality checks we make.

(c) *Solution Plan:* Once you've determined what information needs to be computed ahead of time, design an iterative algorithm that finds the matching substring. Note that there is a probability that our choice is wrong, which indicates our check is not as thorough as possible.

6. How would you change your algorithm so that it runs in *expected* time $O(n)$, but always return the correct answer?

SOLUTION: Modify the algorithm so that whenever the hashes match, before returning "True" it also checks that the pattern P actually matches to the substring (and if not continue the loop).

Checking takes $O(m)$ time, and in expectation we would only have to check $O(n \cdot m/p)$ times. (That's $O(n)$ hash comparisons $\times$ probability m/p of false positive each hash comparison). When $p = \Omega(n \cdot m)$, that's $O(1)$ checks.

7. Suppose that we had one fixed text T and many patterns $P_1, \ldots P_k$ that we want to search in T. How would you extend your algorithm to this setting?

SOLUTION: We can extend our algorithm simply by hashing each of $P_1, \ldots, P_k$ and checking the hash of each substring against this set of hashes.

Hash Tables Gone Crazy

In this problem, we will explore *linear probing*. Suppose we have a hash table H with n buckets, universe $U = \{1, 2, \ldots, n\}$, and a *uniformly random* hash functions $h : U \to \{1, 2, \ldots, n\}$.

When an element u arrives, we first try to insert into bucket $h(u)$. If this bucket is occupied, we try to insert into $h(u) + 1$, then $h(u) + 2$, and so on (wrapping around after n). If all buckets are occupied, output **Fail** and don't add u anywhere. If we ever find u while doing linear probing, do nothing.

Throughout, suppose that there are $m \leq n$ distinct elements from U being inserted into H. Furthermore, assume that h is chosen *after* all m elements are chosen (that is, an adversary cannot use h to construct their sequence of inserts).

1. (Warmup) Can we ever output **Fail** while inserting these m elements?

SOLUTION: No, as there are n spots in the table.

2. Above, we gave an informal algorithm for inserting an element u. Your next task is to give algorithms for searching and deleting an element u from the table.

   Hint: Be careful that the search and delete algorithm work together!

SOLUTION: When deleting, we should take special care that a previously occupied bucket is still marked as "previously occupied". Here's why: suppose $n = 3$ and $h(0) = 0, h(1) = 1, h(2) = 0$. Suppose elements were inserted in the order $0, 1, 2$. Then, $H = [0, 1, 2]$. What happens if we delete 1 and then search for 2? Well, after deleting 1, $H = [0, \square, 2]$ and so naively searching for 2 would return false, as the spot after 0 is empty.

To get around this, we mark such deletions with a "tombstone" value so that search treats those as elements.

```
def Search(H, u):
    start = h(u)
    if H[start] == u:
        return True
    current = start + 1
    while current != start:
        if H[current] == u:
            return True
        elif H[current] == empty: # X is not empty!
            return False
        current = current % n + 1 #adds 1 mod n

def Delete(H, u):
    start = h(u)
    if H[start] == u:
        H[start] = X (tombstone)
        return
    current = start + 1
    while current != start:
        if H[current] == u:
            H[current] = X
            return
        elif H[current] == empty:
            return # u is not in H
        current = current % n + 1 #adds 1 mod n
```

Problem Solving Notes:

(a) *Read and Interpret:* As we approach this problem, we need to remember that our answer to question 1 still holds. We need to ensure that we keep in mind that there are n spots in the table, and that the way in which we choose to delete elements should be accounted for when we search for one.

(b) *Information Needed:* How can we make sure that our algorithm always find the element we are searching for after we make deletions to H? Is there a way to document where elements have been deleted that we can refer back to when necessary?

(c) *Solution Plan:* Define the Search and Delete algorithms with their needed inputs and all possible cases (e.g. when the element does or does not reside in H).

3. In this part, we will analyze the runtime of linear probing, assuming no deletions occur.

   (a) Give an upper bound on the probability that $h(u) = h(v)$ for some u, v that are a part of these first m elements, assuming that $m = n^{1/3}$.
   Hint: You may need that for any $x > 0$, $(1 - x)^n \geq 1 - nx$.

SOLUTION: Number the elements $u_1, u_2, \ldots, u_m$. The probability that any of them collide with u_1 is upper bounded by $\frac{m-1}{n}$ (using a union bound), so the probability that nothing collides with u_1 is lower bounded $1 - \frac{m-1}{n}$. Given that nothing collided with u_1, the probability that nothing collides with u_2 is upper bounded by $1 - \frac{m-2}{n}$, and so on. Hence, the probability that none of these collide is

$$\mathbf{P}(\text{all distinct}) \geq \left(1 - \frac{m-1}{n}\right)\left(1 - \frac{m-2}{n}\right)\cdots\left(1 - \frac{1}{n}\right) \geq \left(1 - \frac{m}{n}\right)^m \geq 1 - \frac{m^2}{n}$$

where we lower bounded each term by $1 - \frac{m}{n}$ (and upper bounded the total number of terms by m instead of $m-1$).

Hence, an upper bound on the probability of any collision is $\frac{m^2}{n} = n^{-1/3}$.

Problem Solving Notes:

i. *Read and Interpret:* We need to upper bound a probability across all of the first m elements, which indicates that we need to create an expression for the probability that accounts for all possible pairs u, v.

ii. *Information Needed:* Remember that the probability of an event E ($P(E)$) is equal to $1 - P(\bar{E})$. Can we find the probability of no collisions using this property? Should we sample with or without replacement? Keep in mind we are adding elements to buckets in linear order.

iii. *Solution Plan:* Once you've found an expression for the probability of no collisions, lower bound the expression, then subtract it from 1 for the upper bound of the probability of at least one collision.

(b) When inserting an element, define the number of *probes* it does as the number of buckets it has to check, including the first empty bucket it looks at. For example, if $h(u), \ldots, h(u) + 10$ were occupied but $h(u) + 11$ was not then we would have to check 12 buckets.

Prove that the expected number of total probes done when inserting $m = n^{1/3}$ elements is $O(m)$.

SOLUTION: Let X be a random variable corresponding to the number of total probes done, and notice that $X \leq m^2$ (each inserted element can only be compared against the other inserted elements' positions)

Furthermore, let E be the event that there is at least one collision. Then, by conditional expectation:

$$\mathbb{E}[X] = \mathbb{E}[X \mid E]\mathbf{P}[E] + \mathbb{E}[X \mid \overline{E}]\mathbf{P}[\overline{E}] \leq \mathbb{E}[X \mid E] \cdot \frac{m^2}{n} + \mathbb{E}[X \mid \overline{E}] \leq \frac{m^4}{n} + \mathbb{E}[X \mid \overline{E}]$$

where we upper bounded the probabilities and then applied $\mathbb{E}[X \mid E] \leq m^2$. Finally, if there are no collisions, the total number of probes done is m: we only have to check one bucket per insertion. Hence, overall we have $\mathbb{E}[X] \leq \frac{m^4}{n} + m = 2m = O(m)$ (remembering that $n = m^3$).

Problem Solving Notes:

i. *Read and Interpret:* We are trying to find the **expected** number of total probes, which suggests to us that we need to represent the number of total probes as a random variable, then find its expected value.

ii. *Information Needed:* We can begin by noting that our random variable X has an upper bound—we are not making an infinite number of probes. From there, we need to determine how to break down our total possible values into discrete events with different probabilities. What are the two possible events that can happen as we attempt to place an element in a bucket?

iii. *Solution Plan:* Once you've determined what information needs to be computed ahead of time, design an iterative algorithm that finds the matching substring. Note that there is a probability that our choice is wrong, which indicates our check is not as thorough as possible.

(c) Suppose there are currently $m = \frac{n}{2}$ elements in our hash table, and we search for an existing element u (which is *not* adversarily chosen with respect to h). Prove that the expected number of probes done when searching for this element is $O(1)$.
Hint: Recall that if $0 < \delta \leq 1$ *and* $Z \sim Binom(n, p)$ *is a Binomial random variable with expectation* μ*, then* $\mathbf{P}[Z \geq (1+\delta)\mu] \leq \exp\left(-\frac{\delta^2\mu}{3}\right)$.

SOLUTION: Let X be the expected number of probes. Now, let $E_{i,k}$ be the event that there exists a set of k contiguously-filled spots in H starting at index i, such that $i-1$ and $i+k$ are not filled (in other words, this block is maximal).

Then, we claim that $\mathbb{E}[X] \leq O(1) + \sum_{i=1}^{n}\sum_{k=1}^{m} O(k^2/n)\mathbf{P}[E_{i,k}]$. The first term is because we always have to do one bucket check (the $O(1)$). The second term comes from realising that we land in block $i+j$ of a maximal block between i and $i+k-1$ with probability $1/n$ and have to do $k-j$ work after that. So if $E_{i,k}$ is true, we have to do $\sum_{j\in[0,k-1]}(k-j)/n = O(k^2/n)$ work in expectation.

Next, we note that $\mathbf{P}r[E_{i,k}]$ are equal for all i, so we may simplify the sum to $\sum_{k=1}^{m} O(k^2)\mathbf{P}[E_k]$ where E_k is the event that the block starting at index 1 is a maximal block of size k.

Next, we may simplify $\mathbf{P}[E_k] \leq \mathbf{P}[F_k]$, the probability that at least k elements hashed into a contiguous block of size k. This need not be the block starting at 1: as long as it is any block containing 1, it implies that all of $1, 2, \ldots, k$ will be filled. Therefore, our final expression is $\mathbb{E}[X] \leq O(1) + \sum_{k=1}^{m} O(k^2)\mathbf{P}[F_k]$.

Finally, we compute $\mathbf{P}[F_k]$. Consider each hash function application as either landing in this contiguous block or not, with probability $\frac{k}{n}$. Since there are a total of $\frac{n}{2}$ hash function applications, let $Y \sim \text{Binom}(\frac{n}{2}, \frac{k}{n})$. Here, $\frac{n}{2}$ is the number of trials since that's

how many elements we are inserting into the
We are interested in

$$\mathbf{P}[Y \geq k] = \mathbf{P}[Y \geq (1+1) \cdot \frac{n}{2} \cdot \frac{k}{n}] = \mathbf{P}\left[Y \geq (1+1)\mu\right] \leq \exp\left(-\frac{k}{6}\right).$$

Hence, this implies that

$$\mathbb{E}[X] \leq O(1) + \sum_{k=1}^{m} O(k^2 e^{-k/6}) = O(1) + \sum_{k=1}^{m} O(e^{-k/10}) = O(1)$$

as desired.

Problem Solving Notes:

i. *Read and Interpret:* We know from the hint above that we will be applying a Binomial random variable to solve this problem. We also know that we need to derive an expression for the expected number of probes done while *searching* for an element, which is a similar problem to what we just tackled in part (b).

ii. *Information Needed:* What events do we need to define to break down the conditional probabilities that make up our expected value calculation? Can we generalize the probability across all of our elements. Because we are calculating big-O, is there an probability that upper-bounds the one that we have found that is easier to solve for?

iii. *Solution Plan:* Once you've determined what your upper bound expression is, apply the formula for a Binomial random variable in order to show that it has an big-O of $O(1)$.