1. Let A be an unsorted array of n distinct integers. An element a stored in A is called an *approximate minimum* if the rank of a in A is at most $n/4$. We want to find an approximate minimum of A. We can find the smallest element of A in $O(n)$ time, and report it. We instead run the following $O(1)$-time Monte Carlo algorithm to find an approximate minimum of A.

> Pick a uniformly randomly from A.
> Return a.

(a) Find the error probability for this algorithm.
(b) How can you reduce the error probability by repeating the experiment a constant number of times? By how much?

Solution

For simplicity, assume that n is a multiple of 4.

(a) A randomly chosen element of A is an approximate minimum with probability $1/4$. So the error probability is $3/4$.

(b) Run the given algorithm k times independently. Return the smallest of the k elements returned by the k runs. Now, an error happens if all of the k return values are in the larger three quarters of A. This event has probability $(3/4)^k$ which decreases exponentially with k.

2. Let A be an unsorted array of n distinct integers. An element a stored in A is called an *approximate median* of A if the rank of a in A is in the range $n/4$ to $3n/4$. The problem in this exercise is to find an approximate median of A. We know that the exact median of A can be found (and reported) in $O(n)$ time. We instead run the following O(1)-time Monte Carlo algorithm for solving the current problem.

> Pick a uniformly randomly from A.
> Return a.

(a) Find the error probability for this algorithm.
(b) How can you reduce the error probability by repeating the experiment a constant number of times? By how much?

Solution

For simplicity, assume that n is a multiple of 4.

(a) A randomly chosen element of A is an approximate median with probability $1/2$. So the error probability is $1/2$ too.

(b) We choose a small positive integer k, and run the given algorithm independently on $2k+1$ occasions. Find the median m of the $2k+1$ returned elements, and return m. An analysis that the error probability decreases exponetially with k is somewhat involved.

Let S consist of the smallest $n/4$ elements of A, L the largest $n/4$ elements of A, and M all the approximate medians of A. The output m of the algorithm is errorneous if and only if $m \in S$ or $m \in L$. The two cases are equiprobable by symmetry, so we compute $\Pr[m \in S]$ only. We have $m \in S$ if and only if at least $k+1$ out of the $2k+1$ return values are in S. Binomial distribution gives

$$\begin{aligned}
\Pr[m \in S] &= \sum_{i=k+1}^{2k+1} \binom{2k+1}{i} \left(\frac{1}{4}\right)^i \left(\frac{3}{4}\right)^{2k+1-i} \\
&\leqslant \binom{2k+1}{k+1} \sum_{i=k+1}^{2k+1} \left(\frac{1}{4}\right)^i \left(\frac{3}{4}\right)^{2k+1-i} \\
&\leqslant \binom{2k+1}{k+1} \left(\frac{1}{4}\right)^{2k+1} \sum_{i=k+1}^{2k+1} 3^{2k+1-i} \\
&\leqslant \binom{2k+1}{k+1} \left(\frac{1}{4}\right)^{2k+1} \frac{3^{k+1}}{2} \\
&\leqslant 3 \times 4^{k-1} \left(\frac{1}{4}\right)^{2k+1} \frac{3^{k+1}}{2} \qquad \text{[By induction on } k \geqslant 1\text{]} \\
&= \frac{1}{2} \times \left(\frac{3}{4}\right)^{k+2}.
\end{aligned}$$

It follows that the error probability is $\leqslant (3/4)^{k+2}$.

3. Given three $n \times n$ real-valued matrices A, B, and C, we need to check if $AB = C$. The simple deterministic algorithm will take $O(n^3)$ time, which can be improved to about $O(n^{2.37})$ using the best-known matrix-multiplication algorithms. Design an $O(n^2)$-time Monte Carlo algorithm for the problem. Is the algorithm yes-biased or no-biased or with both-sided errors? Determine the error probability.

[Answer] The algorithm is as follows:

> Randomly choose a n×1 matrix r with elements from [0,1]
> If ABr = Cr, answer YES
> else answer NO

The NO answer is always correct obviously. But the YES answer may be wrong.

To find the probability of the YES answer being wrong, we need to find the probability that ABr = Cr but AB ≠ C.

Let X = (AB – C)r = Yr where Y = AB – C

So $X = [x_1, x_2, x_3, \ldots, x_n]^T$

Since AB ≠ C, Y has at least one non-zero element. Let it be y_{ij}.
Then $x_i = y_{i1}r_1 + y_{i2}r_2 + \ldots + y_{ij}r_j + \ldots + y_{in}r_n = y_{ij}r_j + z$

$P[x_i = 0] = P[x_i = 0 \mid z = 0] \times P[z = 0] + P[x_i = 0 \mid z \neq 0] \times P[z \neq 0]$ (by Bayes' theorem)

$$= P[r_i = 0] \times P[z = 0] + P[r_i = 1 \text{ and } y_{ij} = -z] \times P[z \neq 0]$$
$$\leq \frac{1}{2} \times P[z = 0] + \frac{1}{2} \times P[z \neq 0]$$
$$= \frac{1}{2}$$

4. Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ be two unsorted arrays. The elements of the arrays are chosen from a large range of integers. Repetitions are allowed in each array. Your task is to find whether A is a permutation of B. This problem can be solved in $O(n \log n)$ time by sorting the two arrays. We instead plan to solve this problem in $O(n)$ time by a Monte Carlo algorithm using a random hash function h. You do not want the error probability to be more than 1/2. You do not have to design h, but assume that for any given s, the hash function h can produce a uniformly random integer in the range 0, 1, 2, …, $s - 1$. Propose a Monte Carlo algorithm to solve this problem, and show that it gives the desired error probability.

Solution

Let A and B be two hash tables of size $2n$ each, initialized to 0.
Let h be a random hash function.
For each number a_i, set $A[h(a_i)] = A[h(a_i)] + 1$.
For each number b_j, set $B[h(b_j)] = B[h(b_j)] + 1$.
For $i = 1, 2, \ldots, 2n$, check if $A[i] \neq B[i]$.
If any such i is found, return *No*, else return *Yes*.

The algorithm is No-biased. So suppose that A is not a permutation of B, and despite that, the algorithm outputs *Yes*. The premise implies that there exists an integer a which occurs different numbers of times in A and B (call these numbers k and l, respectively). One of k and l can be 0. We have either $k > l$ or $k < l$. These two cases are symmetric, so we consider the case $k > l$ only. Let $idx = h(a)$. The element a increases $A[idx]$ exactly $k > 0$ times, whereas the same element increases $B[idx]$ exactly l times. The algorithm says *Yes*, because it detects $A[i] = B[i]$ at all indices i, and, in particular, at the index idx. That is, some element(s) in B other than a increment(s) $B[idx]$ (at least) $k - l > 0$ times. But B contains $n - l \leq n$ other elements. Let X_j be the indicator variable that the j-th other element hashes to idx. Since h is a random hash function, $\Pr[X_j = 1] = 1/(2n)$. The sum X of these variables X_j is the number of the other-than-a elements of B that hash to idx. By linearity of expectation, $E[X] \leq n/(2n) = 1/2$. By Markov's bound, $\Pr[X \neq 0] = \Pr[X \geq 1] \leq 1/2$.

5. Suppose that at each step of the min-cut algorithm, instead of choosing a random edge for contraction, two vertices are chosen at random and are merged into a single vertex. This is repeated until only 2 vertices are left. Show that there exist inputs, for which the modified algorithm finds a min-cut with exponentially small probability.

Solution

Take a large n, and construct a graph $G = (V, E)$ with V consisting of two disjoint parts S and T. Every two vertices of S share an edge. So too do every two vertices of T. Finally, there exists only one edge connecting S and T. Clearly, (S, T) is the unique minimum cut in G of size 1. Therefore the Karger–Stein variant succeeds if and only if S is contracted to one vertex, and T too is contracted to another vertex. Let us call this event A. We need to compute $\Pr[A]$. More importantly, we need to show that this probability is bounded from above by an exponential function of n. Let B be the event that the two vertices produced by the Karger–Stein variant are contractions of n vertices each. There are $\binom{2n}{n}$ such possibilities, each with equal probability (because of the random choices of the vertices). We then have

$$\Pr[A] = \Pr[A|B] \times \Pr[B] \leqslant \Pr[A|B] = \frac{1}{\binom{2n}{n}} \leqslant \frac{1}{2^n}.$$

6. Augment the Bloom filter data structure so that deletion is possible. What is the probability that an element not in the set is deleted?

Solution: Instead of a bit array, maintain an array of s counters. In order to insert an element, increment the counters at the hashed indices. For deletion, decrement the counters (provided that all of the counters store positive counts). This data structure is called a *counting (Bloom) filter*. The probability that an element not in the set is deleted is the same as the probability of false positives.

7. (a) A browser company has a huge database of n = 5,000,000 malicious URLs to check against when a user enters a URL, but it takes a long time to do a database lookup as the database is stored centrally and is not in the client computer where the browser is running. The browser designer wants to have a quick check if the URL entered is malicious or not in the web browser itself. Show how you can do it efficiently using a Bloom filter. Calculate the space saved (over a naive implementation when all URLs are directly stored in the client computer running the browser) if each URL takes on the average around 40 bytes. Assume that k = 5 hash functions are used, and the filter has size s = 40,000,000 bits. What is the false positive probability in this case?

Solution:

Storing n 40-byte strings requires a space of **200 MB**. On the other hand, the storage of an s-bit array (can be packed in an unsigned int array with s/32 entries) takes only **5 MB** space.

Use the formula for the probability p of false positives:

$$\left(1-\left(1-\frac{1}{s}\right)^{kn}\right)^{k}$$

Plugging in the given values of s, n, and k gives $p \approx 0.02168$.

(b) In this part, we use the bloom filter of Part (a). Suppose that an average user attempts to visit 100,000 URLs in a year, only 2,000 of which are actually malicious. The user wants to check whether each of these 100,000 URLs is malicious before accessing the site. Suppose that it takes half a second for the browser to check the central database for each URL, and only 1 millisecond for the browser to check in the local bloom filter.

Compare the following two times to check all 100,000 URLs.

(i) The time (in seconds) taken if no bloom filter is used, that is, every URL check is done by communicating with the central database.

(ii) The expected time (in seconds) taken if a bloom filter + database combination is used. In this case, a central database lookup is made only if the local bloom filter reports a URL as malicious.

Solution

(i) 100,000 database accesses take 100,000 × (1/2) = **50,000** seconds time.

(ii) 100,000 searches in the bloom filter takes 100 seconds of total time.

An enquiry to the central database is needed if and only if the bloom filter identifies a URL as malicious. This happens for two reasons.

True positive: There are 2,000 such cases.
False positive: The expected number of such cases is 0.02168 × 100,000 = 2168.

So a total of about 2000 + 2168 = 4168 database searches are needed requiring a time of 4168 × (1/2) = 2084 seconds.

Therefore the total overhead is 100 + 2084 = **2184** seconds.

8. [*Coupon collector's problem*] The UVW Chocolate Company prepares large numbers of n different types of coupons, and inserts them in chocolate packets to be sold in a city. The company makes sure that it has distributed an equal number of coupons of each type. If a customer can produce all of the n types of coupons to the company, she will receive a sedan as a gift from the company. In order that the company does not end up gifting too many sedans, it adopts a trick. It distributes the coupons in such a way that people from any given locality experience scarcity of some coupon types. Ms. Randoma is determined to win a gift. She guesses that the company may have played some tricks with its customers. She makes a whole-day tour in the entire city, chooses shops at random locations, and buys random chocolate packets. Find the expected number of chocolate packets she should buy in order that all of the n types of coupons are available to her. Because Ms. Randoma picks chocolate packets randomly, assume that in each packet, each of the n types of coupons is equally likely to occur in each buy.

Solution

Let Z denote the random variable standing for the number of packets needed for winning the gift. We need to compute $E[Z]$. For $i = 1,2,3,\ldots,n$, define Z_i to be the random variable that stands for the number of packets to be bought to get the i-th type of coupon *after* exactly $i-1$ different types of coupons are available. We have $Z = Z_1 + Z_2 + \cdots + Z_n$, so by linearity of expectation, $E[Z] = E[Z_1] + E[Z_2] + \cdots + E[Z_n]$. In order to determine $E[Z_i]$, we note that Ms. Randoma has $i-1$ different types of coupons, and she needs any one of the remaining $n-i+1$ types of coupons. The probability that a randomly chosen packet contains a new type of coupon is $\alpha_i = \frac{n-i+1}{n}$. By Part (a), $E[Z_i] = \frac{1}{\alpha_i} = \frac{n}{n-i+1}$, and it follows that

$$E[Z] = n\sum_{i=1}^{n} \frac{1}{n-i+1} = nH_n,$$

where H_n is the n-th harmonic number. Since $H_n \approx \ln n$ (indeed $\ln(n+1) \leqslant H_n \leqslant \ln n + 1$ for all n), about $n \ln n$ packets should be bought to expect a win with significant probability.

9. A thief steals $n > 2$ gold balls of identical shape and size. Later, he comes to know from an informant that exactly one of the balls is made of city gold. He also learns that the city-gold ball has weigth slightly different from a (genuine) gold ball. He wants to separate out the city-gold ball from his collection of n balls so that he can sell the $n - 1$ gold balls, and give the city-gold ball to his wife as a gift. However, he needs a precision instrument to detect slight weight variations. The Bar Foo Jewellery Shop has one such instrument which can only compare the weights of two balls. However, for each use of the instrument, they charge a hefty 1000 Rs fee. So the thief wants to minimize the number of weighings. To that effect, he runs the following Las Vegas algorithm.

Randomly permute the balls as $b_1, b_2, b_3, \ldots, b_n$.
Compare b_1 with b_2.
If the weights are different, then:
    Compare b_1 with b_3.
    If the weights are again different, return b_1, else return b_2.
Else:
    For $i = 3, 4, 5, \ldots, n - 1$, repeat:
        Compare b_1 with b_i.
        If the weights are different, return b_i.
    Return b_n.

Deduce the expected number of weighings done by this algorithm. How does randomization help here?

Solution

Because the balls are initially randomly permuted, each b_i is the city-gold ball with probability $\frac{1}{n}$. If b_1 or b_2 is the city-gold ball, then two weighings are required. If b_i is the city-gold ball for some i in the range $3 \leqslant i \leqslant n-1$, then $i-1$ weighings are required. Finally, if b_n is the city-gold ball, then $n-2$ weighings are required. So the expected number of weighings is

$$\begin{aligned}
& \frac{2}{n} \times 2 + \frac{1}{n} \times (2+3+4+\cdots+n-2) + \frac{1}{n} \times (n-2) \\
= \; & \frac{2+(1+2+3+\cdots+n-1)}{n} \\
= \; & \frac{2+\frac{n(n-1)}{2}}{n} \\
= \; & \frac{n^2-n+4}{2n}.
\end{aligned}$$

10. Consider the situation of the previous exercise. Suppose that the weighing machine of Bar Foo Jewellery Shop can compare any number of balls (equal numbers on both sides). Assume that n is a power of two. Prove that the thief can identify the city-gold ball using $\log_2 n$ weighings.

Solution

Initially, the collection of n balls has the city-gold ball. The thief divides this collection into two equal subcollections of size $n/2$. He picks any subcollection arbitrarily, and compares the weight of $n/4$ balls of this subcollection with the remaining $n/4$ balls of the same subcollection. If the weights are different, then the chosen subcollection has the city-gold ball. Otherwise, the other subcollection contains the city-gold ball. With one weighing, the thief reduces the size of the collection containing the city-gold ball, by a factor of 2. Eventually, he narrows down to only two balls, one of which must be the city-gold ball. By making a single weighing comparison of any of the two balls with any genuine ball discarded earlier, the city-gold ball can be identified. The number of weighings made is therefore $\log_2 n$.

11. Consider again the situation of the previous two exercises. Suppose that the Bar Foo Jewellery Shop charges $1000k$ Rs if k balls are compared with k balls in one weighing. What is the payment that the thief needs to make if he uses the algorithm of the previous exercise? Can randomization help the algorithm?

Solution

The payment is 1000 times

$$\frac{n}{4}+\frac{n}{8}+\cdots+1+1=\frac{n}{2}.$$

This cannot be improved by randomization, because the payment is the same in all cases (best or worst or random).

12. [*Quick Select*] Let A be an unsorted array of n distinct integers, and k an integer in the range $1 \leq k \leq n$. Your task is to find the k-th smallest element of A. To that effect, you choose an element uniformly randomly from A. Using that element as the pivot, you partition the array. Suppose that the pivot occupies the position j in the sorted array (assume that array indexing is 1-based). If $j = k$, you return the pivot. Otherwise, you make a recursive call on the smaller or larger (than the pivot) part. Deduce the expected and worst-case running times of this algorithm.

Solution

The pivot x is an approximate median (Exercise 6.91) with probability $\frac{1}{2}$. If that is the case, at least $n/4$ elements of A (the smallest quarter or the largest quarter of A) are discarded in the recursive call. Otherwise, the recursive call is on an array of size at most $n-1$. Therefore the expected running time $T(n)$ of quick select satisfies

$$T(n) \leqslant cn + \frac{1}{2}T(3n/4) + \frac{1}{2}T(n-1),$$

where c is a constant, and cn stands for an upper bound on the running time of the partition stage. I now show that $T(n) \leqslant dn$ for some constant d for all sufficiently large n. Substituting this form in the recurrence gives

$$\begin{aligned} T(n) &\leqslant cn + \frac{1}{2}T(3n/4) + \frac{1}{2}T(n-1) \\ &\leqslant cn + \frac{1}{2} \times \frac{3dn}{4} + \frac{1}{2} \times d(n-1) \\ &= \left[c + \left(\frac{7}{8} - \frac{1}{2n}\right)d\right] n \\ &\leqslant \left[c + \frac{7}{8}d\right] n. \end{aligned}$$

It we choose any constant $d \geqslant 8c$, we have $T(n) \leqslant dn$.

The worst-case running time has $T(n) \leqslant cn + T(n-1) = \mathrm{O}(n^2)$.

13. A Las Vegas algorithm always outputs correct answers. Let us investigate a similar class of algorithms which may sometimes report *failure*. But whenever the algorithms succeed, the outputs are correct. Let us call such an algorithm a Las Vegas′ algorithm. Let A' be a Las Vegas′ algorithm for some problem. Using this algorithm, design a Las Vegas algorithm A that never outputs *failure*. Assume that A' outputs *failure* with probability $\leq 1/2$. Express the expected running time of A in terms of the expected running time of A'.

Solution

Keep on running A' until an output (other than *failure*) is produced. The expected running time of A is no larger than twice the expected running time of A'.

14. A randomized algorithm is called an *Atlantis City algorithm* if it is probably correct and probably fast. You are given a positive integer l. Your task is to locate a random l-bit prime number. Propose an Atlantis City algorithm for this problem. Deduce the error probability and the expected running time of your algorithm.

You may assume the *prime number theorem* which states that for a positive real number x, the number of primes $\leq x$ is approximately $x / \ln x$.

Solution

By the prime number theorem, the number of l-bit primes is approximately $\frac{2^l-1}{\ln(2^l-1)} - \frac{2^{l-1}-1}{\ln(2^{l-1}-1)} \approx \frac{2^{l-1}}{l \ln 2}$. There are 2^{l-1} integers of bit length l. Therefore a randomly chosen l-bit integer is prime with probability $\frac{1}{l \ln 2}$. A randomized algorithm for generating a random l-bit prime can be designed based on this fact. We keep on choosing random l-bit (odd) integers n. We then run a primality test on n. If the test returns *true*, that n is returned. We expect $\Theta(l)$ iterations to locate a prime.

If we use a polynomial-time deterministic primality test, this is a Las Vegas algorithm. However, if we use a polynomial-time Monte Carlo primality test (like the Miller–Rabin test), the prime returned by the algorithm is composite with some error probability (the same as the primality test itself).

15. Propose an efficient algorithm for computing a random permutation of 1, 2, 3, …, n. You should ensure that your algorithm outputs a permutation with probability $1 / n!$.

Solution sketch:

Initialize an array A as $A[i] = i$ for $i = 1, 2, \ldots, n$.
For $i = 1, 2, \ldots, n - 1$, repeat
    Pick j uniformly randomly from $\{i, i + 1, \ldots, n\}$.
    Swap $A[i]$ with $A[j]$.