

CS 630, Fall 2024, Homework 5

Yifei Bao

November 6, 2024

Problem 1 *Fair numbers from biased coins*

1. The algorithm for $\text{2-WayFair}(p)$ involves flipping the biased coin twice. If the result is “heads-tails” or “tails-heads,” we return 0 or 1, each with an equal probability. If the result is “heads-heads” or “tails-tails,” we disregard the result and try again.

The probability of successfully obtaining either “heads-tails” or “tails-heads” is $2p(1-p)$. Therefore, the expected number of iterations it takes for $\text{2-WayFair}(p)$ to return an answer is $\frac{1}{2p(1-p)}$.

2. We call $\text{2-WayFair}(p)$ twice to generate a two-bit binary number b_1b_2 , the result b_1b_2 as follows: 00=0, 01=1, 10=2. Since each of $b_1b_2 = 00$, 01, and 10 has an equal probability of $\frac{1}{4}$, each outcome (0, 1, 2) is returned with probability $\frac{1}{3}$.

Each iteration (two calls to $\text{2-WayFair}(p)$) is successful with a probability of $\frac{3}{4}$ (since three out of four two-bit combinations are valid). Thus, the expected number of iterations for $\text{3-WayFair}(p)$ is: Expected iterations = $\frac{4}{3}$

Algorithm 1: $\text{3-WayFair}(p)$

```
1 while true do
2   bit1 ← 2-WayFair(p);
3   bit2 ← 2-WayFair(p);
4   /* Combine two bits into a binary number */
5   result = 2 * bit1 + bit2 /* Forms 00, 01, or 10 as decimal values 0, 1, 2 */
6   if result < 3 then
7     return result /* Return 0, 1, or 2 with equal probability */
```

3.

The algorithm generates an n -bit binary number B by calling $\text{2-WayFair}(p)$ n times, where n is chosen so that $2^n \geq k$. This n -bit number can represent any integer from 0 to $2^n - 1$, each occurring with equal probability because of $\text{2-WayFair}(p)$. The algorithm accepts B if it is in the range $\{0, 1, \dots, k-1\}$; otherwise, it repeats. Since each n -bit number has a probability of $\frac{1}{2^n}$ and there are k acceptable numbers, the probability of returning any specific outcome from 0 to $k-1$ is $\frac{1}{2^n} \times \frac{2^n}{k} = \frac{1}{k}$. Thus the algorithm returns values uniformly.

Each iteration succeeds with probability $\frac{k}{2^n}$, as there are k acceptable outcomes out of 2^n possible n -bit values. Therefore, the expected number of iterations needed to generate a valid output is the reciprocal of the success probability, which gives $\frac{2^n}{k}$ iterations on average.

Algorithm 2: k-WayFair(p)

```

1  $n \leftarrow \text{Ceil}(\log_2(k))$  /* Find minimum bits needed,  $n$ , such that  $2^n \geq k$  */
2 while true do
3    $\text{result} \leftarrow 0$ ;
4   for  $i \leftarrow 1$  to  $n$  do
5      $\text{bit} \leftarrow \text{2-WayFair}(p)$ ;
6      $\text{result} \leftarrow 2 * \text{result} + \text{bit}$  /* Combine bits into a number */
7   if  $\text{result} < k$  then
8     return  $\text{result}$  /* Return a number between 0 and  $k-1$  */

```

Problem 2 *Uniform sample*

1.

Space complexity. The space complexity is $O(1)$, because we only need a constant amount of space to store the sample S .

Time complexity. The per-iteration time complexity is $O(1)$.

Algorithm 3: UniformSample($stream$)

```

/* Input: A stream of numbers arriving one at a time */
/* Output: A sample  $S$  of 3 numbers from the stream, with each number
           having probability  $\frac{3}{n}$  to be in  $S$  after  $n$  numbers have arrived */
1  $S \leftarrow$  an empty array of size 3;
2  $\text{count} \leftarrow 0$  /* Count of numbers seen so far */
3 while  $stream.hasNext()$  do
4    $\text{count} \leftarrow \text{count} + 1$ ;
5    $x \leftarrow stream.next$ ;
6   if  $\text{count} \leq 3$  then
7      $S[\text{count} - 1] \leftarrow x$  /* Directly add the first 3 numbers to  $S$  */
8   else
9      $r \leftarrow \text{RandomInt}(1, \text{count})$ ;
10    if  $r \leq 3$  then
11       $S[r - 1] \leftarrow x$  /* Replace an element in  $S$  */
12 return  $S$ ;

```

2. Proof. For the first three numbers in the stream, they are directly placed in the sample S , so each of these has a probability of $\frac{3}{n}$ of remaining in S at any point.

For any subsequent number i (where $i > 3$), the probability of it being selected into the sample S upon arrival is $\frac{3}{i}$. If it is selected into S , the probability that it remains in S until the arrival of the n_{th} number is given by:

$$\prod_{j=i+1}^n \left(1 - \frac{3}{j}\right)$$

After simplifying this product, it gets an overall probability of $\frac{3}{n}$ for any specific number to be in S by the time n numbers have arrived. This ensures that each number has the same probability of being in the sample S , maintaining uniformity.

3. To generalize the algorithm for maintaining a sample of size k , adjust the sample size from 3 to k and to generate a random integer r in the range 1 to $count$. If $r \leq k$, replace the r_{th} element in S with the current number.

Proof. Similarly to the case where the sample size is 3, the probability of selecting the i_{th} number is $\frac{k}{i}$, and the probability of it remaining in S is $\frac{k}{n}$ after considering all subsequent arrivals. This approach maintains a uniform probability for each number to be in S .

Problem 3 *hash*

1. *Time complexity.* Calculating the hash index is $O(1)$. Appending to the end of the linked list is $O(1)$ on average. Thus, the average time complexity for insertion is $O(1)$.

Algorithm 4: Insert($H, x, value$)

```

1 index ← hash(x) % m /* Compute the hash index and find the slot */
2 if H[index] == NULL then
3   H[index] ← new LinkedList /* Create new linked list if the slot is empty */
4 H[index].add((x, value)) /* Add the key-value pair to the linked list */

```

2. *Time complexity.* Calculating the hash index is $O(1)$. Searching the linked list for the first occurrence of x has an average time complexity of $O(\alpha)$, where $\alpha = \frac{n}{m}$. Thus, the average time complexity for this operation is $O(\alpha)$.

Algorithm 5: Contains(H, x)

```

1 index ← hash(x) % m /* Compute the hash index and find the slot */
2 if H[index] == NULL then
3   return false /* Return false if the slot is empty */
4 for (k, v) in H[index] do
5   if k == x then
6     return true /* Return true if the key is found */
7 return false /* Return false if the key is not found */

```

3. Verifying that a key y is not in the hash table is similar to the contain function. First compute the hash index and locate the slot. If the slot is empty, y is not in the table. If the slot contains a linked list, search through it. If y is not found, return False.

Time complexity. Calculating the hash index is $O(1)$. In the average case, if the key y is not in the table, we need to search through the linked list in that slot to confirm its absence. The average runtime for this operation is $O(\alpha)$, similar to checking if a key is in the hash table.