Algorithmica (2018) 80:1439–1458
https://doi.org/10.1007/s00453-017-0335-z

Sampling in Space Restricted Settings

Anup Bhattacharya[1] · **Davis Issac**[2] · **Ragesh Jaiswal**[1] · **Amit Kumar**[1]

Received: 1 November 2015 / Accepted: 9 June 2017 / Published online: 14 June 2017

Abstract Space efficient algorithms play an important role in dealing with large amount of data. In such settings, one would like to analyze the large data using small amount of "working space". One of the key steps in many algorithms for analyzing large data is to maintain a (or a small number) random sample from the data points. In this paper, we consider two space restricted settings—(i) the streaming model, where data arrives over time and one can use only a small amount of storage, and (ii) the query model, where we can structure the data in low space and answer sampling queries. In this paper, we prove the following results in the above two settings:

– In the streaming setting, we would like to maintain a random sample from the elements seen so far. We prove that one can maintain a random sample using $O(\log n)$ random bits and $O(\log n)$ bits of space, where n is the number of elements seen so far. We can extend this to the case when elements have weights as well.

Davis Issac: Major part of this work was done when the author was at IIT Delhi. Ragesh Jaiswal acknowledges the support of ISF-UGC India–Israel Joint Research Grant 2014.

✉ Ragesh Jaiswal
rjaiswal@cse.iitd.ac.in

Anup Bhattacharya
anupb@cse.iitd.ac.in

Davis Issac
dissac@mpi-inf.mpg.de

Amit Kumar
amitk@cse.iitd.ac.in

1 Department of Computer Science and Engineering, IIT Delhi, New Delhi, India

2 Max Planck Institute for Informatics, Saarbrücken, Germany

– In the query model, there are n elements with weights $w_1, \ldots, w_n$ (which are w-bit integers) and one would like to sample a random element with probability proportional to its weight. Bringmann and Larsen (STOC 2013) showed how to sample such an element using $nw + 1$ bits of space (whereas, the information theoretic lower bound is nw). We consider the approximate sampling problem, where we are given an error parameter ε, and the sampling probability of an element can be off by an ε factor. We give matching upper and lower bounds for this problem.

1 Introduction

Space optimization is becoming more and more important, especially with the rise in popularity of the mobile devices. Many of these devices have a small working memory and are expected to do many of the operations performed by devices with larger memory. When the working memory is small and the data is large, space efficiency of algorithms become crucial. Also, in the fields of data mining, social network analysis etc., huge amount of data has to be analyzed in a *streaming* fashion. Randomness required by the algorithms is also an important factor. The amount of randomness that need to be generated by the pseudorandom generator plays a role in determining the power of a device. Power is one of the most important resource to optimize for computing devices.

In this work, we look at optimizing space and randomness for the basic problem of *discrete random sampling*. In discrete sampling, we are given n objects as input. In *uniform discrete sampling*, we are required to output an object uniformly at random from the n input objects. This can be generalized to *weighted discrete sampling* where the probability of an object being the output is proportional to its weight. From now on, when we refer to sampling, we mean discrete sampling. We consider these sampling problems in two different space-restricted settings, *the streaming setting* and *the query model*. In the streaming setting, the objects appear one by one. We do not have access to all data items simultaneously. A discarded object can never again be accessed. Also, we do not have a priori knowledge of the total number of objects in the stream. Maintaining a random sample at all time points in a streaming setting is more challenging than the classical setting where we have access to all objects simultaneously. In the query model, we are allowed to preprocess the data and store a representation of it in small space. The representation should be capable of answering sampling queries with good enough speed.

1.1 Sampling in the Streaming Setting

In this setting, the data objects appear in a streaming manner. That is, at time i, the ith object appears. We are required to maintain a uniform sample of the objects seen so far at all points of time. We will also generalize this to weighted sampling. We want the sampling algorithm to be one-pass. The algorithm is allowed to store only one object at a time. Such a scenario occurs when the objects are very large files or packets.

The most simple way of doing streaming sampling is by *reservoir sampling* which works as follows: Let the objects in the stream be $O_1, O_2, \ldots$ and let the storage space be S. At time $t = 1$, O_1 is stored in S. In all the subsequent time points t, the element in S is swapped with O_t with probability $\frac{1}{t}$. With the remaining probability, O_t is discarded. It is easy to see that this procedure satisfies the uniform sampling requirement at all time points. Now, let us estimate the amount of randomness required by the reservoir sampling. At time t, we need to simulate a probability of $\frac{1}{t}$, which requires at least $\log t$ bits.[1]. This means that when n objects have appeared, we need to use at least $\sum_{t=1}^{n} \log t = \Omega(n \log n)$ random bits.[2] Now, compare this with the classical setting where we need only $O(\log n)$ random bits in expectation to sample 1 object from n objects. It is an interesting question, whether such a gap should exist between the number of random bits required in the classical and streaming settings. In this work, we answer this question by showing that there is no gap. We give an algorithm which can do the streaming sampling with the same order of random bits as in the classical setting. In the streaming model, we do not have the liberty of repeating the procedure and give a bound on the expected number of rounds required. Hence, we consider a model where we define strict bounds on randomness rather than the expected randomness. In other words, we work with a Monte Carlo model instead of a Las Vegas model.

First of all, it is not possible to define deterministic upper bounds on randomness with respect to perfect uniform sampling. This can be shown by considering the case when $n > 1$ and n is odd. Suppose r is the number of random bits required to sample from n objects. This means that there exists function $f : \{0,1\}^r \rightarrow [n]$ such that for all $i, j \in [n]$, $|\{x | f(x) = i\}| = |\{x | f(x) = j\}|$. But then 2^r is divisible by n, which is a contradiction to the fact that n is odd. One way to get around this is to allow the algorithm to output null (which means that it does not output any of the input objects) with a small probability. We say that an algorithm performs *uniform sampling with ε-error*, if the algorithm outputs null(denoted by $\perp$) with at most ε probability and given that it outputs something, each of the input object has the same probability to be the output. We also extend the definition to the weighted case. We say that an algorithm performs *weighted sampling with ε-error*, if the algorithm on given input objects and weights associated with each of them, outputs null(denoted by $\perp$) with at most ε probability and given that it outputs something, the probability of each of the input object to be the output is proportional to its weight.

It is easy to show (see Sect. 2) a lower bound of $\Omega(\log \frac{n}{\varepsilon})$ on the number of random bits required to do uniform sampling with ε-error. Following is a simple algorithm that does uniform sampling with ε-error in the non-streaming setting using $O(\log \frac{n}{\varepsilon})$ random bits: First compute the smallest r such that $2^r \geq n$ and $2^r \ mod\ n \leq \varepsilon \cdot 2^r$. It can be shown that $r = O(\log \frac{n}{\varepsilon})$. Let $k = \lfloor \frac{2^r}{n} \rfloor$. Now, let $f : \{0,1\}^r \rightarrow [n] \cup \{\perp\}$ be

[1] It may require more number of bits when t is not a power of 2. But we can say that in expectation we need $O(\log t)$ number of random bits.

[2] We will later discuss a sampling algorithm by Vitter which can be adjusted to work with $O(\log^2 n)$ random bits.

a function which maps the first k r-bit strings (in any fixed order) to 1, the next k to 2 and so on. The last 2^r *mod* n bit strings are mapped to $\perp$. The sampling algorithm generates an r-bit random string R and then outputs $f(R)$. Can we extend this simple algorithm to the streaming setting to get a randomness efficient streaming sampling algorithm? The answer is negative. The main hindrance in the streaming setting is that we do not know the value of n in advance and that the uniform sample should be maintained at all points of time.

1.2 Succinct Sampling

The second space-restricted setting that we consider is the *query model* used for example by Bringmann and Larsen [2] in their work. Our work in this model can be considered to be a natural extension of their work. In this model we need to sample from the set $\{1, \dots, n\}$ according to a given distribution. The inputs are w-bit numbers $x_1, \dots, x_n$. We are allowed to preprocess the data and construct suitable data structures. The data structure should be capable of answering sampling queries quickly. A sampling query should return i with probability $\frac{x_i}{\sum_j x_j}$ for each $i \in [n]$. The resource that we are primarily interested in optimizing is the storage space required for the data structure.

Bringmann and Larsen [2] analyzed the classical Walker's alias method [12] in the word RAM model (here unit operations may be performed on words of size w bits) and observed that the algorithm has $O(n)$ preprocessing time, $O(1)$ query time and requires a storage space of size $n(w + 2\log n + o(1))$ bits. The *redundancy* of a solution is defined as the number of bits of storage required in addition to the information theoretic minimum required for storing the input. This means that Walker's alias method has a redundancy of $(2n\log n + o(n))$-bits. Bringmann and Larsen [2] distinguish between the *systematic case* , where the given inputs are read only, and the *non-systematic case*, where one may represent data in more clever ways than the given sequential input format. In the systematic case, they give an algorithm with a preprocessing time of $O(n)$, expected query time of $O(1)$ and a redundancy of $n + O(w)$ and in the non-systematic case they give an algorithm which has 1 bit of redundancy. Furthermore, they also proved the optimality of their solutions. But, all their results are for *exact* sampling. In our work, we take a look at how this work can be extended to approximate sampling within the word RAM model.

In many real applications, we may not require to sample exactly as given by the input distribution. For example, the sampling based algorithms for k-means clustering such as the PTAS by Jaiswal et al. [5] are robust to small errors in sampling probability. In fact, this was the starting point of this work. There are many such scenarios where it is sufficient that the sampling probabilities are *close* to the probabilities given by the input distribution $x_1, x_2 \dots x_n$. We will consider two natural models of closeness. First one is the *additive model*, where the sampling probability is allowed to be between $\left(\frac{x_i}{\sum_j x_j} - \varepsilon\right)$ and $\left(\frac{x_i}{\sum_j x_j} + \varepsilon\right)$ for some given small ε. Second is the *Multiplicative model*, where the sampling probability is allowed to be between $(1-\varepsilon) \cdot \left(\frac{x_i}{\sum_j x_j}\right)$ and $(1+\varepsilon) \cdot \left(\frac{x_i}{\sum_j x_j}\right)$ for some given small ε.

Before stating our results, we point out some differences between approximate and exact sampling in terms of space usage. In the case of exact sampling from a distribution given by n w-bit integers, we are limited by an information theoretic lower bound of nw. [3] But in the case of approximate sampling, the information theoretic lower bounds could be much lower because we can use some lossy data representation that saves much space without affecting the sampling probabilities by a lot. Hence, in the approximate setting, the non-systematic case (where we are allowed to restructure the input) seems to be more relevant than the systematic case (where the input data has to be retained as given). So, we only discuss the non-systematic case for approximate sampling in our work.

We say that an algorithm performs *approximate sampling with ε additive error*, if the algorithm on given n w-bit non-negative integers $x_1, x_2, \ldots, x_n$, outputs a $z \in [n]$ such that for all $i \in [n]$, $\left(\frac{x_i}{\sum_j x_j} - \varepsilon\right) \leq Pr[o = i] \leq \left(\frac{x_i}{\sum_j x_j} + \varepsilon\right)$. We say that an algorithm performs *approximate sampling with ε multiplicative error*, if the algorithm on given n w-bit non-negative integers $x_1, x_2, \ldots, x_n$, outputs a $z \in [n]$ such that for all $i \in [n]$, $(1-\varepsilon) \cdot \left(\frac{x_i}{\sum_j x_j}\right) \leq Pr[o = i] \leq (1+\varepsilon) \cdot \left(\frac{x_i}{\sum_j x_j}\right)$. Note that the probabilities are over the randomness used by the algorithm.

1.3 Our Results

Streaming Sampling We give an algorithm that we call the *Doubling–Chopping algorithm* that is optimal in terms of number of random bits used and is also space and time efficient. Moreover, we show that there is no gap between the classical and streaming settings as far as randomness is concerned.

Theorem 1 *The Doubling–Chopping algorithm (described in Sect. 2.2) performs uniform sampling with ϵ-error in the streaming setting using $O(\log \frac{n}{\varepsilon})$ random bits. Moreover, any algorithm that does uniform sampling with ε-error (even in the non-streaming setting) requires $\Omega(\log \frac{n}{\varepsilon})$ random bits. (See Lemma 1) The doubling–chopping algorithm uses $O(\log \frac{n}{\varepsilon})$ bits of working space and runs in time $O(n + \log \frac{1}{\varepsilon})$. The algorithm requires only $O(1)$ time per object and a preprocessing time of $O(\log \frac{1}{\varepsilon})$.*

We also extend the results to weighted sampling with ϵ-error in the streaming setting.

Theorem 2 *There exist an algorithm that performs weighted sampling with ϵ-error in the streaming setting using $O(w + \log \frac{n}{\varepsilon})$ random bits where we assume each of the weights is a w-bit positive integer. Moreover, any algorithm that does weighted sampling with ε-error (even in the non-streaming setting) requires $\Omega(w + \log \frac{n}{\varepsilon})$ random bits. (See Sect. 2.3)*

Succinct Sampling For the multiplicative model, we give a lower bound for the space required and also a sampling algorithm whose space usage matches this lower bound. See Sect. 3.1 for a comparison of the upper and lower bounds.

[3] Note that this is not a trivial observation since $x_1, x_2, \ldots, x_n$ and $\frac{x_1}{2}, \frac{x_2}{2}, \ldots, \frac{x_n}{2}$ both represent the same probability distribution. See lemma 5.1 in [2].

Theorem 3 *There exists an algorithm which performs approximate sampling with ε multiplicative error using a space of $\left(n \log w + n \log \frac{2}{\varepsilon}\right)$ bits. Moreover, any algorithm which performs approximate sampling with ε multiplicative error requires to use at least*

$$\max\left\{\left(n \log \frac{1}{\epsilon} - w - \log n - n\right), \left(n \log w - n \log 4(1+2\epsilon) - \frac{w}{2} \log (e^2 n)\right)\right\}$$

bits. (See Sect. 3.1 for a description of the algorithm and proof of the claims).

In the additive model, we give similar results. However, in this case our algorithms match the lower bound for space usage only in the case when ε is a constant independent of n.

Theorem 4 *There exists an algorithm which performs approximate sampling with ε additive error using a space of $\frac{1}{\varepsilon} \log n$ bits. Any algorithm which performs approximate sampling with ε additive error requires to use a space of $\Omega\left(\frac{1}{\varepsilon} \cdot \log(1+\varepsilon n) + n \log \left(1 + \frac{1}{\varepsilon n}\right)\right)$ bits. (See Sect. 3.2 for a description of the algorithm and proof of the claims)*

1.4 Related Work

Streaming Sampling There has been a lot of work done in streaming sampling starting from [6,10,11], which discussed some of the preliminary techniques and ideas in sampling, especially reservoir sampling. Most of the previous work done tries to optimize the running time and there is no previous work to the current knowledge of the authors which try to optimize randomness. Vitter [11] did some of the earliest work in the area of streaming sampling. The author was interested optimizing the running time and not concerned about the amount of randomness. The algorithm in fact assumes that one can draw a random number of infinite precision from [0, 1]. Li [8] gave a better running time than Vitter's work. These techniques where extended to sampling with replacement by Park et al. [9] and to weighted sampling by Efraimidis and Spirakis [4]. Babcock et al. [1] studied and gave algorithms for the case of sliding window sampling, where it is required to maintain random samples from a window of the most recent items.

Comparison with Vitter's Reservoir Sampling The most relevant previous work to our work in streaming sampling is the work by Vitter [11]. So, we compare our results with those in [11]. Recall that the naïve algorithm which stored ith item with probability $\frac{1}{i}$, uses $O(n \log n)$ random bits in expectation. The large number of random bits required is due to the use of fresh random bits whenever a new item arrives. Vitter gives a more advanced method, which skips some of the items. After i items have been processed, the algorithm chooses a positive integer s according to the probability distribution $f_i(s) = \frac{i}{(i+s)(i+s+1)}$. Then, it skips the next s items. It is shown that this algorithm gives a uniform sample. Now, the randomness is only required in selecting the number of skips. For choosing s, the algorithm assumes that it can choose a real number u with infinite precision from [0, 1]. Then, it chooses s to be the smallest number x such that the cumulative probability $F = \sum_{i \leq x} f_i(x)$ is at least u.

Before analyzing Vitter's algorithm further, we point out two differences in the models used by our work and Vitter [11]. The first difference is that Vitter uses the assumption that random numbers of arbitrary precision can be drawn from [0, 1]. But in our work, we consider randomness in terms of absolute random bits. Secondly, both the basic reservoir algorithm and Vitter's algorithm (after suitable modification discussed below to remove the assumption about arbitrary precision random numbers), give upper bounds on the *expected* amount of randomness. Our work on the other hand, give guarantees on the worst case number of random bits needed, provided that the algorithm is allowed to err with a small input probability. In other words, one can think of our algorithm as a Monte Carlo Algorithm and that of Vitter as a Las Vegas Algorithm.

In order to estimate the number of random bits required by Vitter's algorithm, we need to get rid of the requirement of sampling with arbitrary precision from [0, 1]. So, we address the question whether the process of choosing s according to f_i can be done with few random bits instead of sampling real numbers from [0, 1]. Let us design an algorithm for this. First, Observe that

$$\mathbf{Pr}[s > i] = 1 - \sum_{0 \leq j \leq i} f_i(j) = 1 - \sum_{0 \leq j \leq i} \left(\frac{i}{i+j} - \frac{i}{i+j+1} \right) \leq \frac{1}{2}.$$

Let us define a distribution $\mathcal{D}$ as $\mathcal{D}(j) = f_i(j), \forall 0 \leq j \leq i$ and $\mathcal{D}(i+1) = \sum_{j>i} f_i(j)$. Now, consider the problem of sampling from $\{0, 1 \ldots i+1\}$ according to $\mathcal{D}$. Here, $0 \leq j \leq i$ represent the event that $s = j$ and $i+1$ represent the event that $s > i$. In case $i+1$ is sampled, we repeat the procedure with $0 \leq j \leq i$ representing the event that $s = i+1+j$ and $i+1$ representing the event that $s > 2i+1$ and so on. If we can sample according to $\mathcal{D}$ with R expected number of random bits, then we can clearly sample s according to f_i with $O(R)$ expected random bits. So, we now focus on the problem of sampling according to $\mathcal{D}$. We use the sampling algorithm by Bringmann and Larsen [2] for this (see Sect. 2.1). We will use an array A, which contains elements in $\{0, 1, \ldots i+1\}$. Each element j is present $\lfloor (i+2) \cdot \mathcal{D}(j) \rfloor + 1$ times in A. Now, the sampling algorithm proceeds as follows:

```
1. Pick $k$ uniformly at random from $\{1, \ldots, |A|\}$.
2. If $k = 1$ or $A[k] = A[k-1]$,
   with probability, $(1 - frac((i+2) \cdot \mathcal{D}(A[k])))$ go to step 1.
3. Output $A[k]$.
```

Here, $frac(x)$ represents $x - \lfloor x \rfloor$. Bringmann and Larsen shows that this procedure gives a sample from $\{0, 1, \ldots i+1\}$ according to $\mathcal{D}$. In step 1, we need at most $O(\log i)$ random bits. In step 2, as $\mathcal{D}(j) = \frac{i}{(i+j)(i+j+1)}$, we need only $O(\log i)$ random bits. Also, the steps are repeated a constant number of times in expectation. So, the expected number of random bits required by this procedure is at most $O(\log i)$. This implies that s can be sampled according to f_i using $O(\log i)$ random bits in expectation. Since Vitter shows that the expected number of skips is $O(\log n)$, we get that the total expected number of random bits required by Vitter's algorithm is $O(\log^2 n)$.

In the non-streaming setting, where all the items are available at once in the memory, the expected number of random bits for sampling is $O(\log n)$. So, in the model considered by Vitter, there is still a gap of $\Omega(\log n)$ between the expected number of random bits required in the streaming and non-streaming settings. It is an interesting open question whether this gap should exist. In our model, where we are allowed to err with a small given probability, we show that there is no gap between the streaming and non-streaming settings.

Succinct Sampling The classic solution to the problem of exact sampling from discrete distribution was given by Walker [12]. Kronmal and Peterson [7] improved the preprocessing time of Walker's method. Bringmann and Panagiotou [3] studied some variants of discrete sampling. All of the above works used the Real RAM model of computation. Bringmann and Larsen [2] analyzed Walker's method in the word RAM model. They also gave better algorithms for exact sampling in the word RAM model and proved their optimality. Our work can be thought of as an extension of the ideas in [2] to approximate sampling from discrete distributions.

2 Sampling in the Streaming Setting

The input in this setting consists of a stream of distinct objects denoted as $O_1, O_2, \ldots$, where the object O_i is thought to be arriving at time i. Our objective is to maintain a random sample chosen uniformly at random from the objects seen so far in the stream. More formally, we maintain a random variable X_t for all time t such that $\mathbf{Pr}[X_t = O_i]$ is same for all $i = 1, \ldots, t$. We discussed in the introduction why this objective cannot be met for all values of time t. Therefore, we relax the sampling criteria and allow the sampling algorithm to fail with probability at most ε, where ε is an input parameter. We say the sampling algorithm fails when it returns a null object $\perp$. Therefore, the algorithm is allowed to return $\perp$ with probability at most ε. We want the following property to be true for all time t: $\mathbf{Pr}[X_t = \perp] \leq \varepsilon, \mathbf{Pr}[X_t = O_1] = \mathbf{Pr}[X_t = O_2] = \cdots = \mathbf{Pr}[X_t = O_t]$. Such a sequence of random variables X_t is called *uniform samples* (with error parameter ε, which will be implicit in the discussion).

Streaming algorithms are typically allowed to use poly-logarithmic space. We further constrain our setting by insisting that the algorithm can store only one object at one time and some local variables. The implicit assumption is that the objects in the stream could be arbitrarily large (objects could be large files/packets etc.) and we may not have enough space in the local memory of the program to store more than one object.

Consider the classical setting where all the n items are present in the memory and we need a random sample from these items. It is not difficult to show that $O\left(\log \frac{n}{\varepsilon}\right)$ random bits suffice (w.r.t. uniform sampling with ε-error). In fact, it can be seen easily that any algorithm (even in the non-streaming setting) needs at least these many random bits. We give details of the lower bound on number of random bits in Sect. 2.1. In Sect. 2.2, we show that we can maintain a uniform sample with only $O\left(\log \frac{n}{\varepsilon}\right)$ bits of randomness (till time n). These results may be extended to the weighted case which we discuss in Sect. 2.3.

2.1 Background

We consider the off-line problem of generating a uniform sample with error parameter ε from the set of objects $O_1, \ldots, O_n$.

Lemma 1 *We can generate a uniform sample with error parameter ε from a set of n distinct objects using $O(\log \frac{n}{\varepsilon})$ random bits. Further, any algorithm for generating such a random sample must use $\Omega(\log \frac{n}{\varepsilon})$ random bits.*

Proof Let r be the smallest integer such that $2^r \geq n/\varepsilon$, and let k denote $\left\lfloor \frac{2^r}{n} \right\rfloor$. Now we consider a sequence x of r random bits, and interpret this as a number between 0 and $2^r - 1$. If this number is at least nk, we output $\perp$. Otherwise x is less than nk. Let i be the (unique) integer between 1 and n such that $x \in [(i-1)k, ik)$. In this case, the algorithm outputs the object O_i. Clearly, the probability that the algorithm outputs O_i is $\frac{k}{2^r}$, which is same for all the n objects. The probability that it outputs $\perp$ is

$$\frac{2^r - nk}{2^r} \leq \frac{2^r - n\left(\frac{2^r}{n} - 1\right)}{2^r} = \frac{n}{2^r} \leq \varepsilon.$$

Since r is $O(\log \frac{n}{\varepsilon})$, we have shown the first part of the lemma.

Now we prove the lower bound result. Let R denote the minimum number of required random bits. Clearly, $2^R \geq n$, because there are at least n possible outcomes. Assuming there is at least one sequence of random bits for which the algorithm outputs $\perp$ (recall that for a general n, this will be the case), we get $\varepsilon \geq \frac{1}{2^R}$, which implies $2^R \geq \frac{1}{\varepsilon}$. Thus, $R \geq \frac{1}{2} \log \frac{n}{\varepsilon}$. □

Since we do not know the number of objects in the stream in advance but still need to maintain a random sample at all times, the above upper bound idea does not work in streaming setting. One solution that is known for this problem is reservoir sampling. However, as we have seen, reservoir sampling is costly in terms of the number of random bits used for sampling. In the next section, we design a sampling algorithm in the streaming setting that uses $O(\log \frac{n}{\varepsilon})$ random bits till time n, and hence, matches the lower bound result mentioned above.

2.2 Uniform Samples in the Streaming Setting

Let us try to understand some of the challenges of designing sampling algorithms in the streaming setting. Recall that X_t is the random object maintained by the algorithm at time t. Since the algorithm is allowed to store only one object at any time, it does not store any other object at time t. At time $t+1$, when O_{t+1} arrives, the algorithm has only three choices for X_{t+1}, they are X_t, O_{t+1} or $\perp$. We shall use r_t to denote the number of random bits used by our algorithm till time t. Given a sequence x_t of r_t random bits, let $f_t(x_t)$ denote the object stored by the algorithm at time t, i.e., $X_t = f_t(x_t)$. Note that the functions f_t need to satisfy a "consistency" property: if $x \in \{0,1\}^{r_t}$ is a prefix of a string $y \in \{0,1\}^{r_{t+1}}$, then $f_{t+1}(y)$ is either $f_t(x)$ or O_{t+1} or $\perp$. This is due to the restriction of the streaming setting, in particular because we store only one object at any time. Note that the only stream elements one has access to at time $t+1$ are X_t and O_{t+1}, assuming $X_t \neq \perp$. We now describe our sampling algorithm which we call the *doubling–chopping* algorithm.

2.2.1 The Algorithm

For each time t and $i \in \{1,\ldots,t\} \cup \{\perp\}$, the algorithm will maintain an ordered set $H_i^t \subseteq \{0,1\}^{r_t}$ of strings x for which $f_t(x) = O_i$ (or $\perp$). Note that a naive implementation of these sets would result in huge space overhead, we will show later that these sets can be maintained implicitly. Initially, at time $t=0$, $H_\perp^0 = \emptyset$ and $r_0 = 0$. We first describe the doubling step in Fig. 1. The goal of this step is to ensure that 2^{r_t} stays larger than $\frac{(t+1)^2}{\varepsilon}$. Whenever this inequality is not true, the value of r_t is increased till it is satisfied. The functions f_t are updated accordingly, they just look at the first r_t bits of the input.

Note that after we call the algorithm `Double`, the new $r_t - r_{t-1}$ random bits do not participate in the choice of random sample X_t at time t. In Step 3 of the `Double` algorithm, the set H_i^t is an ordered list—"append" just adds an element to the end of the list.

The next step, called the chopping step, shows how to modify the function f_t so that some probability mass moves towards O_t. The value of r_t remains unchanged during this step. A number of bit strings are moved from H_i^t to H_t^t. The algorithm is described in Fig. 1. The function `append`(T_1, T_2) takes two ordered lists and outputs

```
Algorithm Double(t) :
 1. $r_t \leftarrow r_{t-1}$.
 2. For $i \in \{1,...,t-1\} \cup \{\perp\}$
    • Initialize $H_i^t \leftarrow H_i^{t-1}$.
 3. While $2^{r_t} < \frac{(t+1)^2}{\varepsilon}$
    (i) $r_t \leftarrow r_t + 1$.
    (ii) For $i \in \{1,...,t-1\} \cup \{\perp\}$
       • Initialize $H \leftarrow \emptyset$.
       • For each $x \in H_i^t$ in order
           append $0x$ to $H$.
       • For each $x \in H_i^t$ in order
           append $1x$ to $H$.
       • $H_i^t \leftarrow H$.
```

```
Algorithm Chop(t) :
 1. For every $i \in \{1,\ldots,t-1\}$
    • Define $T_i^t \leftarrow$ last $\left(|H_i^t| - \lfloor\frac{2^{r_t}}{t}\rfloor\right)$ strings
      in $H_i^t$.
    • Define $H_i^t \leftarrow H_i^t \setminus T_i^t$.
 2. Initialize $T \leftarrow \emptyset$.
 3. For $i = 1,\ldots,t-1$
    $T \leftarrow$ append$(T, T_i^t)$.
 4. If $|T| > \lfloor 2^{r_t}/t \rfloor$
     (a) $T_t^t \leftarrow$ last $\left(|T| - \lfloor\frac{2^{r_t}}{t}\rfloor\right)$ strings in $T$.
     (b) $H_t^t \leftarrow T \setminus T_t^t$ and $H_\perp^t \leftarrow$ append$(H_\perp^t, T_t^t)$
    Else
     (i) $T_\perp^t \leftarrow$ last $\lfloor 2^{r_t}/t \rfloor - |T|$ strings of $H_\perp^t$.
     (ii) Set $H_\perp^t \leftarrow H_\perp^t \setminus T_\perp^t$ and
       $H_t^t \leftarrow$ append$(T_\perp^t, T)$.
```

Fig. 1 The doubling and chopping steps

a new list obtained by first taking all the elements in T_1 followed by the elements in T_2 (in the same order). The algorithm maintains the sets H_i^t, where $i \in \{1, \dots, t\} \cup \{\perp\}$. Given these sets, the function f_t is immediate. If the string $x \in \{0,1\}^{r_t}$ lies in the set H_i^t, then $f_t(x) = O_i$.

To summarize, at time $t > 1$, we first call the function `Double`(t) and then the function `Chop`(t). The initial conditions (at time $t = 1$) are $r_1 = \lceil \log \frac{4}{\varepsilon} \rceil$, $H_1^1 = \{0,1\}^{r_1}$, and $H_\perp^1 = \emptyset$. It is easy to check that the functions f_t satisfy the consistency criteria.

Lemma 2 *Suppose* $x \in \{0,1\}^{r_{t-1}}$ *and* $y \in \{0,1\}^{r_t - r_{t-1}}$*. Then,* $f_t(yx)$ *is either* $f_{t-1}(x)$ *or* O_t *or* $\perp$*.*

Proof Let x and y be as above. Suppose $x \in H_i^{t-1}$ (and so, $f_{t-1}(x) = i$). After the call to `Double`(t), $yx \in H_i^t$. Now consider the function `Chop`(t). If $yx \notin H_i^t$ after Step 1, then it must be the case that yx gets added to the set T. Now, notice that the strings in T get added to either H_t^t or $H_\perp^t$. This proves the lemma. □

The lemma above implies that we can execute the algorithm by storing only one object at any time. Now, we show that the number of random bits used by the algorithm is small.

Lemma 3 *The number of random bits used by the algorithm till time n is* $O(\log \frac{n}{\varepsilon})$*.*

Proof Till time n, the algorithm uses at most r_n bits and $2^{r_n} \le \frac{2(n+1)^2}{\varepsilon}$. □

The correctness of the algorithm follows from the next lemma.

Lemma 4 *For all time* $t > 0$*, and* $i \in \{1, \dots, t\}$*,* $|H_i^t| = \left\lfloor \frac{2^{r_t}}{t} \right\rfloor$*, and* $|H_\perp^t| \le \varepsilon \cdot 2^{r_t}$*.*

Proof The proof is by induction on t. The base case ($t = 1$) is true from the initial conditions $r_1 = \lceil \log \frac{4}{\varepsilon} \rceil$, $H_1^1 = \{0,1\}^{r_1}$, and $H_\perp^1 = \emptyset$. Now suppose the lemma is true for $t - 1$. At time t, we first call `Double`(t). For each $x \in H_i^{t-1}$, we just append all bit strings of length $r_t - r_{t-1}$ to it and this set of strings to H_i^t. Therefore, when this procedure ends, $|H_i^t| = 2^{r_t - r_{t-1}} \cdot \lfloor \frac{2^{r_{t-1}}}{t-1} \rfloor$, for $i = 1, \dots, t-1$ (using induction hypothesis) and we have

$$|H_i^t| = 2^{r_t - r_{t-1}} \cdot \left\lfloor \frac{2^{r_{t-1}}}{t-1} \right\rfloor \ge 2^{r_t - r_{t-1}} \cdot \left(\frac{2^{r_{t-1}}}{t-1} - 1 \right) \ge \frac{2^{r_t}}{t} \quad (\text{since } 2^{r_{t-1}} \ge t^2/\varepsilon)$$

In Step 1 of the procedure `Chop`(t), we ensure that $|H_i^t|$ becomes $\lfloor \frac{2^{r_t}}{t} \rfloor$ (this step can be done, because the $|H_i^t|$ was at least $\lfloor \frac{2^{r_t}}{t} \rfloor$). After this step, we do not change H_i^t for $i = 1, \dots, t-1$, and hence, the induction hypothesis is true for these sets. It remains to check the size of H_t^t and $H_\perp^t$.
First assume that $|T| \ge \lfloor \frac{2^{r_t}}{t} \rfloor$. In this case, H_t^t gets exactly $\lfloor \frac{2^{r_t}}{t} \rfloor$ elements. Now suppose $|T| < \lfloor \frac{2^{r_t}}{t} \rfloor$. First observe that $H_\perp^t$ and T are disjoint. Since all strings not in $H_i^t, i = 1, \dots, t-1$ belong to either $H_\perp^t$ or T, it follows that

$$|H_\perp^t| + |T| = 2^{r_t} - (t-1) \cdot \left\lfloor \frac{2^{r_t}}{t} \right\rfloor \ge \left\lfloor \frac{2^{r_t}}{t} \right\rfloor.$$

```
Double(t)
  1. If $(t = 1)$
       • Set $s \leftarrow 2^{\lceil \log \frac{4}{\epsilon} \rceil}$
       • Set $h \leftarrow s$
       • Set $h_\perp \leftarrow 0$
       • Set $l \leftarrow 1$
       • Set rank $\leftarrow$ random$(s)$
  2. While $(s < \frac{(t+1)^2}{\epsilon})$
       • Set $b \leftarrow$ random_bit()
       • If $(b = 1)$ and $(l = \perp)$
           ◇ Set rank $\leftarrow$ rank $+ h_\perp$
       • If $(b = 1)$ and $(l \neq \perp)$
           ◇ Set rank $\leftarrow$ rank $+ h$
       • Set $h \leftarrow 2h$
       • Set $h_\perp \leftarrow 2h_\perp$
       • Set $s \leftarrow 2s$
```

```
Chop(t)
  1. If $(t = 1)$
       • Exit
  2. Set $h' \leftarrow \lfloor \frac{s}{t} \rfloor$
  3. Set $h_t \leftarrow (t-1)(h-h')$
  4. If $(l \neq \perp)$ and (rank $> h'$)
       • Set rank $\leftarrow (l-1)(h-h')+($rank$-h')$
       • Set $l \leftarrow t$
  5. If $(h_t > h')$
       • Set $h_{\perp'} \leftarrow h_\perp + h_t - h'$
       • If $(l = t)$ and (rank $> h'$)
           ◇ Set $l \leftarrow \perp$
           ◇ Set rank $\leftarrow h_\perp +$ rank $- h'$
  6. If $(h_t \leq h')$
       • Set $h_{\perp'} \leftarrow h_\perp + h_t - h'$
       • If $(l = \perp)$ and (rank $> h_{\perp'}$)
           ◇ Set $l \leftarrow t$
           ◇ Set rank $\leftarrow h_t +$ rank $- h_{\perp'}$
  7. Set $h \leftarrow h'$
  8. Set $h_\perp \leftarrow h_{\perp'}$
```

Fig. 2 Space efficient pseudocode for the doubling and chopping algorithms

Therefore, $|H_\perp^t|$ is at least $\lfloor \frac{2^{r_t}}{t} \rfloor - |T|$, and Step 4(i) in this case can be executed. Clearly, $|H_t^t|$ becomes $\lfloor \frac{2^{r_t}}{t} \rfloor$ as well. Finally,

$$|H_\perp^t| = 2^{r_t} - t \cdot \left\lfloor \frac{2^{r_t}}{t} \right\rfloor \leq 2^{r_t} - t\left(\frac{2^{r_t}}{t} - 1\right) = t \leq \varepsilon \cdot 2^{r_t},$$

where the last inequality follows from the definition of r_t. □

2.2.2 Space Complexity

Note that the use of the sets H_i^t in our algorithm was just to provide the intuition. We only need to maintain the heights of these sets. This involves only storing two quantities h and $h_\perp$, where at any point of time t we maintain $h = H_1^t = H_2^t = \cdots = H_t^t$ and $h_\perp = H_\perp^t$. We will also need to maintain the *location*, l of the current random string(means that the random string belongs to H_l^t) and the *rank* of the current random string in H_l^t. Since each H_i^t is an ordered list, *rank* is the position in this order. We give a space efficient implementation of the `Double` and `Chop` methods in Fig. 2. [4] We assume that the function `random`(y) returns a random integer from $[y]$ and the function `random_bit`() returns a random bit. `random`(y) may be easily implemented using `random_bit`() when y is a power of 2 which is indeed the case here.

[4] A python implementation of this pseudocode can be found at http://www.cse.iitd.ac.in/~rjaiswal/Research/Sampling/sampling.py.

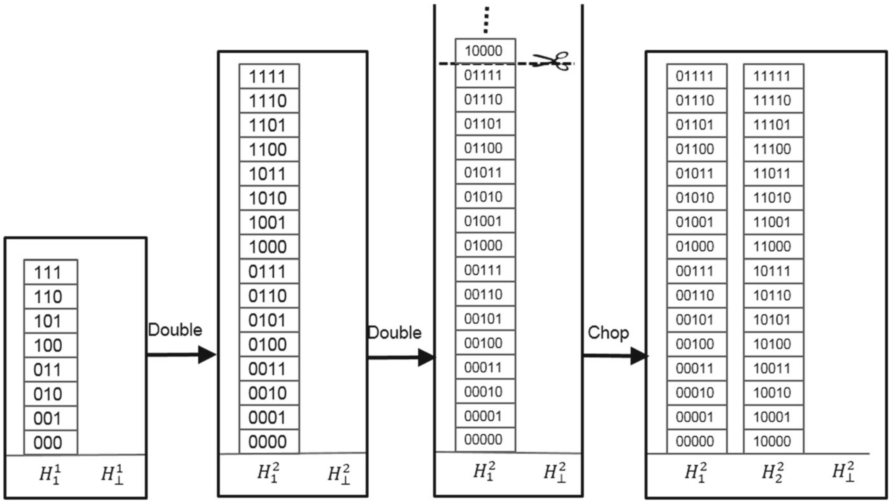

Fig. 3 The figure shows simulation of the doubling–chopping algorithm at time $t = 2$ when the value of $\varepsilon = 1/2$

Lemma 5 *For every time t, the location and the rank of the current random string x_t can be maintained using $O(\log \frac{t}{\varepsilon})$ bits of space.*

Proof The space efficient `Double` and `Chop` methods given in Fig. 2 only stores $s, h, h_\perp, l$ and $rank$, each of which is at most 2^{r_t} at any time t. Hence, we only need $O(r_t) = O(\log \frac{n}{\varepsilon})$ bits of space. □

2.2.3 Running Time

Finally, we analyze the running time requirement of the algorithm after n time steps have been completed. We will analyze using the pseudocode given in Fig. 2. It is easy to see that `Chop` only requires constant number of operations. Also, all the operations inside the `While` in `Double` are constant operations. The total number of iterations of `While` loop across all invocations of `Double` until time step n is at most r_n, i.e., $O(\log \frac{n}{\varepsilon})$. Hence, the total time required after n steps is $O(\log \frac{n}{\varepsilon})$. It is easy to see that $r_2 \le r_1 + 3, r_3 \le r_2 + 3$ and for $t \ge 3, r_{t+1} \le r_t + 2$. So, the `While` loop is executed only a constant number of times during each call to *Double*. Hence, our algorithm requires $O(1)$ updation time per item. [5]

Figure 3 shows the operations performed by the sampling algorithm after the arrival of the second item. The techniques in this section generalize to weighted sampling discussed next.

[5] Note that this is under the common word RAM assumption that arithmetic operations on $O(\log n)$ bit words can be done in $O(1)$ time.

2.3 Weighted Sampling

In Sect. 2, we discussed uniform sampling. We can extend our sampling algorithm to the case where object i in the stream is associated with integer weight w_i and the sampling algorithm is required to output object i with probability proportional to w_i. Further, as in the uniform sampling case, the algorithm is allowed to output $\perp$ with probability at most ε. More specifically, let the algorithm output $\perp$ with probability p. Then $p \leq \varepsilon$, and the probability that it outputs object i is given by $(1-p) \cdot \frac{w_i}{\sum_i w_i}$. The most obvious way to extend our uniform sampling algorithm to the weighted case is to consider w_i copies of object i and simulate our sampling algorithm. The number of required random bits would be $\log\left(\frac{\sum_i w_i}{\varepsilon}\right)$. Assuming each of the weights $w_1, \ldots, w_n$ to be a w-bit integer, the upper bound becomes $O(\log \frac{n \cdot 2^w}{\varepsilon}) = O(w + \log \frac{n}{\varepsilon})$. From Lemma 1, we know that the lower bound is $\Omega(\log \frac{n}{\varepsilon})$ when $w_1 = w_2 = \cdots = 1$. Furthermore, given that $w_1 = 1$ and $w_2 = 2^w - 1$, any sampling algorithm would need at least w bits for uniform sampling. This gives another lower bound of $\Omega(w)$. From the last two statements, we get that the lower bound on the number of random bits required for weighted sampling is $\Omega(w + \log \frac{n}{\varepsilon})$ which matches with our upper bound. In this setting, simple space/time optimizations lead to a sampling algorithm with running time $O(n+w+\log \frac{1}{\varepsilon})$ (with per item time $O(w)$) and space $O(w+\log \frac{n}{\varepsilon})$ bits.

3 Succinct (Approximate) Sampling

In this section, we consider the approximate sampling problem in the succinct data-structure model. We are given as input a set of n elements labelled $1, \ldots, n$ and weights $x_1, \ldots, x_n$ associated with these elements. Each of these weights x_i is assumed to be a w-bit integer where the assumption is $w = o(n)$. The assumption is reasonable since typically w is a single precision ($w = 32$) or double precision ($w = 64$) number. Let p_i denote $x_i/(\sum_j x_j)$. Given an error parameter ε, we consider additive and multiplicative approximate sampling. More formally, in the multiplicative model, the probability of returning element i as output lies in the range $[p_i(1-\varepsilon), p_i(1+\varepsilon)]$. In the additive model, the respective probability lies in the range $[p_i - \varepsilon, p_i + \varepsilon]$. We are also allowed suitable representation of these weights so as to facilitate efficient sampling.

3.1 Approximate Sampling: Multiplicative Model

In this section we consider approximate sampling with (multiplicative) error ε and give upper and lower bounds of space usage in this model. We first describe the upper bound by giving our sampling algorithm. Next we provide matching lower bounds. For simplicity, we assume that ε is a power of 2 (this only affects the bounds by a constant factor). The following definition of closeness of distributions will be useful in the analysis.

Definition 1 We say that a distribution given by $(y_1, \ldots, y_n)$ is ε-close to a distribution given by $(x_1, \ldots, x_n)$ if $\forall i$, $(1-\epsilon) \cdot \frac{x_i}{\sum_j x_j} \leq \frac{y_i}{\sum_j y_j} \leq (1+\epsilon) \cdot \frac{x_i}{\sum_j x_j}$. In such a case, $(y_1, \ldots, y_n)$ may be used to (approximately) represent the distribution $(x_1, \ldots, x_n)$.

Upper Bound For each i, let f_i denote the location of the most significant bit (MSB) in x_i which is 1 (i.e., the first $f_i - 1$ bits of x_i are 0). Let x'_i denote the number obtained by taking the first $f_i + \log \frac{2}{\epsilon}$ bits of x_i followed by $\left(w - f_i - \log \frac{2}{\varepsilon}\right)$ many 0's. It is fairly easy to check that exact sampling with respect to the weights x'_i leads to approximate sampling with respect to x_i with error at most ε. This is because the distribution given by $(x'_1, \ldots, x'_n)$ is ε-close to the distribution given by $(x_1, \ldots, x_n)$.

Lemma 6 *For all i, $(1-\varepsilon) \cdot p_i \leq \frac{x'_i}{\sum_j x'_j} \leq (1+\varepsilon) \cdot p_i$.*

Proof Observe that for all i, $x_i - x'_i \leq \frac{\varepsilon}{2} \cdot x_i$, which implies that $x_i \geq x'_i \geq (1-\varepsilon/2) \cdot x_i$. Using this fact, we get $(1-\varepsilon)p_i \leq \frac{x_i(1-\varepsilon/2)}{\sum_j x_j} \leq \frac{x'_i}{\sum_j x_j} \leq \frac{x'_i}{\sum_j x'_j} \leq \frac{x_i}{(1-\varepsilon/2)\sum_j x_j} \leq (1+\epsilon)p_i$. □

Therefore, it is enough to run an exact sampling algorithm with weights x'_i for all element i. We use the exact sampling algorithm of Bringmann and Larsen [2] for this purpose with weights x'_i for element i. The space usage of this algorithm is $O(n + w')$ bits in addition to the space required for storing input data, where w' denotes the required number of bits to store any of these weights. In our case, w' can be as high as w, and so the space used by the algorithm is $O(n + w)$ bits in addition to the space required for storing input data. There is one catch though: we need to store all the x'_i using the same number of bits, and using w bits would be a waste of space. Instead we store each x'_i as a tuple—we first store the value of f_i and then the value of the next $\log \frac{2}{\epsilon}$ bits. Note that this representation uses $(\log w + \log \frac{2}{\epsilon})$ bits for each x'_i. It is not difficult to check that the algorithm of Bringmann and Larsen [2] works with this representation as well. Thus, the total space needed by our algorithm is $O(n \log w + n \log \frac{2}{\epsilon})$ bits.

Lower Bound In this section, we derive lower bound on the amount of space needed for approximately sampling the elements with error ε. To get a lower bound on the space, we will estimate the size of a set of tuples $S \subseteq [\{0,1\}^w]^n$ such that for any tuple $\bar{x} \in [\{0,1\}^w]^n$, there exists at least one element $\bar{y}$ in S such that $\bar{y}$ is ε-close to $\bar{x}$. Let $\mathcal{S}$ denote the minimum amount of space needed in bits. We get a lower bound on $\mathcal{S}$ using the next two lemmas.

Lemma 7 $\mathcal{S} \geq n \log \frac{1}{\epsilon} - w - \log n - n$.

Proof Let $U \subseteq [\{0,1\}^w]^n$ denote a universe of n tuples of w-bit numbers such that for any $(u_1, \ldots, u_n) \in U$, $\sum_i u_i = T$, where T will be specified later. Given a tuple $\bar{x} = (x_1, \ldots, x_n) \in U$, $\texttt{Ball}(\bar{x})$ denotes the set of all tuples $\bar{y} \in U$ such that $\bar{y}$ is ε-close to $\bar{x}$. Recall that this implies that for all $i = 1, \ldots, n$,

$$(1-\varepsilon)x_i \leq y_i \leq (1+\varepsilon)x_i.$$

So, y_i can have at most $2\varepsilon x_i$ different values around x_i. This gives the following:

$$|\texttt{Ball}(\bar{x})| \leq (2\varepsilon)^n \cdot x_1 \cdot x_2 \cdots x_n \leq 2^{wn} \cdot (2\varepsilon)^n, \tag{1}$$

where the last inequality follows from the fact that each x_i is a w-bit number. For any tuple $\bar{t} \in [\{0,1\}^w]^n$, the sum of elements in the tuple belongs to the set $\{0, 1, \ldots, n \cdot (2^w - 1)\}$. This means that there is one value $T' \in \{0, 1, \ldots, n \cdot (2^w - 1)\}$ such that the number of tuples whose sum is equal to T' is at least $\frac{2^{nw}}{n\cdot(2^w-1)+1} \geq \frac{2^{nw}}{n \cdot 2^w}$. We will use $T = T'$. This implies that $|U| \geq \frac{2^{nw}}{n \cdot 2^w}$. Combining this fact with inequality (1), we get

$$2^{\mathcal{S}} \geq \frac{|U|}{2^{wn} \cdot (2\varepsilon)^n} \geq \frac{1}{n \cdot 2^w \cdot (2\varepsilon)^n}$$

This gives $\mathcal{S} \geq n \log \frac{1}{\varepsilon} - w - n - \log n$. □

Lemma 8 $\mathcal{S} \geq n \log w - n \log 4(1 + 2\epsilon) - \frac{w}{2} \log (e^2 n)$.

Proof Let $U \subseteq ([\{0,1\}^w])^n$ denote the subset of n-tuples of w-bit numbers $\bar{x}$ of the following form: it should be possible to divide the n coordinates in $\bar{x}$ into w blocks, each block consisting of n/w coordinates (note that these coordinates need not be consecutive). Let the set of indices for block l is denoted by B_l. For any index $i \in B_l$, the first $(l-1)$ bits are 0, and the lth bit is 1. The remaining bits of index i can be arbitrary though. Consider any tuple $\bar{x} = (x_1, \ldots, x_n) \in U$. Define $\texttt{Ball}(\bar{x})$ as the set of all tuples $\bar{y} \in U$ which are ε-close to $\bar{x}$, i.e., for all $i = 1, \ldots, n$,

$$(1 - \varepsilon) \cdot \frac{x_i}{S} \leq \frac{y_i}{S'} \leq (1 + \varepsilon) \cdot \frac{x_i}{S}, \tag{2}$$

where $S = \sum_j x_j$ and $S' = \sum_j y_j$. Let $S_{\min} = \min_{\bar{y} \in \texttt{Ball}(\bar{x})} \sum_i y_i$ and $S_{\max} = \max_{\bar{y} \in \texttt{Ball}(\bar{x})} \sum_i y_i$. Then for all i we have

$$\frac{S_{\min}}{S} \cdot (1 - \varepsilon) \cdot x_i \leq y_i \leq \frac{S_{\max}}{S} \cdot (1 + \varepsilon) \cdot x_i.$$

Therefore, number of possible values of y_i is upper bounded by

$$\frac{x_i}{S} \cdot ((S_{\max} - S_{\min}) + \varepsilon(S_{\max} + S_{\min})) \leq (1 + 2\varepsilon) \cdot \frac{x_i \cdot S_{\max}}{S}.$$

Using this, we get that

$$|\texttt{Ball}(\bar{x})| \leq \left(\frac{S_{\max}}{S}\right)^n \cdot (x_1 \cdots x_n) \cdot (1 + 2\varepsilon)^n \tag{3}$$

We will now try to get an upper bound on $|\texttt{Ball}(\bar{x})|$ by obtaining suitable bounds on the quantities on the RHS of the above inequality. First, note that due to the nature of the tuples under consideration, we have:

$$S \geq \sum_{i=1}^{w} \frac{n}{w} \cdot 2^{w-i} = \frac{n}{w} \cdot (1 + 2 + \cdots + 2^{w-1}) = \frac{n}{w} \cdot (2^w - 1).$$

Furthermore, for any $\bar{y} \in U$, we have

$$S_{\max} \leq \sum_{i=1}^{w} \frac{n}{w}(2^{w-i+1} - 1) = \frac{n}{w}(2^{w+1} - 2 - w) \leq 2 \cdot \frac{n}{w} \cdot (2^w - 1)$$

Next we upper bound the product of $x_1, \ldots, x_n$. Since, each number in ith group is $< 2^{w-i+1}$, we can write,

$$x_1 \cdots x_n < \prod_{i=1}^{w} (2^{w-i+1})^{n/w} = 2^{n(w+1)/2}$$

Putting these bounds in inequality (3), we get that

$$|\texttt{Ball}(\bar{x})| \leq 2^n \cdot 2^{n(w+1)/2} \cdot (1 + 2\varepsilon)^n$$

Now, we try to get an estimate on $|U|$. The number of ways w blocks can be arranged is $\frac{n!}{(\frac{n}{w}!)^w}$. We now use the following Stirling's approximation of $a!$ for any positive integer a:

$$\sqrt{2\pi} a^{a+1/2} e^{-a} \leq a! \leq e a^{a+1/2} e^{-a},$$

to get

$$\frac{n!}{(\frac{n}{w}!)^w} \geq \sqrt{2\pi} \left(\frac{w}{e^2 n}\right)^{\frac{w}{2}} n^{1/2} w^n.$$

So, we get

$$\begin{aligned} |U| &\geq \left(\sqrt{2\pi} \left(\frac{w}{e^2 n}\right)^{\frac{w}{2}} n^{1/2} w^n\right) \cdot \prod_{i=1}^{w} (2^{w-i})^{\frac{n}{w}} \\ &= \left(\sqrt{2\pi} \left(\frac{w}{e^2 n}\right)^{\frac{w}{2}} n^{1/2} w^n\right) \cdot 2^{n(w-1)/2} \end{aligned}$$

Using this bound, we have

$$\begin{aligned}
2^{\mathcal{S}} &\geq \frac{|U|}{2^n \cdot 2^{n(w+1)/2} \cdot (1+2\varepsilon)^n} \\
&\geq \frac{\left(\sqrt{2\pi}\left(\frac{w}{e^2 n}\right)^{\frac{w}{2}} n^{1/2} w^n\right) \cdot 2^{n(w-1)/2}}{2^n \cdot 2^{n(w+1)/2} \cdot (1+2\epsilon)^n} \\
&= \frac{1}{2^{2n}} \cdot \sqrt{2\pi n} \cdot w^n \cdot \left(\frac{w}{e^2 n}\right)^{w/2} \cdot \left(\frac{1}{1+2\varepsilon}\right)^n
\end{aligned}$$

which implies that

$$\begin{aligned}
\mathcal{S} &\geq n \log w + \log \sqrt{2\pi n} + \frac{w}{2} \log \frac{w}{e^2 n} + n \log \frac{1}{4(1+2\varepsilon)} \\
&\geq n \log w - n \log 4(1+2\varepsilon) - \frac{w}{2} \log (e^2 n)
\end{aligned}$$

This conclude the proof of the lemma. □

Comparing Upper Bounds with Lower Bounds The upper bound that we obtained on the space requirement was $(n \log \frac{2}{\epsilon} + n \log w)$ bits. We break the comparison into the following two parts:

1. $\frac{1}{\epsilon} > w$: In this case, the upper bound is $O(n \log \frac{1}{\epsilon})$ bits. Using Lemma 7, we get that the lower bound is $\Omega(n \log \frac{1}{\epsilon})$ bits assuming $w = o(n)$.
2. $\frac{1}{\epsilon} \leq w$: In this case, the upper bound is $O(n \log w)$ bits. Using Lemma 8, we get that the lower bound is $\Omega(n \log w)$ bits assuming $w = o(n)$.

So, we obtain matching lower and upper bounds assuming $w = o(n)$.

3.2 Approximate Sampling: Additive Model

We discuss the additive error model in this section. In this model, given error parameter ε, the probability of sampling element i lies in the range $[p_i - \varepsilon, p_i + \varepsilon]$. We first describe the upper bound by giving our algorithm. Next we give matching lower bounds. Again, we assume wlog that $\frac{1}{\varepsilon}$ is an integer. Let S denote $\sum_j x_j$.

Upper Bound We create a sorted array A of size $\frac{1}{\varepsilon}$ which stores copies of numbers from 1 to n as follows: for each i, first we store $\lfloor \frac{1}{\varepsilon} \cdot \frac{x_i}{S} \rfloor$ many copies of i; then we pick $\frac{1}{\varepsilon} - \sum_i \lfloor \frac{1}{\varepsilon} \cdot \frac{x_i}{S} \rfloor$ elements arbitrarily from $[n]$ and add an extra copy of each. To generate a random element, the algorithm picks a uniformly random location in A and outputs the number stored in that location in A. Clearly, the probability of sampling i lies in the range $\left[\varepsilon \left\lfloor \frac{1}{\varepsilon} \cdot \frac{x_i}{S} \right\rfloor, \varepsilon \left(\left\lfloor \frac{1}{\varepsilon} \cdot \frac{x_i}{S} \right\rfloor + 1\right)\right] \subseteq \left[\varepsilon\left(\frac{1}{\varepsilon} \cdot \frac{x_i}{S} - 1\right), \varepsilon\left(\frac{1}{\varepsilon} \cdot \frac{x_i}{S} + 1\right)\right] = [p_i - \varepsilon, p_i + \varepsilon]$, which is what we need. The space usage by the algorithm is the space required to store array A, i.e., $O\left(\frac{1}{\epsilon} \log n\right)$ bits.

Lower Bound We now prove the lower bound result. We come up with a set of distributions such that each pair of them differ by more than ε on at least one coordinate.

Consider the following set of n-tuples: $(x_1 \cdot \varepsilon, x_2 \cdot \varepsilon, \ldots, x_n \cdot \varepsilon)$ where $x_1, \ldots, x_n$ are non-negative integers such that $\sum_i x_i = \frac{1}{\varepsilon}$. If we pick any two such distinct vectors, they will differ on at least one coordinate by at least ε. Clearly, the size of the set of such possible vectors (or distributions) is at least $\binom{\frac{1}{\varepsilon}+n-1}{n}$. Therefore, the space needed for sampling with ε additive error is at least $\log \binom{\frac{1}{\varepsilon}+n-1}{n} \geq \Omega\left(\frac{1}{\varepsilon}\log n\right)$ bits, provided ε is some constant independent of n.

Lower bound for smaller ε is discussed in the following manner. As shown above, the lower bound on space is given by the expression:

$$\mathcal{S} \geq \log \binom{1/\varepsilon + n - 1}{n} = \Omega\left(\frac{1}{\varepsilon} \cdot \log(1 + \varepsilon n) + n \log\left(1 + \frac{1}{\varepsilon n}\right)\right).$$

So, we get the following lower bounds in the following two cases:

1. $\varepsilon \geq 1/n$: In this case, we get that $\mathcal{S} = \Omega\left(\frac{1}{\varepsilon} \cdot \log \varepsilon n\right)$.
2. $\varepsilon < 1/n$: In this case, we get that $\mathcal{S} = \Omega\left(n \cdot \log \frac{1}{\varepsilon n}\right)$.

Matching these lower bounds for small ε is left as an open problem.

4 Conclusions and Open Problems

We introduced sampling with ε-error and justified why it is the right model for the streaming setting. We achieved matching upper and lower bounds for uniform sampling with ε-error in the streaming setting. The upper bound is achieved by the doubling–chopping algorithm which is also efficient in terms of space usage and running time. The algorithm can also be extended to weighted sampling with ε-error in the streaming setting and is still optimal in terms of the amount of random bits used. The doubling–chopping algorithm in fact matches the lower bound for the number of random bits required even in the non-streaming setting. Thus we show that there is no gap between the classical and streaming settings as far as number of random bits is concerned.

We initiated the study of space complexity of approximate sampling with multiplicative and additive errors. We gave tight upper and lower bounds for the space required for approximate sampling with ε multiplicative error. For approximate sampling with ε additive error, we gave upper and lower bounds which are tight only when ε is a constant independent of the number of items.

The major question left open is to find matching upper and lower bounds for the space required for approximate sampling with ε additive error, when ε is not a constant, but is dependent on the number of items.

Acknowledgements RJ and AK would like to thank Karl Bringmann for discussion on Succinct Sampling. They would also like to thank the Indo-German IMPECS program for making the interaction possible.

References

1. Babcock, B., Datar, M., Motwani, R.: Sampling from a moving window over streaming data. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA ’02, pp. 633–634, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics (2002)
2. Bringmann, K., Larsen, K.G.: Succinct sampling from discrete distributions. In: Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing. STOC ’13, pp. 775–782. NY, USA, ACM, New York (2013)
3. Bringmann, K., Panagiotou, K.: Efficient sampling methods for discrete distributions. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) Automata. Languages, and Programming, volume 7391 of Lecture Notes in Computer Science, pp. 133–144. Springer, Berlin Heidelberg (2012)
4. Efraimidis, P.S., Spirakis, P.G.: Weighted random sampling with a reservoir. Inf. Process. Lett. **97**(5), 181–185 (2006)
5. Jaiswal, R., Kumar, A., Sen, S.: A simple D^2-sampling based PTAS for k-means and other clustering problems. Algorithmica **70**(1), 22–46 (2014)
6. Knuth, D.E.: The Art of Computer Programming, vol. 2. Addison-Wesley, Boston (1981)
7. Kronmal, R.A., Peterson Jr., A.V.: On the alias method for generating random variables from a discrete distribution. Am. Stat. **33**(4), 214–218 (1979)
8. Li, K.-H.: Reservoir-sampling algorithms of time complexity $o(n(1 + \log N/n))$. ACM Trans. Math. Softw. **20**(4), 481–493 (1994)
9. Park, B.-H., Ostrouchov, G., Samatova, N.F.: Sampling streaming data with replacement. Comput. Stat. Data Anal. **52**(2), 750–762 (2007)
10. Vitter, J.S.: Faster methods for random sampling. Commun. ACM **27**(7), 703–718 (1984)
11. Vitter, J.S.: Random sampling with a reservoir. ACM Trans. Math. Softw. **11**(1), 37–57 (1985)
12. Walker, A.J.: New fast method for generating discrete random numbers with arbitrary frequency distributions. Electron. Lett. **10**(8), 127–128 (1974)