Lecture 1

1 Introduction

The topic of this class is computation over data streams, and its various manifestations. A data stream is a sequence of data elements (numbers, geometric points, etc), which is much larger than the amount of available memory. The goal is to compute (perhaps approximately) some function of the data using only one pass\(^1\) over the data stream. The difficulty in designing data stream algorithms comes from the fact that any data element that has not been stored is essentially lost forever. As a result, one must exercise care to make sure that the “important” data elements are properly selected and preserved.

Data streams occur in many applications. For example, a network router must process terabits of packet data, which cannot be all stored by the router. At the same time, there are various statistics and patterns of the network traffic that are useful to know, e.g., in order to be able to detect unusual network behavior. Data stream algorithms enable computing such stats quickly and using very little memory.

Other application areas include: databases, sensor networks, etc. See [Mut03] for more on motivation and applications.

2 Model, assumptions and notation

The stream consists of data “elements”. More often than not, an element will be simply an (integer) number from some range. However, it is often convenient to allow other data types, such as: multidimensional points, metric points, graph vertices and edges, etc. Of course, all of these can be translated into numbers from a proper universe, but this requires specifying various nitty-gritty details (such as what is the precision of the point coordinates), so it is often more convenient to sweep this under the carpet. For now, we just restrict our interest to streams of integers, from the range \{0, \ldots, m\}.

The amount of space used by our algorithms will of course vary from problem to problem. Often, we will be able to achieve \(\log^c n\) space bounds for some constant \(c > 0\), but we will be content with something like \(n^a\) for \(a < 1\) as well. In general we will express the amount of storage used by the algorithm in “units”, not bits. A “unit” could be a number from some universe, or a multi-dimensional point, etc (see the previous paragraph). Again, at the end of the day one can translate all of this into bits.

We will denote the number of stream elements, i.e., the length of the stream, by \(n\). There is a great debate in the streaming community if the algorithm can be assumed to know the value of \(n\). In this class, we will assume that it is the case, i.e., the algorithm knows \(n\). More often than not, this is not a restrictive assumption. Also, it can be often removed by running a logarithmic

\(^1\)Later in the class we will also investigate algorithms that make multiple passes over the data.

\(^2\)Recently, researchers discovered very interesting connections between data stream algorithms and problems in signal processing. We will cover that as well.
number of algorithms in parallel, each assuming that the value of \( n \) is in the range \( 2^i \cdots 2^{i+1} - 1 \). However, this can multiply the space usage by a logarithmic factor.

3 Number of distinct elements

The first problem that we consider is that of estimating the number of distinct elements in the stream. Recall we have a stream of \( n \) elements, where each element is an integer from \{0, \ldots, m\}.

Without the memory restrictions, one could solve this problem as follows. In the first step, allocate an array \( x[0\ldots m] \) initialized to 0. Then, for each element \( e \), increment \( x[e] \). At the end of the process, the total number of non-zero entries in the vector \( x \) provides the exact number of distinct elements. Unfortunately, the algorithm uses \( \Theta(m) \) units of space.

Our algorithm will use a similar approach. However, we do not represent the vector \( x \) explicitly. Instead, we maintain a “sketch” of the vector \( x \). The sketch can be efficiently maintained under increments of \( x \), but it can be stored in small amount of space. There are several ways of computing such a sketch. Here we will see a method that is inspired by the algorithm of [FM85]. The version presented here is not the most efficient one, but it provides a good introduction to the rest of the material.

First, a little bit of notation. Let \( DE \) be the number of distinct elements in our stream. Our algorithm will be randomized and approximate. That is, it will have parameters \( \epsilon > 0 \) and \( P \in (0, 1) \), and the goal will be to compute an estimate \( E \) such that \((1 - \epsilon)DE \leq E \leq (1 + \epsilon)DE \) with probability \( 1 - P \). As we will see, allowing the algorithm to return an approximate answer will enable us to reduce the total amount of space to polynomial in \( \log n \) (and \( 1/\epsilon \) and \( \log(1/P) \)).

In the first step, we consider a decision version of the problem. That is, we will supply another parameter \( T \in \{1\ldots m\} \). The goal of the decision algorithm is to perform the following task with probability \( 1 - P \):

- If \( DE \geq T(1 + \epsilon) \), then answer YES
- if \( DE \leq T(1 - \epsilon) \), then answer NO

Note that if \( T(1 - \epsilon) \leq DE \leq T(1 + \epsilon) \) then the algorithm is allowed to report either YES or NO. Such problems are often called promise problems.

Observe that if we solve the decision problem using space \( S \) with probability \( 1 - P \), then we can also solve the original problem using space \( O(S \log(n)/\epsilon) \) with probability \( 1 - O(P \log(m)/\epsilon) \). This is done by running \( d = O(\log(m)/\epsilon) \) copies of the decision algorithm, with parameters \( T = 1, (1 + \epsilon), (1 + \epsilon)^2, \ldots \). The probability that all of them return correct answers is at least \( 1 - dP \).

Given correct answers to all the decision problems, we can easily locate the approximate value of \( DE \) by returning the largest \( T \) such that the decision algorithm outputs YES.

Now we can focus on the decision problem. Our algorithm will make use of the (implicit) count vector \( x \) defined above. We will start from an algorithm whose probability of correctness will be somewhat low, and then we will amplify it to \( 1 - P \).

Algorithm A:

1. Select a random set \( S \subset \{1\ldots m\} \) containing each \( i \) independently with probability \( 1/T \).
2. Make a pass over the stream, maintaining \( \text{Sum}_S(x) = \sum_{i \in S} x_i \)
3. If \( \text{Sum}_S(x) > 0 \), return YES, otherwise return NO

Note that the algorithm does not need to store the set \( S \) explicitly. Instead, we can use a psuedorandom hash function \( h : \{1 \ldots m\} \rightarrow \{1 \ldots T\} \), such that each value \( h(i) \) is distributed uniformly over \( \{1 \ldots T\} \), and have \( i \in S \) if and only if \( h(i) = 1 \).

Intuitively, \( \mathbb{E} [\text{Sum}_S(x)] = \text{DE}/T \). Thus if \( \text{DE} \gg T \) it is very unlikely that \( \text{Sum}_S(x) = 0 \) and if \( \text{DE} \ll T \) it is very unlikely that \( \text{Sum}_S(x) > 0 \). Quantitatively, we need the following lemma:

**Lemma 1.** Consider \( \Pr = \Pr[\text{Sum}_S(x) = 0] \). If \( T > \frac{1}{\epsilon^2} \), then for sufficiently small \( \epsilon \) we have

- If \( \text{DE} \geq T(1 + \epsilon) \), then \( \Pr < \frac{1}{e} - \epsilon/3 \)
- If \( \text{DE} \leq T(1 - \epsilon) \), then \( \Pr > \frac{1}{e} + \epsilon/3 \)

**Proof.** Clearly, we have \( \Pr = (1 - 1/T)^{\text{DE}} \), which can be approximated by \( e^{-\text{DE}/T} \) for large \( T \), which can be well approximated by its local linearization \( 1 + 1/e - \text{DE}T \) for \( \text{DE} \approx T \). If we are willing to accept this approximation, the lemma follows immediately.

More formally, notice that \( \Pr \) is monotonically decreasing in \( \text{DE} \), so it is sufficient to verify the given inequality for \( \text{DE} = cT \) for \( c \in \{1 - \epsilon, 1 + \epsilon\} \). The Taylor series for \( \log(1 + x) \) is convergent, so we have

\[
\log \left(1 - \frac{1}{T}\right) = -\frac{1}{T} + O \left(\frac{1}{T^2}\right)
\]

Using this we compute:

\[
\Pr = (1 - 1/T)^cT
= \exp \left( \log \left(1 - \frac{1}{T}\right) cT \right)
= \exp \left( \left( O \left(\frac{1}{T^2}\right) - \frac{1}{T}\right) cT \right)
= \exp \left( -c + O \left(\frac{1}{T}\right) \right)
= \exp \left( -c + O(\epsilon^{-2}) \right)
= \frac{1}{e} (c + O(\epsilon^{-2}))
\]

Which gives us our result for \( \epsilon \) small enough. \( \square \)

This lemma holds only for \( T > \frac{1}{\epsilon^2} \). If \( T < \frac{1}{\epsilon^2} \) then we use a different algorithm, which succeeds with probability 1: maintain a list of all distinct elements we have seen so far, and return YES if the length of this list ever exceeds \( T \). This requires space \( \Theta(T) = \Theta(\epsilon^{-2}) \).

In the next step we amplify the probability of correctness of the algorithm, obtaining the following:

**Algorithm B:**

...
1. Select $k$ sets $S_1 \ldots S_k$ as in Algorithm A, for $k = C \log(1/P)\epsilon^{-2}$, $C > 0$

2. Let $Z$ be the number of values of $\text{Sum}_{S_j}(x)$ that are equal to 0, $j = 1 \ldots k$

3. If $Z < k/e$ then report YES, otherwise report NO

**Lemma 2.** If the constant $C$ is large enough, then the algorithm B reports a correct answer with probability $1 - P$.

**Proof.** The events $\text{Sum}_{S_j}(x) > 0$ are independent, and Lemma 1 says that the probability of each is at least $1/e + \epsilon/3$ if DE $< (1 - \epsilon)T$ and at most $1/e - \epsilon/3$ if DE $> (1 + \epsilon)T$. Thus by the Chernoff bound, the probability of failure is $2 \exp\left(-\frac{k}{3\epsilon^2}\right)$, which is $1 - P$ when $k = C \log(1/P)/\epsilon^2$ for some $C > 1$.

Altogether, a decision version of the distinct element problem can be solved using $O(\log(1/P)/\epsilon^2)$ units of space, where each unit is an integer in the range $0 \ldots n$. Therefore we get:

**Theorem 3.** The number of distinct elements in the stream can be $(1 \pm \epsilon)$-approximated with probability $1 - P$ using $O(m \cdot \log(1/P)/\epsilon^3)$ units of space, where each unit is an integer in the range $0 \ldots n$.

The space bound is not optimal. The best known algorithm, given in [KNW10], uses only $O(1/\epsilon^2 + \log n)$ bits, including random seed. As we will see later, the $1/\epsilon^2$ term cannot be improved, so that algorithm is optimal. In practice, another algorithm [DF03] needs only 128 bytes to estimate the number of distinct words in all works of Shakespeare with 10% error.

### 3.1 Linear sketches

The essence of the algorithm B is in maintaining several “sketches” of the form $\text{Sum}_{S}(x) = \sum_{i \in S} x_i$. Such sketch is a **linear** function of the vector $x$. This feature enables us to maintain it under increments to the coordinates of $x$. However, we could maintain it under **decrements** as well. It can be easily verified that decrements do not affect the correctness of the algorithm, as long as we have $x \geq 0$ at the end of the stream. Therefore, the algorithm B can be used even if the stream consists of insertions and deletions of elements, as long as no element is ever deleted more times than it is inserted. Such streams are often called **well-formed**.

Randomized linear sketches provide a very powerful method for designing efficient streaming algorithms. In fact, essentially **all** known streaming algorithms that support both insertions and deletions of elements use randomized linear sketches. We will see many examples in the next few lectures.

### 4 Norm Estimation

The algorithm for distinct element estimation used the “vector representation” of the stream. That is, we defined a conceptual vector $x$, showed how to maintain its sketch, and showed how to approximate the desired function of $x$ (the number of non-zero coordinates) using the sketch.

---

3This scenario is often called the **turnstile model** [Mut03].
There are many other functions of $x$ that are of interest. One such class of functions are the $L_p$ norms\footnote{Technically, we should use the “$\ell_p$ norm” notation, since we are dealing with finite-dimensional vectors. However, the “$L_p$ norm” notation seems more common in the streaming community.}, i.e.,

$$\|x\|_p = \sum_i (|x_i|^p)^{1/p}$$

We can also define $\|x\|_\infty = \max_i |x_i|$ and $\|x\|_0 = |\{i : x_i \neq 0\}|$.

What is the space complexity of estimating the $L_p$ norm of the stream (say, under insertions and deletions of elements, and up to, say, $\epsilon = 1/2$)? It took a while, but finally we know the answer to this question, at least up to factors polynomial in $\log m$. Specifically, the answer is

- For $p \in [0, 2]$, $\log^{O(1)} m$ space is sufficient
- For $p > 2$, $n^{1-2/p} \log^{O(1)} m$ space is necessary and sufficient

The pieces of the proof have appeared in more than half-a-dozen papers.

## 5 $L_2$ Norm Estimation

We describe an algorithm for estimating the $L_2$ norm of the count vector due to [AMS99]. It may not be immediately obvious why this quantity is of interest, but it will appear frequently as a useful subroutine in later lectures.

Again, we will start with a very crude approximation and then improve it by repetition.

Algorithm A:

1. Let $r_1, \ldots, r_m$ be independent and identically distributed random variables such that

$$\Pr[r_i = 1] = \Pr[r_i = 0] = 1/2$$

2. Define

$$Z = \sum_i r_ix_i$$

3. Output $Z^2$ as an estimate of $\|x\|_2^2$.

Note that we don’t need to store the $r_i$ explicitly; instead we can use a pseudorandom generator with seed $i$ to generate an appropriate 0–1 random variable. We will see that this approach succeeds as long as the outputs of the pseudorandom generator are 4-wise independent, which can be attained using $O(\log n)$ randomness.

In order to show that $Z^2$ is a good estimator of $\|x\|_2^2$, we will show that it has the correct mean and low variance.
First we compute

\[ E[Z] = E \left[ \left( \sum_i r_i x_i \right)^2 \right] \]
\[ = \sum_{i,j} E[r_i r_j x_i x_j] \]
\[ = \sum_{i,j} x_i x_j E[r_i] E[r_j] \]
\[ = \sum_i x_i^2 E[r_i^2] \]
\[ = \sum_i x_i^2 \]

where we used the fact that \( r_i, r_j \) are independent for \( i \neq j \), as well as that \( E[r_i] = 0, E[r_i^2] = 1 \).

Next we compute the variance,

\[ E[Z^4] = E \left[ \left( \sum_i r_i x_i \right)^4 \right] \]
\[ = \sum_{i,j,k,l} E[r_i r_j r_k r_l] x_i x_j x_k x_l \]

If any of \( i, j, k, l \) is different from the other three, say \( i \neq j, k, l \), then we can write

\[ E[r_i r_j r_k r_l] = E[r_i] E[r_j r_k r_l] = 0 \]

So we have

\[ \text{Var}[Z^4] \leq E[Z^4] \]
\[ \leq 3 \sum_{i,j} E[r_i^2] E[r_j^2] x_i^2 x_j^2 \]
\[ = 3 \sum_{i,j} x_i^2 x_j^2 \]
\[ \leq 3 \left( \sum_i x_i^2 \right)^2 \]
\[ \leq 3 E[Z^2]^2 \]

To improve the quality of our algorithm, we average many independent estimates. That is, we maintain \( k \) independent estimates \( Z_1, Z_2, \ldots, Z_k \) using an independent set of random variables \( r_i \) for each. We then output the estimator \( Z^2 = \frac{1}{k} \sum_i Z_i^2 \). We have \( E[Z^2] = ||x||_2^2 \) and \( \text{Var}[Z^2] = k^{-1} \text{Var}[Z_i^2] = k^{-1} ||x||_2^2 \).

Chebyshev’s inequality implies that

\[ \Pr||Z^2 - ||x||_2^2 > \epsilon||x||_2^2 \leq \frac{1}{k \epsilon^2} \]
In space $O(\frac{1}{\epsilon^2 P})$ we can obtain a $1 \pm \epsilon$ approximation with probability $1 - P$. The dependence on $P$ is polynomial rather than exponential– the distribution of the estimate has “heavy tails.” In order to get exponentially small tails we need to use additional moments of the estimate, as we will see in the next lecture.

References


