Lecture 1

Lecturer: Sofya Raskhodnikova *Scribe(s): Madhav Jha*

1 Introduction

This course will examine algorithms in which the running time is sublinear in the size of the input. In particular, this means that only part of the input can be read. There are many examples in which this setting is interesting, mainly when dealing with large data sets. Some specific examples are scientific databases, the world wide web, data from the Genome Project, and high-resolution images. Another setting where sublinear-time algorithms are interesting include settings where it takes a long time to access the data. This could, for example, be due to to a slow connection or to the fact that data is only available implicitly (such as when we need to run an experiment to get each data point).

We will take a theoretical point of view, and examine general techniques used for solving problems in sublinear time, consider different models for sublinear-time computation, and see some impossibility results. In the course of this study, we will encounter sublinear-time algorithms for a variety of objects, such as graphs, strings, functions, codes, metric spaces and distributions. The analyses of these algorithms make use of tools from probability theory, Fourier analysis, combinatorics and other fields. We will also briefly consider sublinear-space algorithms: specifically, streaming algorithms.

A natural question to ask is what can an algorithm compute if it reads only a sublinear portion of the data? Or more generally, if it runs in sublinear time? While for some problems exact deterministic sublinear-time algorithms exist, for most problems, these algorithms only give "approximate" answers and must be randomized. We will study the tradeoff between the quality of approximation and the resources (namely, the number of samples or the running time) available to the algorithm.

1.1 A Deterministic Sublinear-Time Algorithm

We begin with a rare example of a deterministic sublinear-time algorithm. Consider the following problem:

Given: a metric space on n points, specified by a distance matrix D, where $D_{ij} = \text{distance}(i, j)$.

Output: output (k, ℓ) such that $d(k, \ell) \geq \frac{1}{2} \max_{i,j} D_{ij}$.

Here is an algorithm for this problem:

Algorithm 1: Diameter of a metric space.

1. Pick k arbitrarily.
2. Select ℓ to maximize $D_{k\ell}$.
3. Output (k, ℓ).

Clearly, the running time of the algorithm is $O(n)$, which is sublinear, since the input size is $\Theta(n^2)$. But how close is the output to optimal value $d = \max_{i,j} D_{ij}$?

Claim 1. $\frac{d}{2} \leq z \leq d$.

Proof. The second inequality is trivial. For the first inequality, we use the facts that distances obey the triangle inequality, and that they are symmetric. Suppose the pair u, v is the pair that achieves the maximum distance, i.e. $D_{uv} = d$. Then

$$d = D_{uv} \leq D_{up} + D_{pv} = D_{pu} + D_{pv} \leq 2z.$$

Thus, $z \geq \frac{d}{2}$. □

2 Notions of Approximation

There are various notions of approximation. In this course, we focus on two types of approximation: the classical approximation (more pertinent for search problems) and property testing (defined for decision problems). As mentioned, the classical approximation deals with problems where one needs to compute a value (such the average of a list of numbers) or output a structure (such as the minimum spanning tree of a graph or the furthest pair of points in a metric space). In the former case, the quality of approximation is judged by how close the output is to the desired value. For the latter, one compares the cost of the output to the cost of the optimal solution. For example, in Section 1.1, we saw that the output of Algorithm 1 was between $\frac{d}{2}$ and d, when the optimal solution was d. It is easy to generalize this notion of approximation to all optimization problems:

Definition 2. *Let $\pi(x)$ be the optimal output on input x. For $\gamma > 1$, an algorithm A is a γ-*multiplicative approximation algorithm *if $\forall x$,*

$$\frac{\pi(x)}{\gamma} \leq A(x) \leq \gamma \cdot \pi(x).$$

*Similarly, for $\alpha > 0$, we say that A is an α-*additive approximation algorithm *if $\forall x$,*

$$\pi(x) - \alpha \leq A(x) \leq \pi(x) + \alpha.$$

In decision problems, the outputs are always yes/no or accept/reject, so what does it mean to have an approximate solution? The intuition for our notion (which is not quite accurate, but hopefully helpful) will be that the output is correct for the input to the problem, or to some other input which is “close” to our input.

More formally, suppose we split the set of all possible inputs into YES-instances (on which an exact algorithm should accept) and NO-instances (on which an exact algorithm should reject). What we will require is that on a YES-instance, the algorithm will accept with probability at least $\frac{2}{3}$. The NO-instances will be split into two groups: one group contains all inputs that are “far” from all YES-instances, and one group that contains inputs that are “close” to some YES-instance. We require that the algorithm reject inputs of the first kind with probability at least $\frac{2}{3}$, and we do not care how it handles inputs of the second kind.

Terminology and definitions of property testing. The research area that studies the notion of approximation for decision problems is called *Property Testing*. Recall that a *language* is a class of objects. In Property Testing, a language is sometimes called a *property*. However, depending on the context, a property might also refer to objects of size n in the language $\mathcal{P}$.

Distance. As we saw in the $0^\star 1^\star$ example, our notion of approximation for decision problems calls for a distance function on the inputs $\delta(x, y) \in [0, 1]$. The distance function depends on the problem at hand. Examples of distance functions that have been considered in the literature:

- **Relative Hamming distance**: the fraction of bits/characters/matix entries on which x and y differ.
- **Relative Edit distance**: the minimum number of character substitutions, inserts and deletes needed to transform x into y divided by the length of x.

We will usually look at the Hamming distance.

A *distance from an input x to the property $\mathcal{P}$* is defined as the distance between x and the input in $\mathcal{P}$ closest to x:

$$\delta(x, \mathcal{P}) = \min_{y \in \mathcal{P}} \delta(x, y).$$

We say an input is ϵ-*far* from $\mathcal{P}$ if $\delta(x, \mathcal{P}) \geq \epsilon$.

Input representation. An important issue in defining the distance between inputs is the input representation. For example, a graph can be represented by an adjacency matrix or adjacency lists for all its vertices. The distance, as we defined it, depends on the representation. Also, representation defines what an algorithm can access in one step. We will usually work with the *random access model*, where in a single step the algorithm can access one bit/character/matrix entry or an entry in an adjacency list.

Definition 3 (ϵ-tester). *An algorithm A is an* ϵ-tester *for property $\mathcal{P}$ if it*

1. *accepts all $x \in \mathcal{P}$ with probability at least $\frac{2}{3}$ and*
2. *rejects all x that are ϵ-far from $\mathcal{P}$ with probability at least $\frac{2}{3}$,*

where the probability is taken over the internal coin tosses of A.

The tester has one-sided *error if it always accepts functions satisfying $\mathcal{P}$. If the queries made by the algorithm do not depend on the answers of the queries, A is called a* nonadaptive *tester. Otherwise, it is called an* adaptive *tester.*

As always with probabilistic algorithms, the error in the definition of the ϵ-tester is set arbitrarily to $\frac{1}{3}$. Any constant below $\frac{1}{2}$ would be fine. We can reduce the error to any constant Δ by repeating the algorithm $O(\log \frac{1}{\Delta})$ times and taking the majority answer. As an exercise, if you are not familiar with this fact, try proving it using Chernoff bound.

It is important to analyze both the query complexity and the running time in terms of the input length, n, and the distance parameter, ϵ.

3 Toy Examples

Consider the following example.

Input: a string $w \in \{0,1\}^n$.

Goal: determine if w is an all 0 string, that is, answer the question: Is $w = 00\ldots0$?

To get an exact answer all the time, it is necessary to read the entire input. This is because there may be only 1 bit with value 1, and we must see that bit in order to detect that the string is actually a NO-instance.

But suppose we only want an approximate answer. Namely, we are interested in distinguishing the all 0 string from strings which have a large fraction of 1's ('errors') say at least an ϵ fraction of them. More formally, now what we require from our algorithm is that if $w = 00\ldots0$, then it accepts. If number of 1's in w is at least ϵn, the our algorithm should reject with probability at least $\frac{2}{3}$. Towards this, consider Algorithm 2.

Algorithm 2: Zero tester.

1. Sample $s = \frac{2}{\epsilon}$ positions uniformly and independently at random.
2. If 1 is found, reject; otherwise, accept.

We will now analyze this algorithm. We state and prove the following theorem.

Theorem 4. *There is a nonadaptive one-sided error tester for the* all-zero *property of strings $w \in \{0,1\}^n$ (that is, for the property of being an all 0 string) that runs in $O(\frac{1}{\epsilon})$ time.*

Proof. The required algorithm is Algorithm 2. First note that if $w = 00\dots0$, it is always accepted. Now suppose that w has at least ϵn 1's. The following calculation shows that the algorithm rejects such w with probability $\geq \frac{2}{3}$.

$$\begin{aligned}\Pr[\text{algorithm rejects}] = \Pr[\text{a 1 is found in the sample}] &\leq (1-\epsilon)^s \\ &\leq e^{-\epsilon s} \qquad \text{Using: } 1-x \leq e^{-x} \\ &= e^{-2} < \frac{1}{3}\end{aligned}$$

□

It is interesting to note that the query complexity (the number of samples chosen) of the algorithm is $O(\frac{1}{\epsilon})$, which is independent of n! The running time is identical because as the value is sampled one can check if it is 1 and reject immediately if it is so. Thus, the algorithm runs in time $O(\frac{1}{\epsilon})$.

Question: Don't you need $\log n$ time to write down a location of the bit you want to query?

Answer: It depends on the model. In this class, we will view each query to the input as an operation of constant cost, even though in a more realistic model this cost would be logarithmic. (In your algorithms class, you probably also ignored the cost of comparing two numbers of unbounded length when you analyzed your sorting algorithms.) It should not be hard to convert between the models, since we will always carefully analyze both query and time complexity.

General observation. A general outline of what happened in the above algorithm is that points were picked at random, and they were tested as witnesses for a bad event. This leads to the following lemma.

Lemma 5 (Witness Lemma)**.** *If a test catches a witness with probability $\geq p$ then $s = 2/p$ iterations of the test suffices to catch a witness with probability $\geq 2/3$.*

Proof.

$$\Pr\left[\text{pick no witness in } \frac{2}{p} \text{ samples}\right] < (1-p)^{\frac{2}{p}} \leq e^{-p \cdot \frac{2}{p}} = \frac{1}{e^2} < \frac{1}{3}.$$

□

This is a good observation to keep in mind for analyzing probabilistic algorithms.

We now give an example of randomized approximation.

Input: a string $w \in \{0,1\}^n$.

Goal: estimate the fraction of 1's in w (like in polls).

Here again we are interested in the approximate version of the problem. Specifically, we would like an approximation algorithm with additive error ϵ. Towards this, consider the Algorithm 3:

Algorithm 3: Average estimator.

---

1. Sample $s = \frac{1}{\epsilon^2}$ positions uniformly and independently at random.
2. Output average of the sample.

We will now analyze this algorithm. We state and prove the following theorem.

Theorem 6. *There is an algorithm which outputs an ϵ-additive approximation of the fraction of 1's in a 0-1 string with probability at least $\frac{2}{3}$ and runs in time $O(\frac{1}{\epsilon^2})$.*

In the proof, we make use of the following inequality, called the Hoeffding bound, which is indispensable in the analysis of randomized algorithms.

Fact 7 (Hoeffding Bound). *Let $Y_1, \ldots, Y_s$ be independently distributed random variables in* $[0,1]$ *and let* $Y = \sum_{i=1}^{s} Y_i$. *Then,* $\Pr[|Y - E[Y]| \geq \delta] \leq 2e^{-2\delta^2/s}$.

Proof of Theorem 6: Algorithm 3 is the required algorithm. We bound the probability that the additive error is greater than ϵ. For each $i \in [s]$, let $Y_i \in \{0,1\}$ be the random variable that denotes the value of the i'th sample and let $Y = \sum_{i \in [s]} Y_i$ be the sample sum. Using the linearity of expectation, $E[Y] = \sum_{i \in [s]} E[Y_i] = \frac{s \times (\#\text{ of 1's in } w)}{n}$. Therefore, we have,

$$\begin{aligned}
\Pr[|(\text{sample average}) - (\text{fraction of 1's in } w)| \geq \epsilon &= \Pr[|Y - E[Y]| \geq \epsilon s] \\
&\leq 2e^{\frac{-2(\epsilon s)^2}{s}} \qquad \text{Using Hoeffding bound; Fact 7} \\
&= 2e^{-2} < \frac{1}{3}.
\end{aligned}$$

The last equality follows by substituting $s = \frac{1}{\epsilon^2}$. □

4 Property Testing Examples

4.1 Testing if an Image is a Half-plane

In this section, we present an example of a property tester for testing properties of images. We use image representation popular in learning theory. Each image is represented by an $n \times n$ matrix M of pixel values. We focus on black and white images given by binary matrices with black denoted by 1 and white denoted by 0. To keep the correspondence with the plane, we index the matrix by $\{0, 1, \ldots, n-1\}^2$, with the lower left corner being $(0,0)$ and the upper left corner being $(0, n-1)$.

An image is a half-plane if there is a vector $w \in \mathbb{R}^2$ and a number $a \in \mathbb{R}$ such that a pixel x is black if and only if $w^T x \geq a$. We now state the problem.

Input: an $n \times n$ matrix M consisting of 1's and 0's (representing black and white pixel values respectively).

Goal: an ϵ-tester for testing half-plane with running time $O\left(\frac{1}{\epsilon}\right)$.

The test. We will use what might be termed as *testing by implicit learning* paradigm. Namely, we first try to learn the outline of the image by querying a few pixels. Then, we test if the image conforms to the outline by random sampling, and reject if something is wrong. Specifically, the algorithm first finds a small region within which the dividing line falls. Then it checks if pixels on one side of the region are white and on the other side are black. Call pixels $(0,0), (0,n-1), (n-1,0), (n-1,n-1)$ *corners*. Call the first and the last row and the first and the last column of the matrix *sides*. For a pair of pixels p_1, p_2 , let (p_1, p_2) denote the line through (p_1, p_2). Let $R_1(p_1, p_2)$ and $R_2(p_1, p_2)$ denote the regions into which (p_1, p_2) partitions the image pixels not on the line.

Theorem 8 (Half-plane tester). *There is a nonadaptive one-sided error tester for the* Half-plane *property of images (that is, for the property of being a half-plane) that runs in* $O(\frac{1}{\epsilon})$ *time.*

Algorithm 4: Half-plane tester [Ras03].

1 Query the four corners. Let s be the number of sides with differently colored corners.

2 If $s = 0$ (all corners are of the same color c), query $\frac{\ln 3}{\epsilon}$ pixels independently at random. Accept if all of them have color c. Reject otherwise.

3 If $s = 2$,

1. For both sides with differently colored corners, do binary search of pixels on the side to find two differently colored pixels within distance less than $\epsilon n/2$. For one side, call the white pixel w_1 and the black pixel b_1 . Similarly, define w_2 and b_2 for the second side.

2. Let $W_i = R_i(w_1, w_2)$ and $B_i = R_i(b_1, b_2)$ for $i = 1, 2$. W.l.o.g., suppose W_2 and B_1 intersect while W_1 and B_2 do not. Query $\frac{2\ln 3}{\epsilon}$ pixels from $W_1 \cup B_2$ independently at random. Accept if all pixels from W_1 are white, all pixels from B_2 are black. Otherwise, reject.

4 If s is not 0 or 2, reject.

Proof. Algorithm 4 is the required algorithm. The algorithm queries at most $\frac{2\ln 3}{\epsilon} + O(\log(\frac{1}{\epsilon}))$ pixels. To prove correctness, we need to show that all half-planes are always accepted, and all images that are ϵ-far from being half-planes are rejected with probability at least 2/3. Observe that the number of sides with different corners is 0, 2, or 4. This is because the cycle $((0, 0), (n - 1, 0), (n - 1, n - 1), (0, n - 1), (0, 0))$ visits a vertex of a different color every time it moves along such a side. We analyze each of these cases separately.

- Case (a) [4 differently colored sides]: In this case, the image cannot be a half-plane.

- Case (b) [0 differently colored sides]: In this case, the image is a half-plane if and only if it is unicolored. Therefore, we proceed as in the toy example (Section **??**) to test if all pixels have the same color. If it is unicolored, the test always accepts since it never finds pixels of different colors. If the image is ϵ-far from being a half-plane, it has at least ϵn^2 pixels of a wrong color. Since tester samples $\frac{\ln 3}{\epsilon}$ pixels uniformly and independently, the tester rejects with probability at least 2/3.

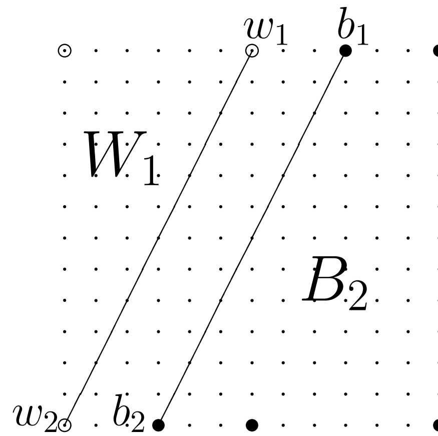
- Case (c) [2 differently colored sides]: The test always accepts all half-planes because it rejects only if it finds two white pixels and two black pixels such that the line through the white pixels intersects the line through the black pixels. It remains to show that if an image is ϵ-far from being a half-plane, it is rejected with probability $\geq 2/3$. We prove the contrapositive, namely, that if an image is rejected with probability $< 2/3$, modifying an ϵ fraction of pixels can change it into a half-plane. Suppose that an image is accepted with probability $\geq 1/3 = e^{-\ln 3} > (1 - \frac{\epsilon}{2})^{\frac{2\ln 3}{\epsilon}}$.

  That means that $< \epsilon/2$ fraction of pixels from which we sample in step 3.2 differ from the color of their region (white for W_1 and black for B_2). Note also that there are at most $\epsilon n^2/2$ pixels outside of $W_1 \cup B_2$. Changing the color of all black pixels in W_1 and all white pixels B_2 (see Figure 1) and making all pixels outside of those regions white, creates a half-plane by changing $< \epsilon$ fraction of the pixels, as required.

□

Other properties of images that has been considered in “pixel” model include convexity and connectedness by [Ras03] and partitioning by [KKNBZ11]. Another model for sparse images has also been considered in the setting of property testing by [TR10].

Figure 1: Half-plane test



References

[KKNBZ11] Igor Kleiner, Daniel Keren, Ilan Newman, and Oren Ben-Zwi. Applying property testing to an image partitioning problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(2):256–265, 2011.

[Ras03] Sofya Raskhodnikova. Approximate testing of visual properties. In *RANDOM-APPROX*, pages 370–381, 2003.

[TR10] Gilad Tsur and Dana Ron. Testing properties of sparse images. In *FOCS*, pages 468–477, 2010.