CS 7870: Seminar for Algorithms for Big Data (Spr'26) **Northeastern University**

Lecture 1

Jan 13, 2026

Instructor: Soheil Behnezhad *Scribe: Soheil Behnezhad*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications.*

1 Overview

Linear-time linear-space algorithms have long been considered the gold standard of efficiency. Indeed, it is hard to imagine more efficient algorithms since even reading and storing the input just once requires linear time and linear space. However, as the size of inputs get larger and larger, even such algorithms become inefficient. Our goal in this course is to study extremely resource-efficient algorithms that can process these massive inputs. Let us start with two simple examples of such algorithms:

- **Sublinear Time Algorithms:** These algorithms assume the input is already in memory and attempt to provide an answer with $\ll n$ queries to the input of size n.
- **Sublinear Space Algorithms:** Also known as **streaming algorithms**. In this model, we assume that the input arrives piece by piece one at a time and the algorithm does not have enough space to store all of the input.

We will start with some toy problems to illustrate these models. We will see more realistic problems later in the course.

2 Sublinear Time Example

The following problem is a motivating example for sublinear time algorithms.

Problem 1.

Input: An array $A[1,\dots,n]$ which is some permutation of $[n]$.

Goal: Output an index i such that $A[i]$ is even.

Let us first see a simple deterministic algorithm for this process.

Algorithm (Deterministic): For $i \in \{1,\dots,n\}$, query each $A[i]$ in order.
If $A[i]$ is even, return i.

Now consider an adversarial input which contains all the odd elements in $[n]$ in its first $\lceil \frac{n}{2} \rceil$ indices. The running time of this algorithm on such an input is $\Omega(n)$, since it must iterate through $\lceil \frac{n}{2} \rceil + 1$ indices to find an even element. As such, deterministic techniques are insufficient to solve this problem quickly.

Remark. This result holds for any deterministic algorithm. One may construct an adversarial input by placing the odd elements of $[n]$ in the first $\lceil \frac{n}{2} \rceil$ indices that the algorithm checks. Then the algorithm takes $\Omega(n)$ time to run on that adversarial input. We will see examples of such lower bound proofs in future lecture notes.

One solution to this problem is to permit the use of *randomized algorithms*, which are allowed to flip coins and branch on the outcome of the coin flips. Here is one such algorithm:

Algorithm (Randomized): Choose a random $i \in n$ and query $A[i]$.
If $A[i]$ is even, return i. Otherwise, repeat.

While this algorithm seems to have a similarly-bad worst case, it is resistant to the kind of adversarial inputs that plague deterministic approaches to this problem. Moreover, the expected runtime is bounded above by a constant value.

Proposition 1. *Let k be the number of iterations required for the above algorithm to produce an output. Then, on any input, the algorithm satisfies $\mathbb{E}[k] \leq 2$.*

Proof. For simplicity, we assume that n is even. Let $Y_i := 1(k \geq i)$ be the indicator random variable that the algorithm takes at least i iterations. Note that $k = \sum_{i=1}^{\infty} Y_i$. Taking expectation, we have

$$\mathbb{E}[k] = \mathbb{E}\left[\sum_{i=1}^{\infty} Y_i\right] = \sum_{i=1}^{\infty} \mathbb{E}[Y_i] = \sum_{i=1}^{\infty} \Pr(Y_i = 1),$$

where the second equality follows from the linearity of expectation (we will discuss this in the next lecture). Since n is even, each query succeeds with probability $\frac{1}{2}$. Thus,

$$\Pr(Y_i = 1) = \frac{1}{2^{i-1}}$$

Substituting back into the sum yields

$$\mathbb{E}[k] = \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} = 1 + \frac{1}{2} + \frac{1}{4} + \cdots \leq 2. \quad \square$$

3 Sublinear Space Example

A modification of the previous problem demonstrates some of the fundamental difficulties with devising sublinear space algorithms.

Problem 2.

Input: An array $A[1, \ldots, n-1]$ which is some permutation of $[n] \setminus \{k\}$, where $k \in [n]$ is not known. Elements of the array are sent one by one.

Goal: Determine k.

One trivial solution is to store all the elements of A and then iterate over them to determine which is missing. However, storing all the elements of A requires $\Theta(n)$ space. We can do better and achieve constant space complexity with the following algorithm:

Algorithm: Define a variable $s = 0$.
For each element $a \in A$ that is sent, set $s = s + a$.
After all elements are sent, output $\frac{n(n+1)}{2} - s$.

The proof of correctness is as follows:

Proof. Recall that $\sum_{i\in[n]} i = \frac{n(n+1)}{2}$, and note that $s = \sum_{i\in[n]\setminus\{k\}} i$.
Since $\sum_{i\in[n]\setminus\{k\}} i = \left(\sum_{i\in[n]} i\right) - k$, solving for k yields

$$k = \sum_{i\in[n]} i - \sum_{i\in[n]\setminus\{k\}} i = \frac{n(n+1)}{2} - s$$

□

Remark. There is only a single variable stored, which takes $\Theta(1)$ space. However, this variable must be able to store a value of size at most $\frac{n(n+1)}{2} - 1$, which requires $\Theta(\lg(n^2)) = \Theta(\lg n)$ *bits* of space. It is important to note whether a space requirement is given in terms of exact bits or abstracted into words.

Exercise: Prove that $\Omega(\lg n)$ bits of space are required, even for randomized algorithms.

Problem 3.

Input: An array $A[1,\ldots,n-2]$ which is some permutation of $[n]\setminus\{a,b\}$, where $a,b\in[n]$ are not known and $a\neq b$. Elements of the array are sent one by one.

Goal: Determine a and b.

We can still solve the problem using $O(\log n)$ bits of space by maintaining the sum and sum of squares of the elements.

Algorithm: Define variables $s=0$ and $t=0$
For each element $x\in A$ that is sent:

- set $s = s + x$,
- set $t = t + x^2$.

After all elements are sent:

- set $u = \frac{n(n+1)}{2} - s$,
- set $v = \frac{n(n+1)(2n+1)}{6} - t$,
- output $\dfrac{u \pm \sqrt{2v - u^2}}{2}$.

Proof. Let the missing elements be a and b.

Recall that

$$\sum_{i\in[n]} i = \frac{n(n+1)}{2} \quad\text{and}\quad \sum_{i\in[n]} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

Since A contains all elements of $[n]$ except a and b, we have

$$s = \sum_{i\in[n]\setminus\{a,b\}} i, \qquad t = \sum_{i\in[n]\setminus\{a,b\}} i^2.$$

Thus,

$$u = a + b \quad\text{and}\quad v = a^2 + b^2.$$

Using the identity $(a+b)^2 = a^2 + b^2 + 2ab$, we obtain

$$ab = \frac{u^2 - v}{2}.$$

Consider the polynomial

$$p(z) = (z-a)(z-b) = z^2 - (a+b)z + ab.$$

Substituting $a + b = u$ and $ab = \frac{u^2-v}{2}$ gives

$$p(z) = z^2 - uz + \frac{u^2 - v}{2}.$$

Therefore a and b are exactly the roots of the quadratic equation

$$z^2 - uz + \frac{u^2 - v}{2} = 0,$$

which by the quadratic formula yields

$$z = \frac{u \pm \sqrt{u^2 - 4 \cdot \frac{u^2-v}{2}}}{2} = \frac{u \pm \sqrt{2v - u^2}}{2}.$$

Hence the algorithm correctly recovers a and b. $\square$

Remark. The algorithm stores a constant number of variables, so it uses $\Theta(1)$ words of space. However, the values s, t, u, and v can be as large as $\Theta(n^2)$ and $\Theta(n^3)$, so they require $\Theta(\lg n)$ bits per word. Thus the algorithm uses $\Theta(\lg n)$ bits of space overall.

Exercise: Consider the general version of the problem, where $A[1, \dots, n-p]$ is some permutation of $[n] \setminus \{k_1, k_2, \dots, k_p\}$ for unknown $k_1, \dots, k_p \in [n]$ and the goal is to find all of the k_i.

- Show that this is solvable in $O(k^2 \lg n)$ bits of space deterministically.
- Show that this is solvable in $O(k \lg \frac{n}{k})$ bits using a randomized algorithm.