Time Complexity Analysis

Chan Ju Arya Sapkal Saurav Tiwari

1 Key Variables

- S: Total number of **unique symptoms** across all diseases.
- D: Total number of **diseases**.
- M: Number of **user-entered symptom primes** in one run.

Our goal is to understand the computational costs in two phases:

1. **Precomputation (Once at Startup)**
2. **Per-Analysis (When User Inputs Symptoms)**

2 Phase 1: Precomputation

2.1 1. Generating S Prime Numbers

We first generate the first S primes. If we use the naive primality check (dividing by all integers up to $\sqrt{n}$), the approximate time can be viewed as:

$$\mathcal{O}(S \times \sqrt{p_S}),$$

where p_S is roughly the size of the S-th prime. For small S (e.g., 10–20), this is very fast. For larger S, it can become a bottleneck (though generating primes is well-understood and generally not severe for typical use cases).

2.2 2. Building Disease SQFs & an Inverted Index

- *Disease SQF:* Multiply the prime for each symptom in a disease. For each of the D diseases, this is roughly $\mathcal{O}(\text{number of symptoms per disease}) \leq \mathcal{O}(S)$.
- *Inverted Index:* A dictionary from prime $\rightarrow$ (set of diseases). Inserting entries involves the same pass over each disease's symptoms, again $\mathcal{O}(D \cdot S)$ in the worst case.

Overall, precomputation is dominated by:

$$\mathcal{O}\big(S\sqrt{p_S} + D \cdot S\big).$$

With small S and D, this is typically very quick.

3 Phase 2: Per-Analysis (User Input)

When a user provides M symptom primes, we do:

1. **Validation and Parsing** (*checking each prime* against known primes): $\mathcal{O}(M)$.
2. **Matching** with the **Inverted Index**:
   - For each of the M user primes, look up the corresponding diseases in a hash map (average $\mathcal{O}(1)$).
   - If every prime happened to map to most of the D diseases, this step is at worst $\mathcal{O}(M \cdot D)$.
   - In many cases, it is much faster because not all primes appear in all diseases.
3. **Optional Graph Visualization**: checks each (prime, disease) pair to draw an edge if relevant. In a worst-case scenario, this can be another $\mathcal{O}(M \times D)$ operation. Typically fast for small M and D.

Hence, **each user query** takes up to:

$$\mathcal{O}\big(M + (M \times D) + (M \times D)\big) \;=\; \mathcal{O}(M \times D).$$

For small M and D, this is generally quite efficient.

4 Summary

- **Precomputation (One-Time)**: $\mathcal{O}\big(S\sqrt{p_S} + D \cdot S\big)$.
- **Per-Analysis (Each Time a User Inputs M Symptoms)**: $\mathcal{O}(M \times D)$.

Most Common Use Cases: In practical settings where S (number of unique symptoms) and D (number of diseases) are both relatively small (e.g., $S \approx 10$–20, $D \approx 5$–10), the time complexities are effectively very low. Generating a handful of primes, building the index, and matching a few user-entered primes are all near-instant in typical hardware environments.
If S or D grow large, the following become potential bottlenecks:

1. **Prime Generation** via naive trial division. (Use a faster sieve if S is large.)
2. **Matching/Visualization** if $M \times D$ becomes large, causing many lookups and edge plots.