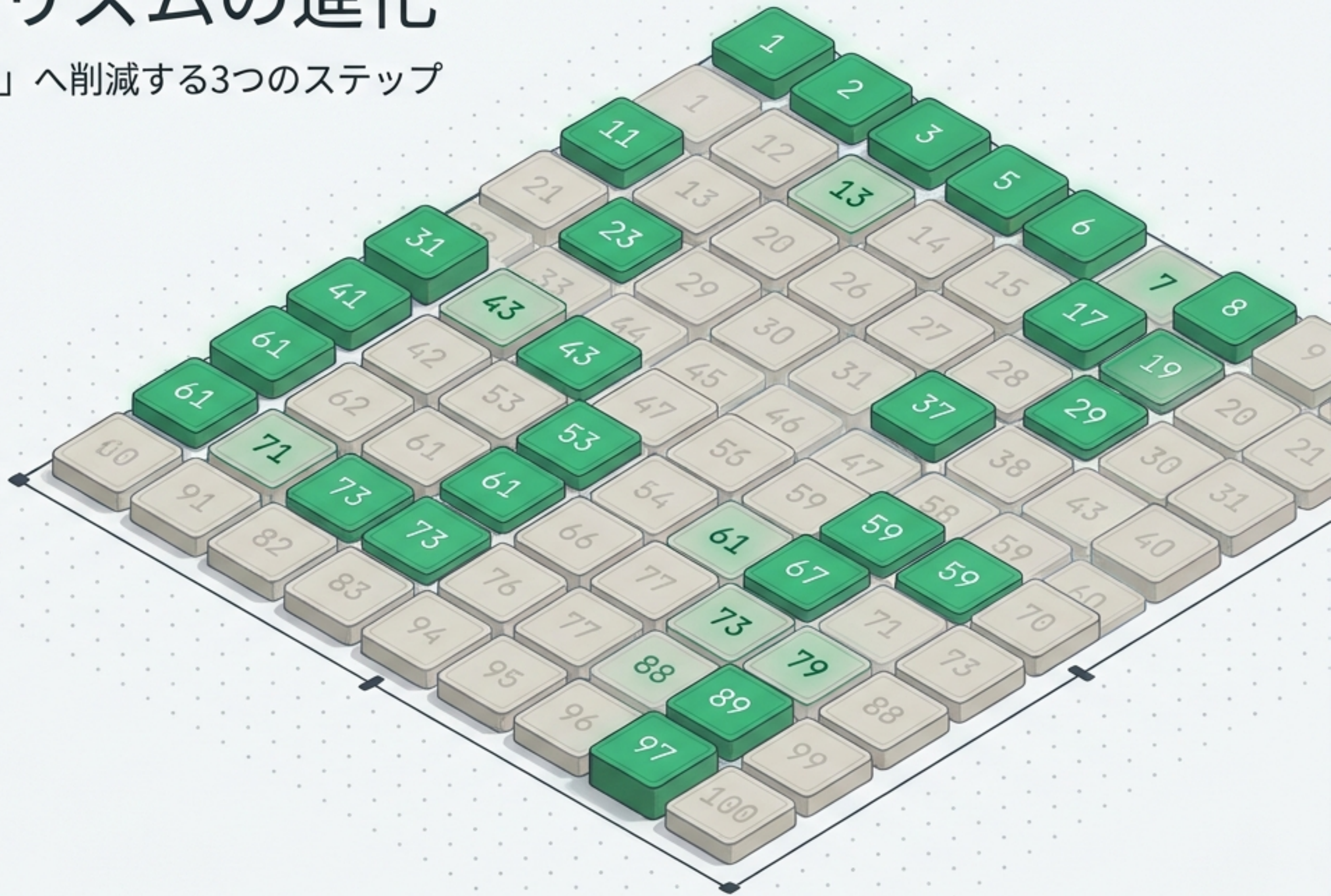


素数列挙アルゴリズムの進化

実行回数を「171回」から「2回」へ削減する3つのステップ



The Challenge: 素数をいかに速く見つけ出すか



素数の定義

2以上の自然数のうち、正の約数が1と自身だけである数。

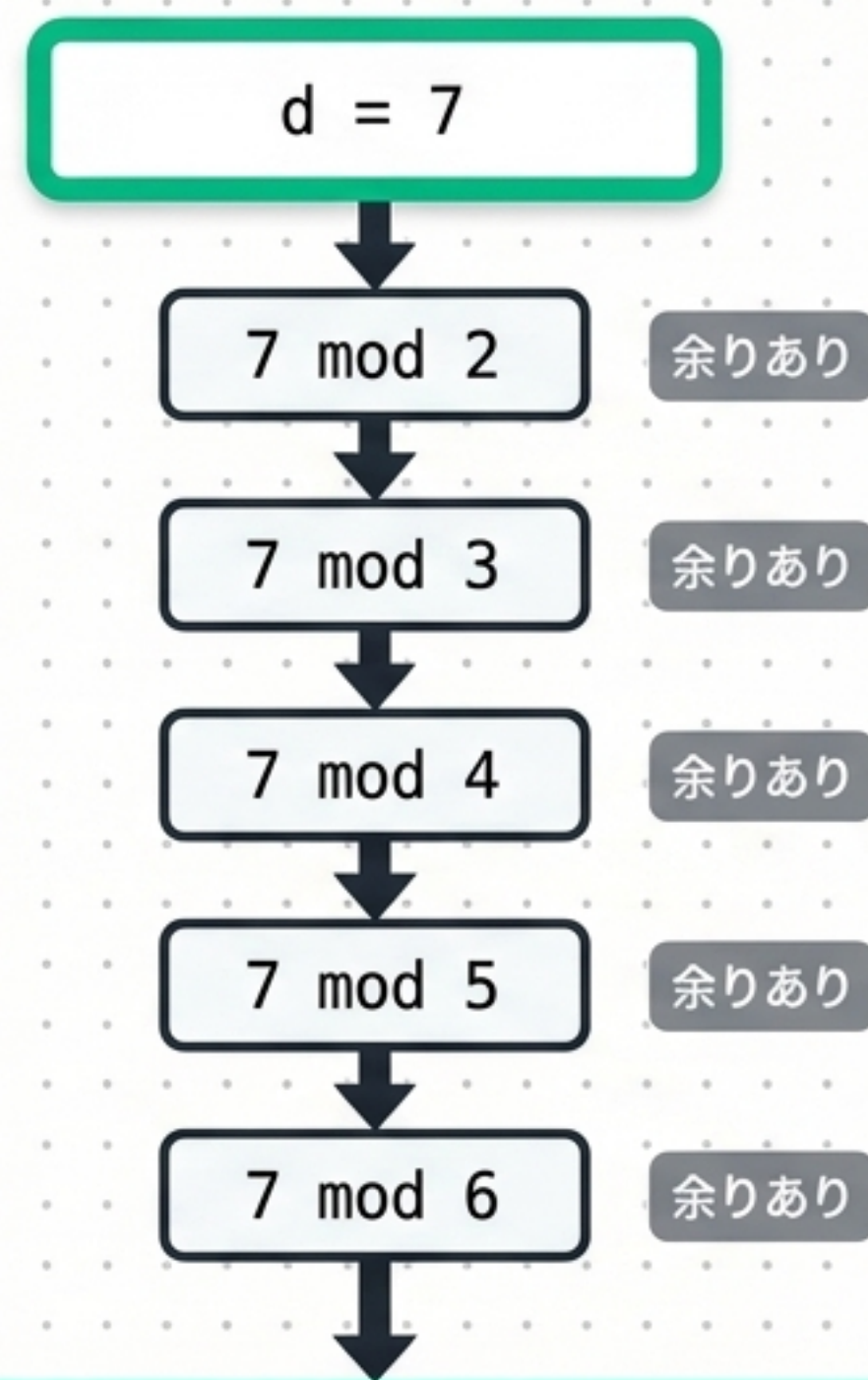
前提条件

本アルゴリズムにおける配列の要素番号は「1」から始まる。

Phase 1: 愚直な探索 (関数 prime1)

アプローチ1: すべての数を割ってみる (Trial Division)

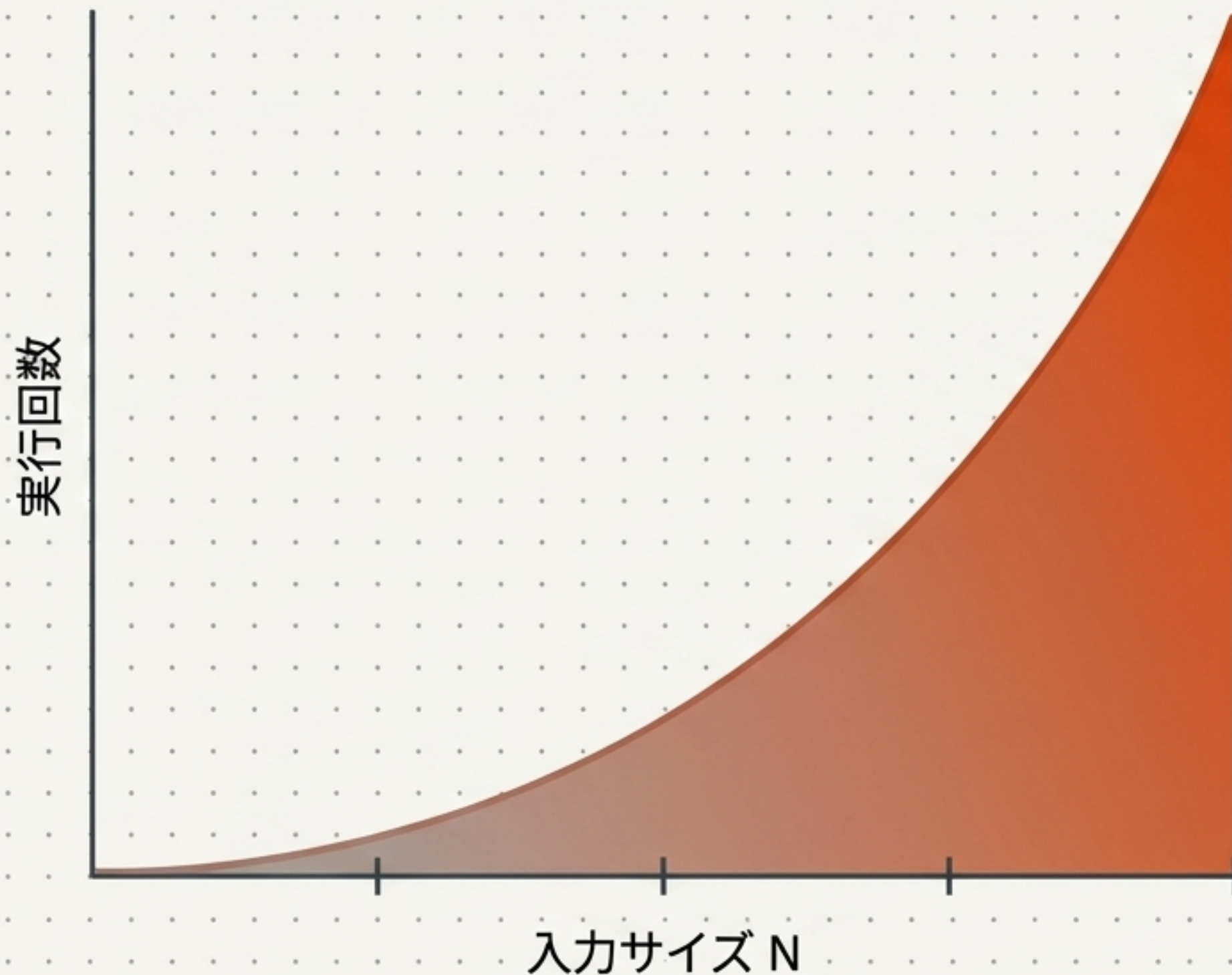
```
while (d <= N)
  isPrime <- true
  t <- 2
  while (t < d)
    if (d mod t == 0)
      isPrime <- false
    endif
    t <- t + 1
  endwhile
  if (isPrime == true)
    /* primes の末尾に d の値を追加する */
  endif
  d <- d + 1
endwhile
```



一度も割り切れなかったため「素数」と判定

确实だが、対象の数より小さいすべての数で割り算を試行するため泥臭い手法。

設問ア: なぜこの方法は遅いのか？



構造的ブレイクダウン

外側のループ: 2から N までの $N-1$ 回



内側のループ: 最大 $N-2$ 回実行

The Answer Box

結論

時間計算量: $O(N^2)$

コンテキスト例 ($N=20$)

$N=20$ の場合、内側のループは $0+1+2+\dots+18$ と蓄積し、171回も実行されてしまう。

Phase 2: 発想の転換「エラトステネスの篩」 (関数 prime2)

アプローチ2: 「割れるか確かめる」から「倍数を消していく」へ

すべてを素数候補とする

Step 1: 初期状態



2の倍数を除外

Step 2:

d=2 の倍数を除外



3の倍数を除外

Step 3:

d=3 の倍数を除外



素数を見つけたら、その倍数をすべて「素数ではない」とマークし、無駄な割り算を完全に排除する。

設問イ・ウ: 倍数スキップのロジックを読み解く

設問イ

対象が素数かどうかの判定

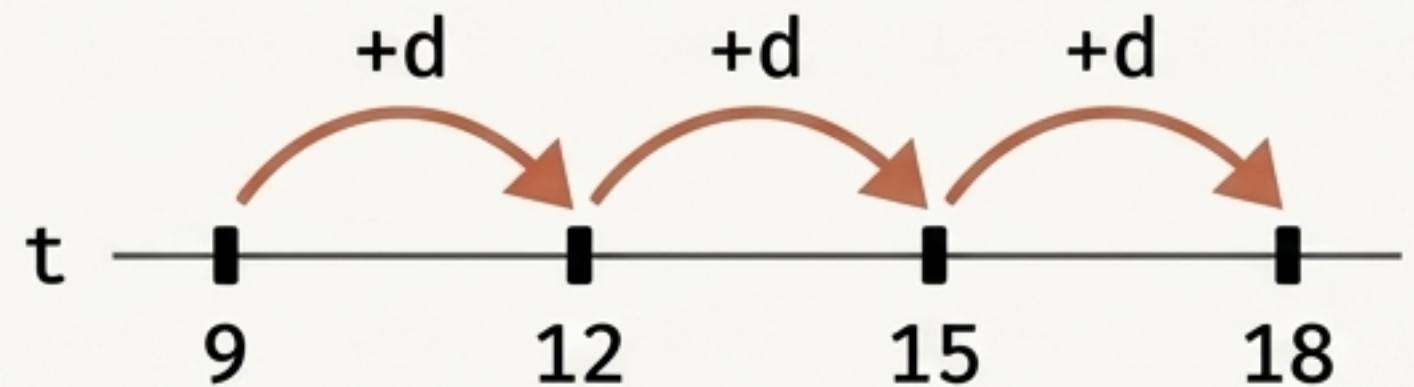


isPrime[d]

対象の数 d がFalseにされていなければ、それは素数である。

設問ウ

次の倍数へのジャンプ



$t + d$

$t = d \times d$ から始まり、歩幅 d でFalseに塗りつぶしていく。

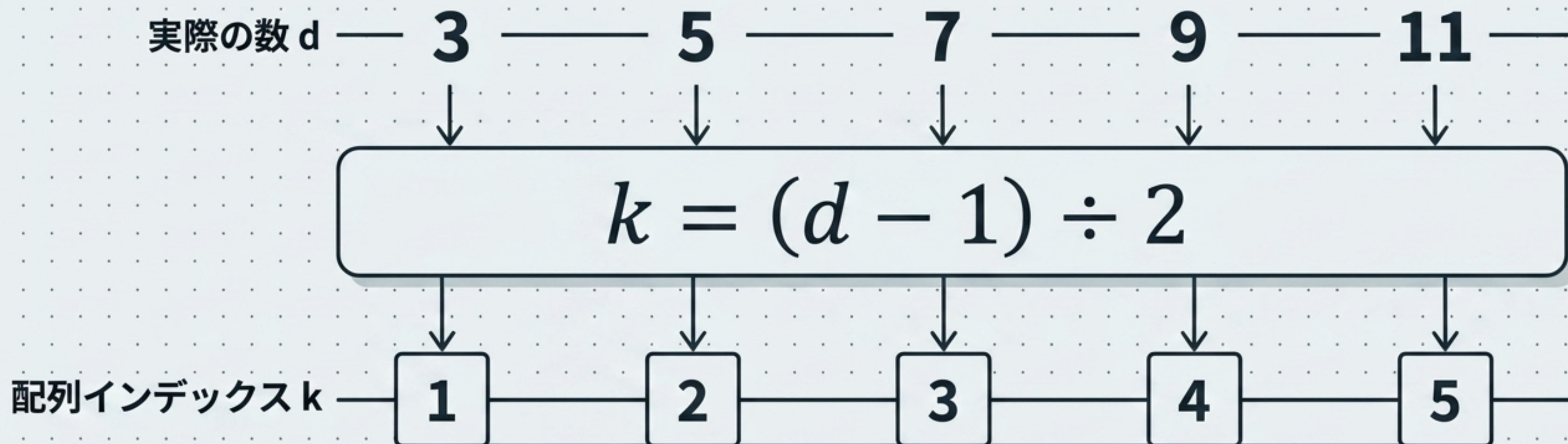
Phase 3: 数学的性質による究極の洗練 (関数 prime3)

アプローチ3: 「4以上の偶数」という無駄を最初から捨てる



4以上の偶数はすべて2の倍数であり、絶対に素数ではない。
この自明の理を利用し、探索空間を最初から半分に絞り込む。

設問エ・オ: メモリ空間を圧縮するインデックス変換



[エ] 素数判定

```
if ( isPrime[ (d-1)÷2 ] )
```

[オ] 倍数の除外

```
isPrime[ (t-1)÷2 ] <- false
```

設問力: 偶数を踏まない「歩幅の調整」

Phase 2
の歩幅

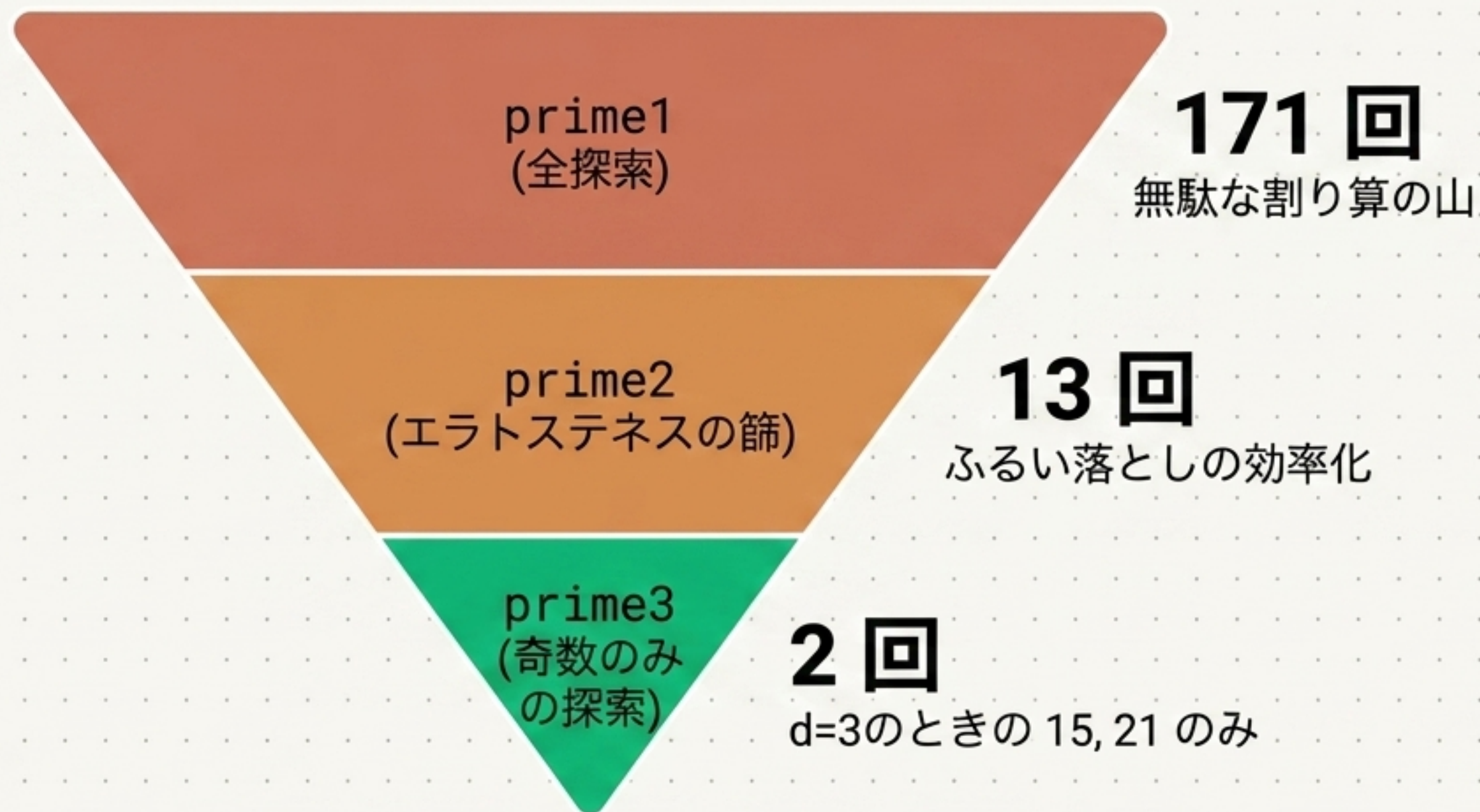


Phase 3
の歩幅



$$[力] \quad t \leftarrow t + 2 \times d$$

設問4: 最適化の証明 (N=20 の実行回数比較)



Nがたった20でも、アルゴリズムの選択が「171」対「2」という圧倒的な差を生む。

Architectural Comparison Matrix

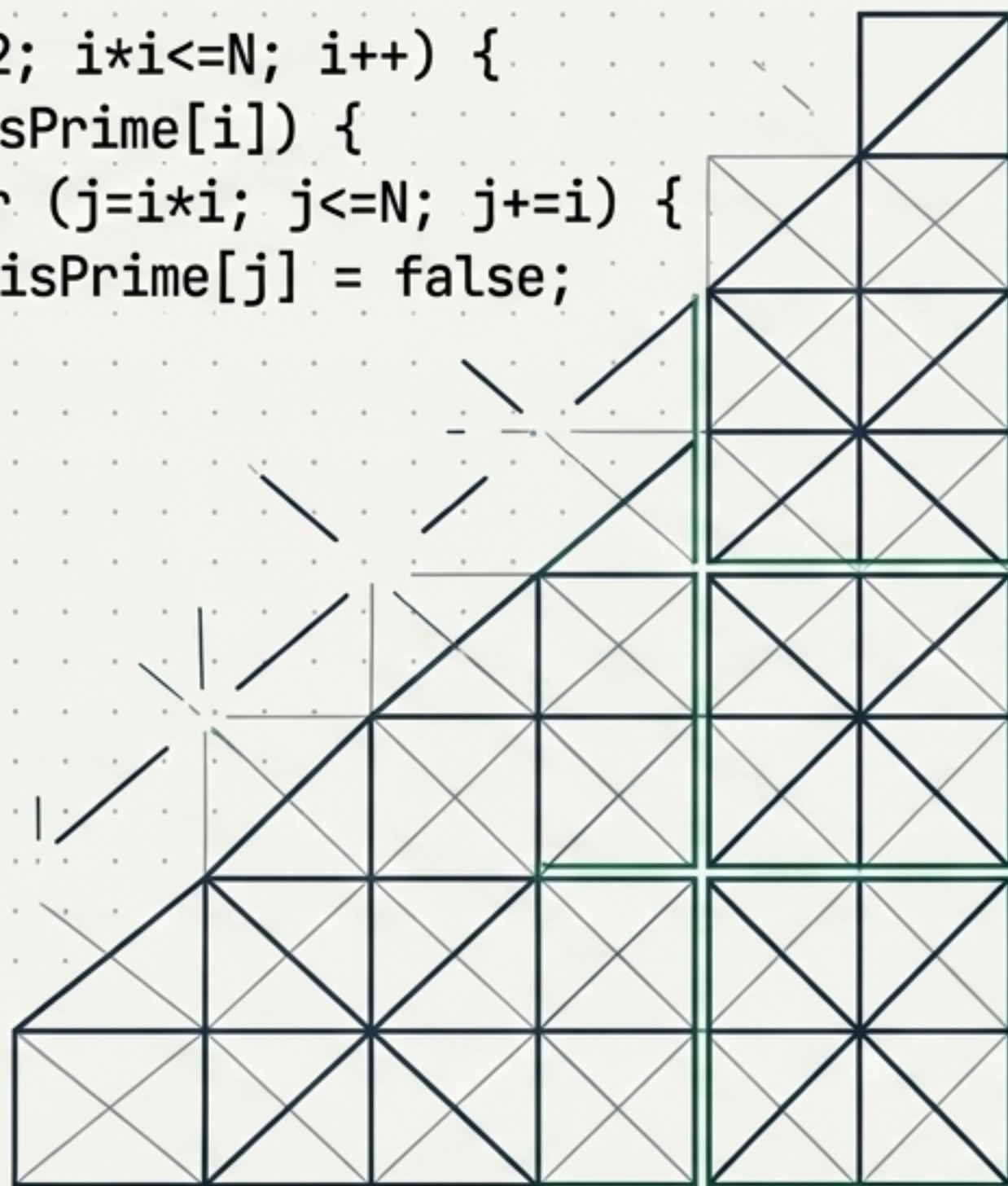
アルゴリズム特性の総括表

	prime1	prime2	prime3
着眼点 (Approach)	全探索（割り算）	篩（倍数の除外）	篩＋数学的性質 (奇数のみ対象)
時間計算量 (Time Complexity)	$O(N^2)$	大幅減	さらに半分
必要なメモリ・配列 (Space/Memory)	なし	$N + 1$ 	$(N + 1) / 2$ 

実行速度（時間）とメモリ空間（空間）はトレードオフになりがちだが、
数学的性質を利用することで両方を同時に最適化できた。

Key Takeaways: プログラミングは「数学のレンズ」で進化する

```
for (i=2; i*i<=N; i++) {  
    if (isPrime[i]) {  
        for (j=i*i; j<=N; j+=i) {  
            isPrime[j] = false;  
        }  
    }  
}
```



1.

前提を疑う

「割れるか」を確かめるのではなく、「倍数を消す」という逆転の発想が劇的な速度向上を生む。

2.

データの性質を利用する

「偶数は素数ではない」という自明の理をモデルに組み込むことで、処理空間とメモリの両方を半減できる。

3.

真の最適化とは

コードの小手先の記述テクニックではなく、事象の数学的なモデリングこそがアルゴリズムの進化を決定づける。