

```
# 素数判定アルゴリズム prime3
```

```
def prime3(N):
```

```
    if N <= 1: return []
```

```
    prime_list = [2]
```

```
    limit = int(N**0.5)
```

```
    # 奇数のみをチェック
```

```
    for i in range(3, N + 1, 2):
```

```
        is_prime = True
```

```
        # 平方根までをチェック
```

```
        for p in prime_list:
```

```
            if p * p > i:
```

```
                break
```

```
            if i % p == 0:
```

```
                is_prime = False
```

```
                break
```

```
        if is_prime:
```

```
            prime_list.append(i)
```

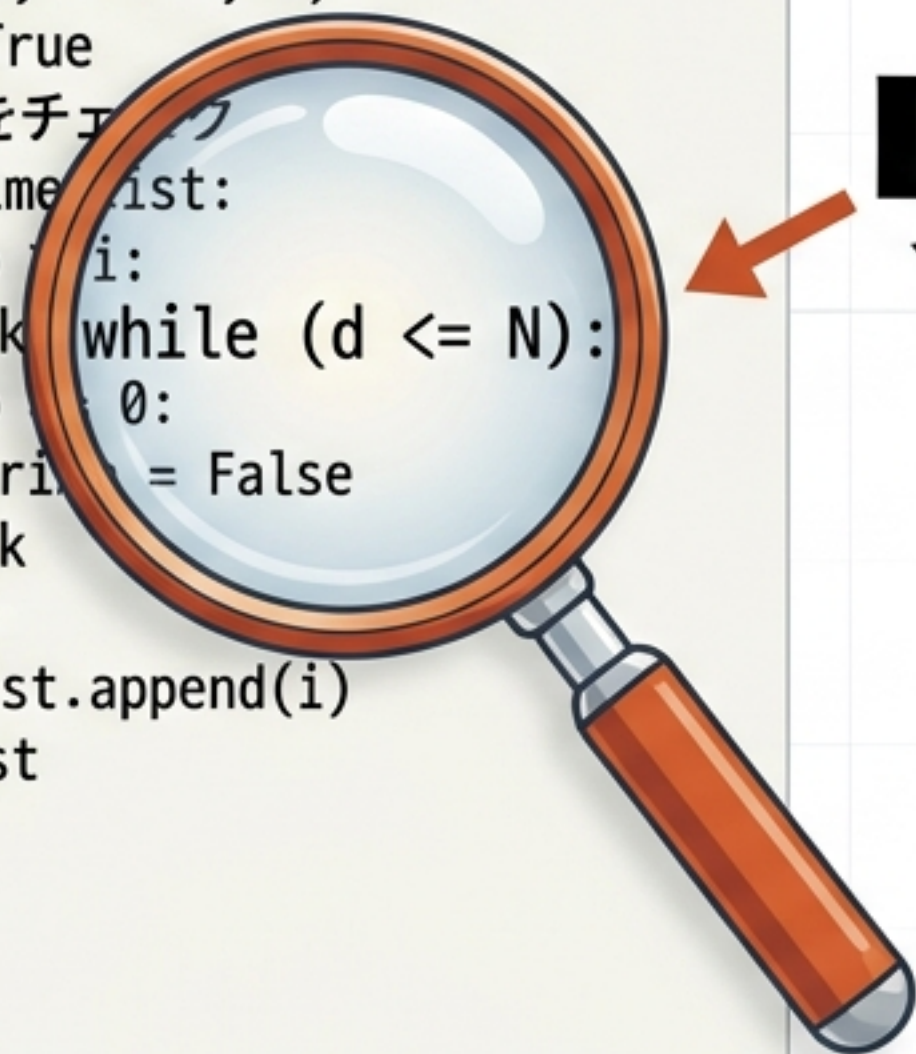
```
    return prime_list
```

```
# 実行例
```

```
N = 97
```

```
result = prime3(N)
```

```
print(f"N={N} までの素数: {result}")
```

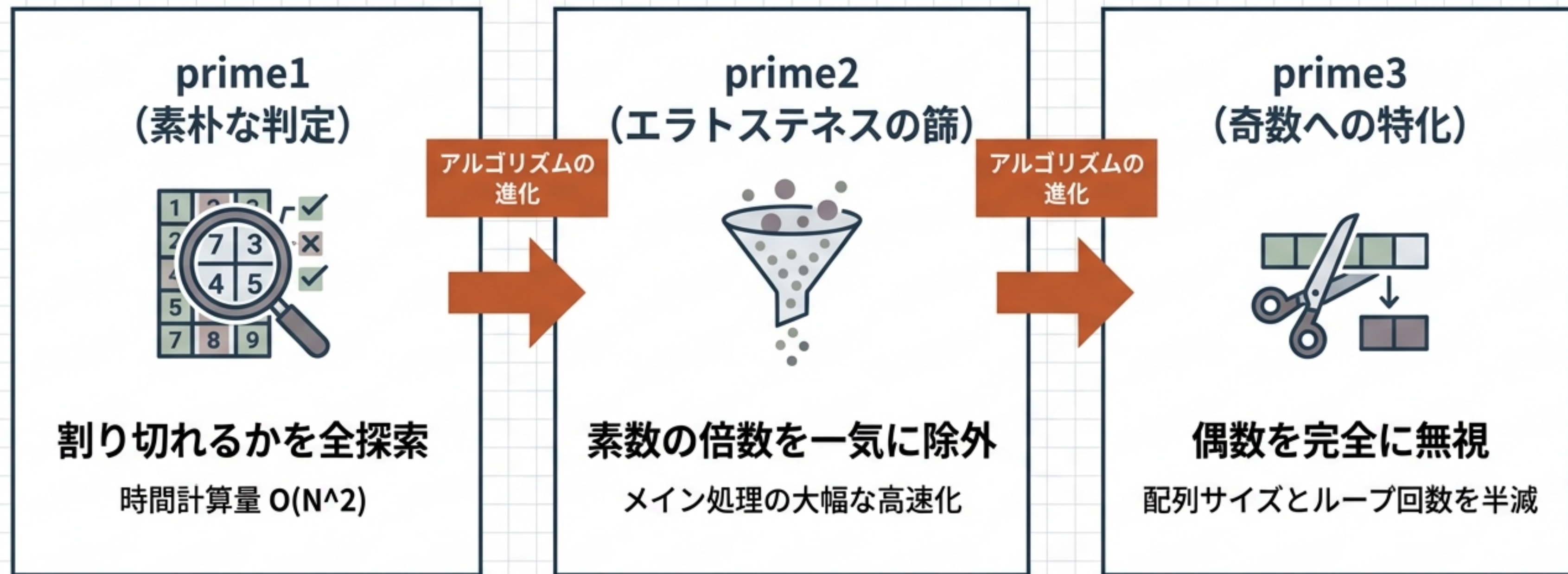


2024年秋期 応用情報技術者試験 午後問3

素数判定アルゴリズムとPython コード完全解説

prime1からprime3へ。計算量 $O(N^2)$ を劇的に削減する「エラトステネスの篩」と奇数最適化の軌跡。

全体像：3つのアルゴリズムの進化の軌跡



prime1 の構造：素朴な素数判定

```
def prime1(N):  
    primes = []  
    d = 2  
    l1_count = 0  
    while (d <= N):  
        isPrime = True  
        t = 2  
        while (t < d):  
            l1_count += 1  
            if (d % t == 0):  
                isPrime = False  
            t = t + 1  
        if (isPrime == True):  
            primes.append(d)  
        d = d + 1  
    return primes, l1_count
```

構造: 二重ループによる素朴な判定

- 「N以下の数dに対して、2からd-1までの数tで割り切れるか」を全て検証する。

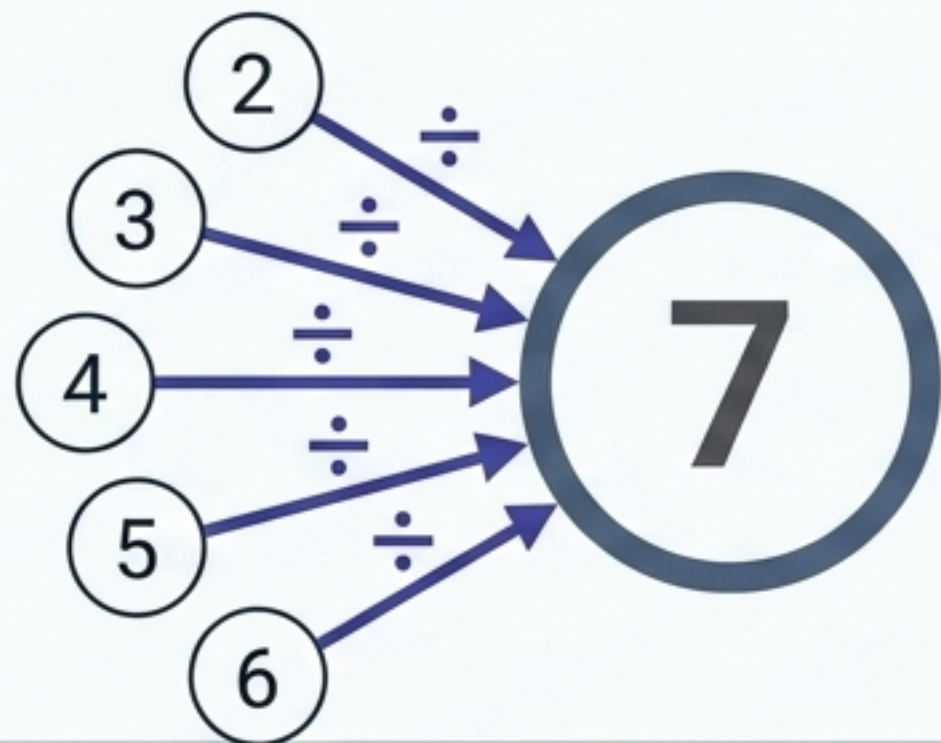
設問[ア]の解説

- 外側のループ: N-1 回実行。
- 内側のループ: 2 ~ N 回実行。

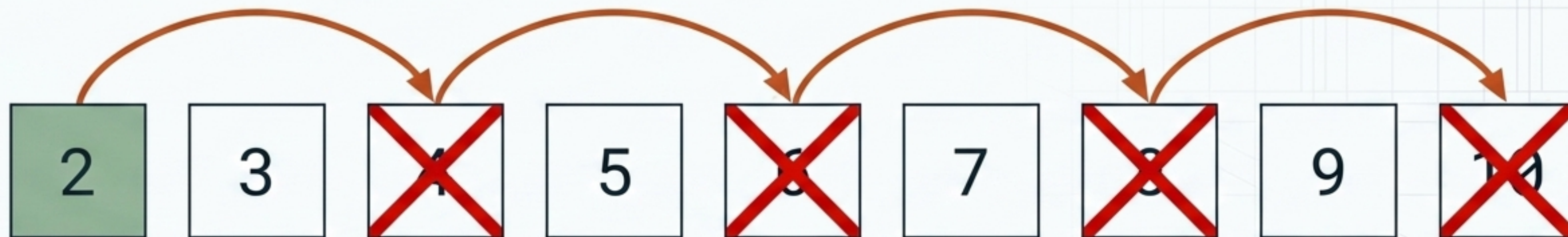
結論: 計算量は N に比例して二乗で増加するため、時間計算量は $O(N^2)$ となる。

prime2 へのパラダイムシフト

prime1: 割り算による検証



prime2: 倍数のマーキング(エラステネスの篩)



発想の転換: 「割れるか調べる」から「倍数を塗りつぶす」へ。

2以上の自然数について、自身を除く倍数は必ず合成数（素数ではない）である性質を利用する。

設問[イ]の解説：素数確定のロジック

```
for d in range(2, N + 1):  
    if isPrime[d]:  
        primes.append(d)  
        t = d * d
```

	2	3	4	5
isPrime	True	True	False	True

素数確定のロジック

- 配列 isPrime[k] が True であれば k が素数であることを表す。
- 処理が d に到達した時点で、isPrime[d] がまだ True のまま（過去の倍数として除外されていない）ならば、それは素数と確定できる。

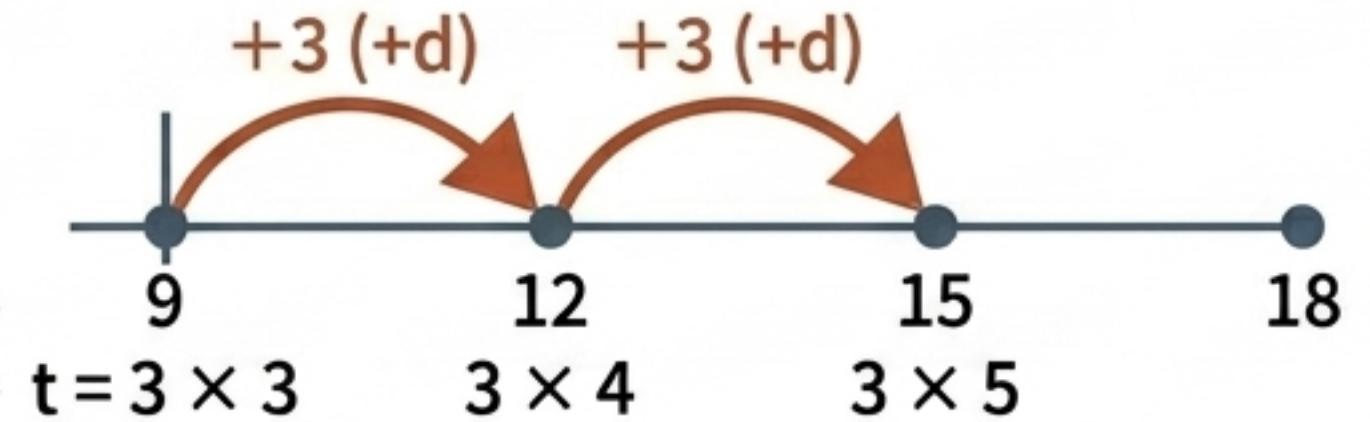
設問[イ]の正解: isPrime[d]

設問[ウ]の解説：倍数のマーキング処理

```
while t <= N:  
    isPrime[t] = False # dの倍数な  
    ので素数から除外。dの倍数をすべて「素数で  
    はない(False)」としてマークする。
```

```
t = t + d # 設問[ウ]：次の倍数へ移動
```

```
calc_num += 1 # 処理回数を加算
```



素数 d の倍数を効率的に塗りつぶす。
初期値は $t = d \times d$ 。次の倍数は
 $d \times (d+1)$ 、その次は $d \times (d+2)$ 。
つまり、現在の t に d を足し合わせる
だけで次の倍数へ移動できる。

設問[ウ]の正解: $t + d$

prime3 の構造：奇数のみに特化する

prime2 の配列
(サイズ: $N+1$)

2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	----



偶数を完全に排除

prime3 の配列
(サイズ: $N//2 + 1$)

3	5	7	9
---	---	---	---

- 奇数への特化 (Improvement 2)
- 4以上の偶数はすべて2の倍数であり、素数ではない。
- 偶数を配列から完全に排除することで、メイン処理の二重ループ実行回数と、必要な配列メモリ量を半分以下に削減する。

設問[エ]の解説：数値からインデックスへの変換

Mapping Matrix	
配列インデックス (k)	実際の数値 (d = 2k+1)
1	3
2	5
3	7

数式による逆算:

$$\begin{aligned}d &= 2k + 1 \\2k &= d - 1 \\k &= (d - 1) / 2\end{aligned}$$

問題文の論理

配列要素が奇数のみを扱うため、実際の数値 **d** と配列インデックス **k** を変換する計算式が必要。

設問[エ]の正解 (1始まり) : $(d-1) \div 2$

Python実装上の注意

Pythonの配列は0始まりであるため、コード上では `d_idx = (d - 3) // 2` として実装されている。

設問[オ]・[カ]の解説：奇数専用のスキップ処理



奇数配列におけるスキップ処理： d の倍数の次の奇数へ移動

設問[オ] 塗りつぶしインデックス

設問[エ]と同様、奇数の倍数 t を配列インデックスに戻す。

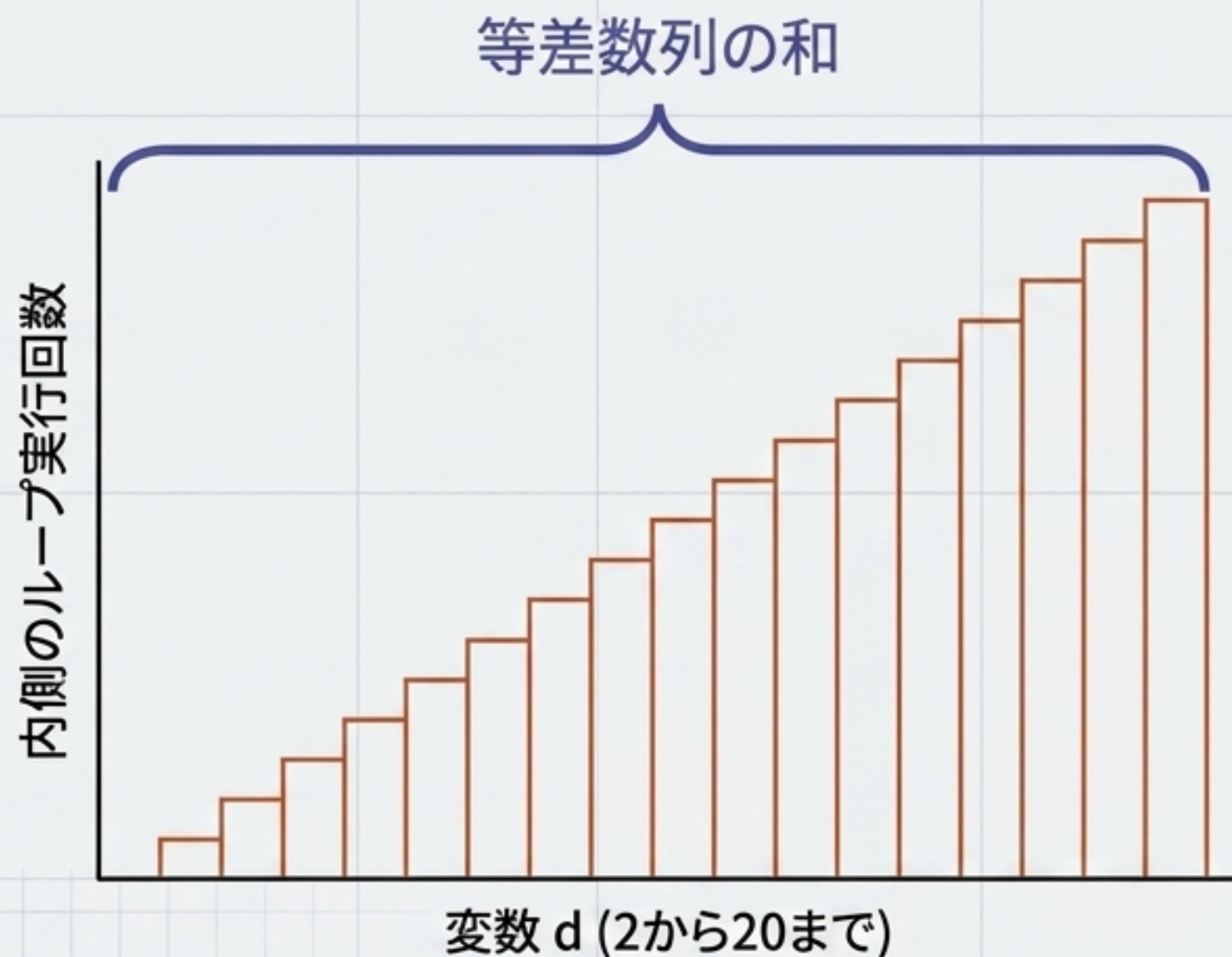
正解: $(t-1) \div 2$

設問[カ] スキップ処理の倍増

t が奇数のとき、 $t+d$ は「奇数+奇数=偶数」となり、奇数専用配列では無駄な計算になる。次の「奇数の倍数」を正確に撃ち抜くためには、 $t+2d$ のステップが必要。

正解: $t + 2 \times d$

実行回数の検証（設問4）：L1の計算



条件： $N = 20$

外側のループは $d = 2 \sim 20$ の合計19回実行。

内側のループは d の値に応じて、0回、1回、2回...と順次増加する。

計算式：

$$\begin{aligned} &0 + 1 + 2 + \dots + 18 \\ &= [18 \times (1 + 18)] / 2 = 171 \end{aligned}$$

設問4 L1の正解: 171回

実行回数の検証（設問4）：L2・L3の計算

(L2) prime2 の実行回数

	ステップ幅 +d
d=2 のとき:	2×2 から 2×10 まで (9回)
d=3 のとき:	3×3 から 3×6 まで (4回)
d=5 のとき:	5×5 = 25 > 20 (0回)

計: 13回

(L3) prime3 の実行回数

	ステップ幅 +2d
d=3 のとき:	3×3 (15), 次は 15+2×3=21 > 20 (2回)
d=5 のとき:	5×5 = 25 > 20 (0回)

計: 2回

N=20という小さな数でも、ループステップの最適化（+d から +2d）により、無駄な処理が 13回 から 2回 へと劇的に削減されることが証明される。

アルゴリズム比較マトリクス

指標	prime1 (ベース)	prime2 (篩)	prime3 (奇数特化)
概念	割り算全探索	倍数マーキング	奇数のみマーキング
時間計算量	$O(N^2)$	$O(N \log \log N)$	より高速
配列サイズ	なし	$N+1$	$N/2 + 1$ (半分)
ループ回数 ($N=20$)	171回	13回	2回

考察：メモリ（配列）を消費することで時間を買い（prime1 → prime2）、さらに論理的マッピング（奇数限定）を用いることで時間とメモリの両方を極限まで削ぎ落とした（prime2 → prime3）。

学習のポイント (Summary)



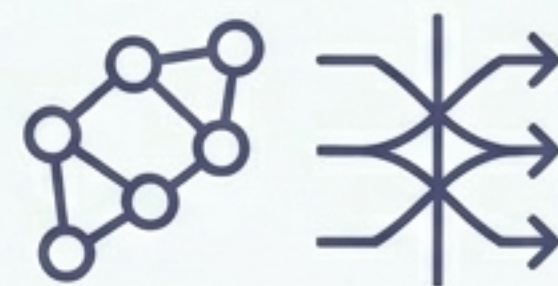
判定から除外へ

素数判定は「割れるか」を探すより、「倍数を塗りつぶす」方が圧倒的に速い。発想のパラダイムシフトを理解する。



ループの刻み幅が命

対象を奇数に絞るなら、イテレーションも $+2d$ で偶数をスキップしなければ真の最適化にはならない。



インデックス・マッピング

配列は単なる箱ではない。抽象的な配列インデックス k と実際の数値 d を $k = (d-1)/2$ の数式で変換する思考が応用情報の鍵。

アルゴリズムの進化は、コード行数ではなく「数学的性質の理解」から生まれる。