

Target: 2023s_Pythonカラツバ法.txt

$O(N^{1.58})$

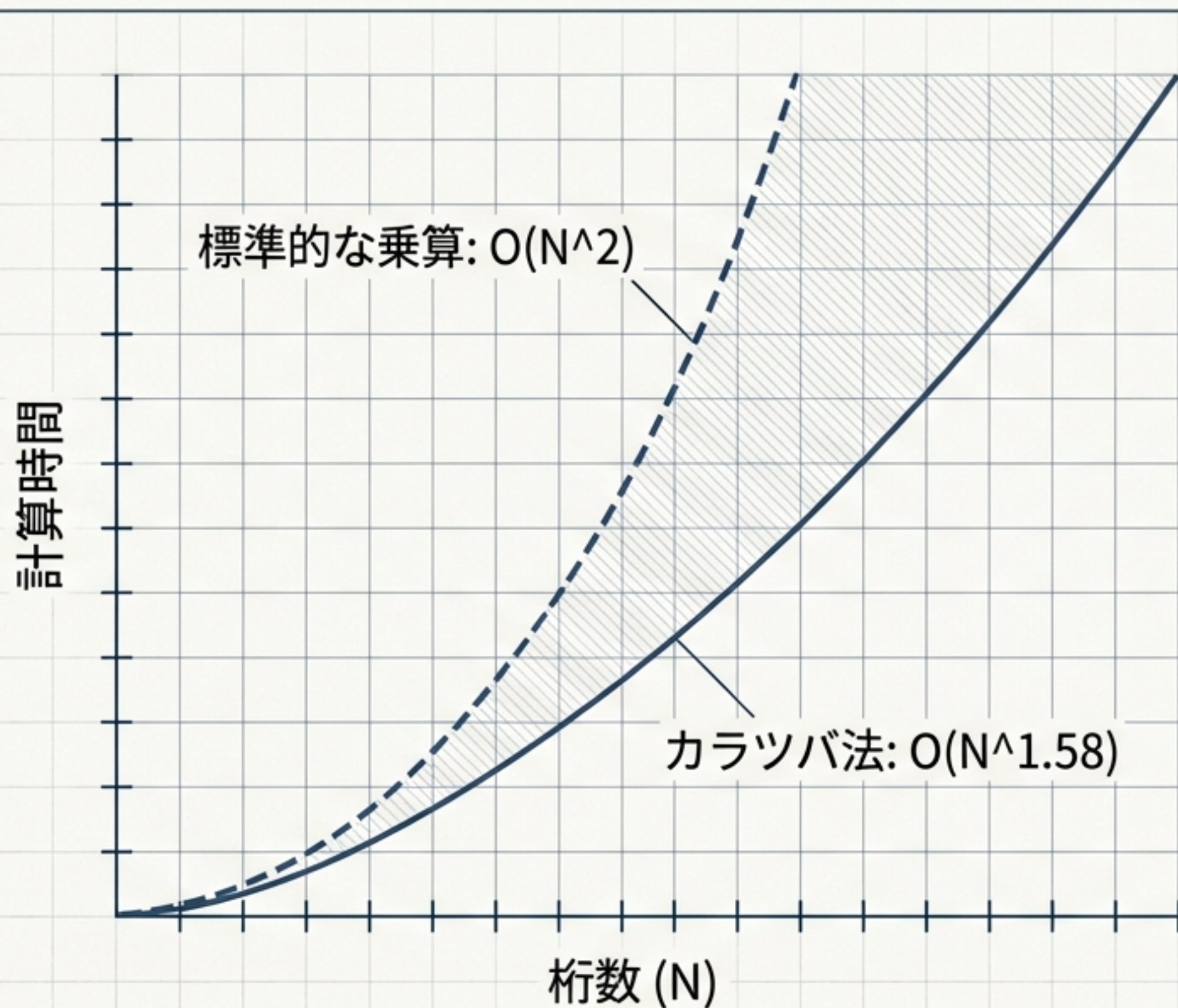


```
def karatsuba(x, y):  
    if x < 10 or y < 10:  
        return x * y;  
  
    n = max(len(str(x)), len(str(y)));  
    m = n // 2;  
  
    x_h = x // 10**m;  
    x_l = x % 10**m;  
    y_h = y // 10**m;  
    y_l = y % 10**m;  
  
    a = karatsuba(x_h, y_h);  
    b = karatsuba(x_l, y_l);  
    c = karatsuba(x_h + x_l, y_h + y_l);  
  
    return a * 10**(2*m) + (c - a - b) * 10**m + b;
```

2023s Pythonコード详解：カラツバ法 (Karatsuba Algorithm) の完全解剖

多倍長整数のツリー構造と再帰的乗算アルゴリズムの視覚的理解

カラツバ法の優位性：計算量 $O(N^2)$ から $O(N^{1.58})$ への壁を越える



トレードオフの魔法

乗算（コスト大）の回数を減らし、加減算（コスト小）に置き換えることで、計算量の壁を突破するアルゴリズム。

2023sコードの目標

非常に巨大な桁数（多倍長整数）を扱うため、1次元配列とツリー（木）構造を用いてこの数学的最適化をPythonで効率的に実装すること。

コードアーキテクチャの全容（4つの機能ブロック）

Block 4: メインアルゴリズム (Core Logic)

elements, layer_top, 再帰的カラツバ演算

Block 3: 補助関数 (Helper Functions)

pow_func, mod, new_elem, 文字列変換

データ定義から数学的抽象化
へと積み上がる依存関係

Block 1: 構造体 (Data Structures)

LargeInt, Node

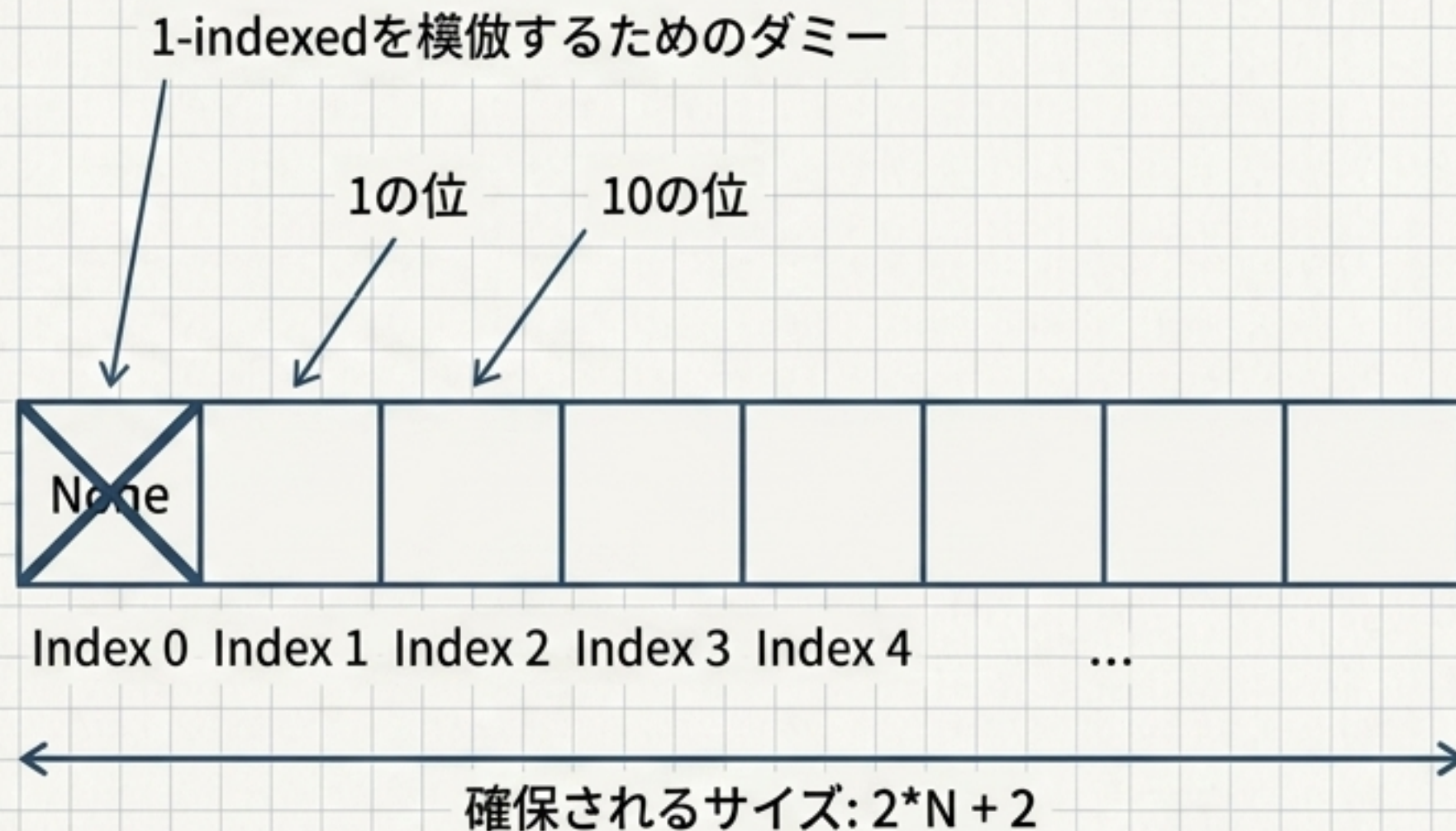
Block 2: 基本操作 (Basic Operations)

add, sub, left, right, shift, carry

Block 1: 構造体の定義と多倍長メモリ管理

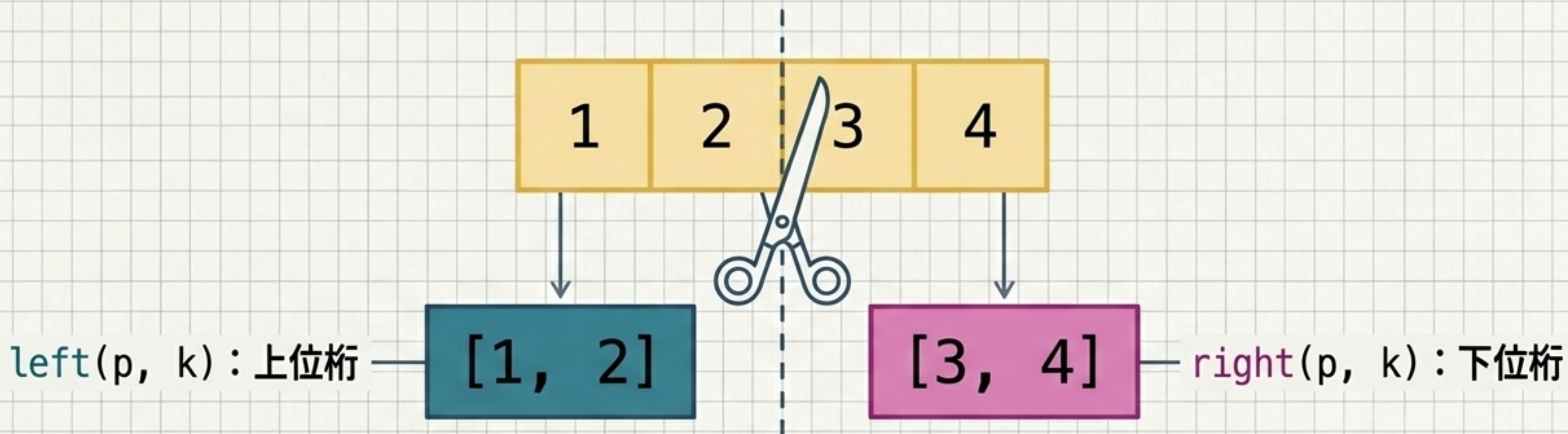
```
class LargeInt:  
    self.values = [None] + [0] * (2 * n + 2)
```

```
class Node:  
    self.val1 = v1    # 左辺  
    self.val2 = v2    # 右辺  
    self.result = ... # 結果
```



設問5のロジック：N桁 × N桁 の乗算結果は最大で 2N 桁になるため、十分なメモリ空間を事前に確保する。

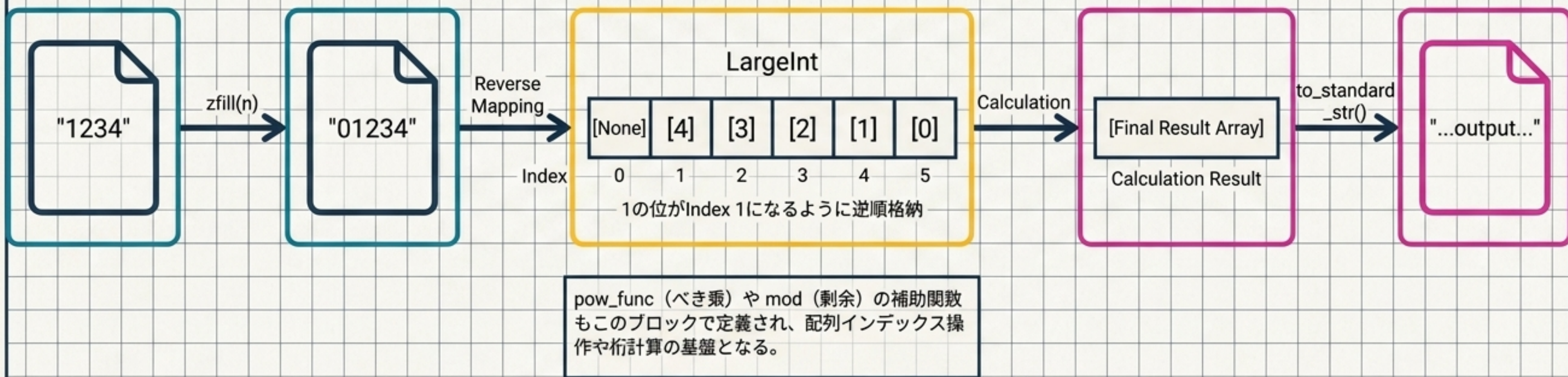
Block 2: 多倍長整数の基本操作（桁の分離と演算）



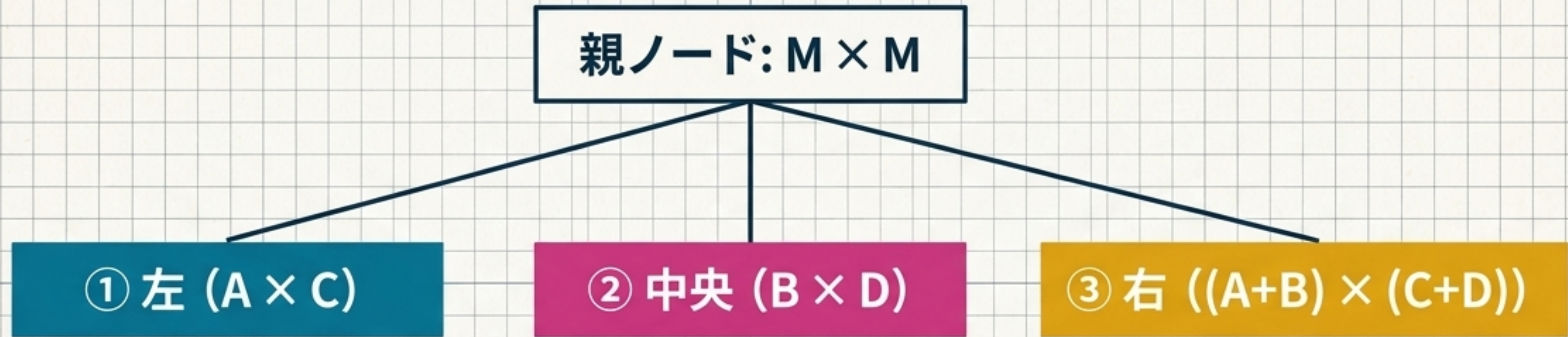
最重要ルール：遅延評価 (Deferred Carry)

加算・減算の計算時は「桁ごとの加減算のみ」を行い、配列内に『14』のような2桁の値が一時的に混在することを許容する。繰り上がり (Carry) の正規化処理はツリーの最期で一括して行う。

Block 3: 文字列から配列への変換パイプライン



理論的背景：カラツバ・ツリーの3分岐ルール



設問1の例：34 × 78

A=3, B=4, C=7, D=8

└→ 【ア】 左端 = 3 × 7

設問1の例：46 × 134

未繰り上がり状態（10の位=12, 1の位=14）として分解

A=4, B=6, C=12, D=14

└→ 【イ】 左端 = 4 × 12

Block 4.1: 子ノードの生成 (空欄 [オ] ・ [カ] の解法)

```
pe = elements[...]  
elements[tidx] = new_elem(cn, left([オ], cn), left([カ], cn))  
elements[tidx+1] = new_elem(cn, right([オ], cn), right([カ], cn))  
elements[tidx+2] = new_elem(cn, lradd([オ], cn), lradd([カ], cn))
```

① $A \times C$ の生成

② $B \times D$ の生成

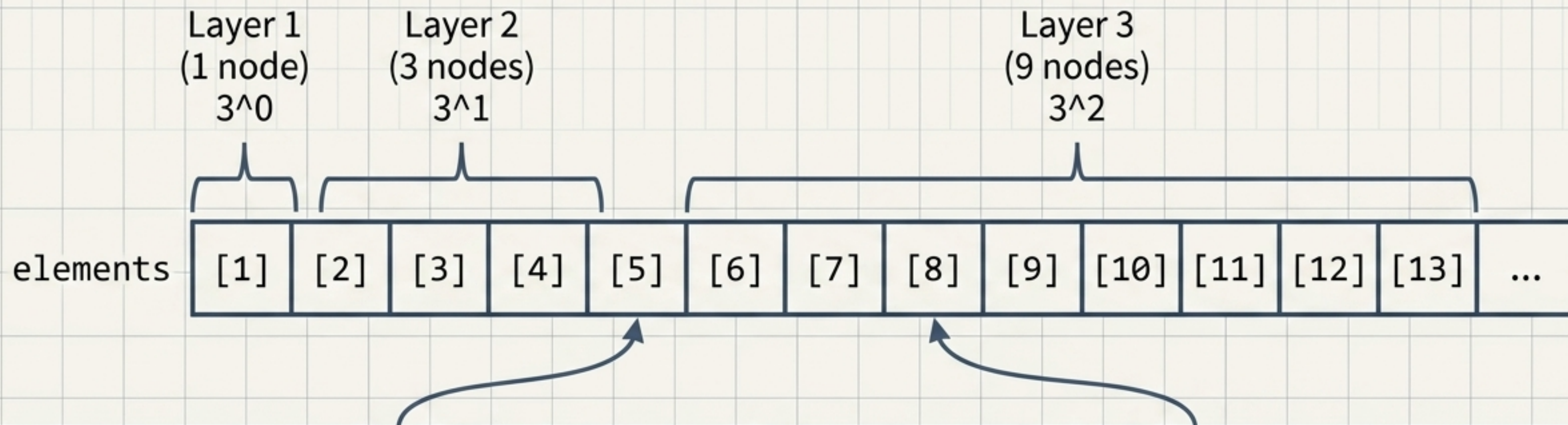
③ $(A+B) \times (C+D)$ の生成

引数の正体

[オ] = $pe.val1$ (親の左辺値)

[カ] = $pe.val2$ (親の右辺値)

Block 4.2: 2Dツリーの1D配列化（インデックスの計算）



各層の左端インデックス layer_top (空欄 [ウ])

子ノードの格納先オフセット tid_x (空欄 [エ])

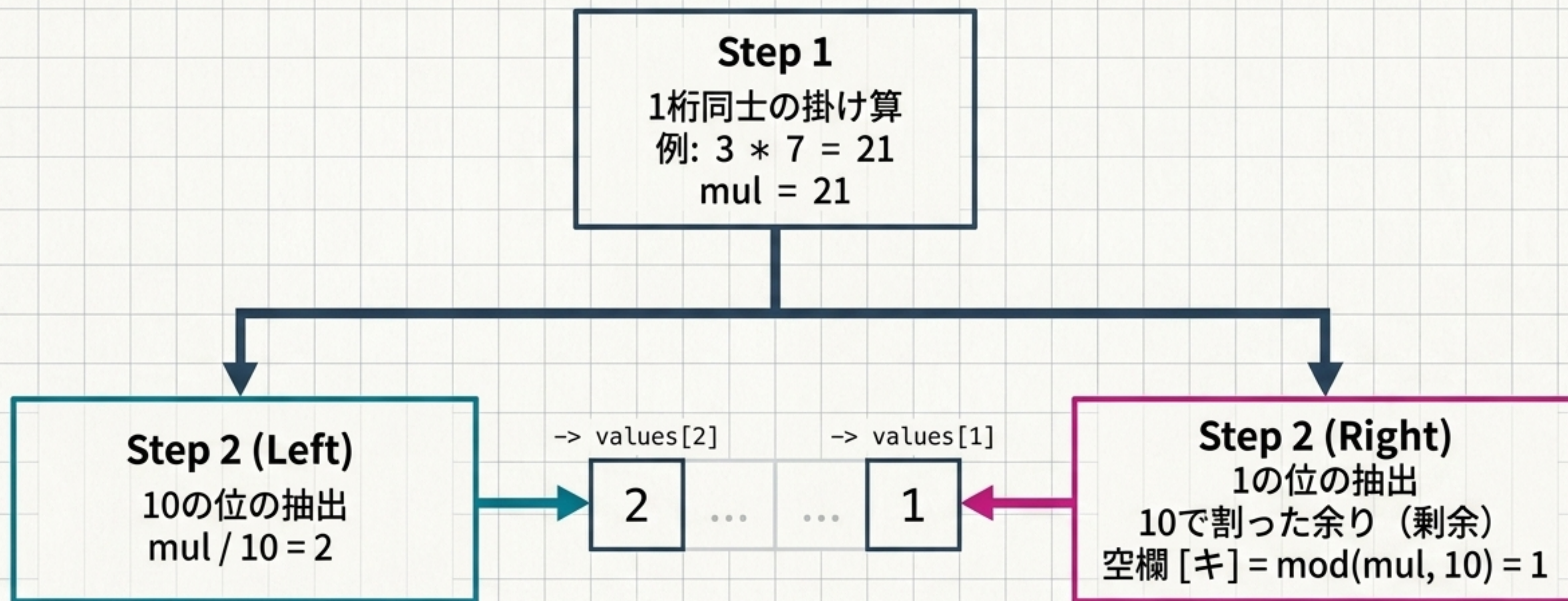
現在の層のノード数（べき乗）分だけ増加。

親が1つ進むごとに子ノードは3つ分右にずれる。

式: $\text{pow}(3, i - 1)$

式: $3 * (i - 1)$

Block 4.3: 最下層のベースケース計算（空欄 [キ] の解法）



両方の値が10未満（1桁）になった段階で再帰を終了し、結果を2桁の多倍長配列として正規化格納する。

Block 4.4: カラツバ法の核 $\gamma - a - \beta$ (空欄 [ク]・[ケ] の解法)

$$s2 = \gamma - a - \beta$$

引かれる数 [ク] = 子ノード③ (γ)
= `elements[cidx + 2]`

引く数 [ケ] = 子ノード① (a)
= `elements[cidx]`

引く数 = 子ノード② (β)

$$s1 = \text{sub}([ク], [ケ])$$

$$s2 = \text{sub}(s1, \text{elements}[cidx + 1])$$

逆算思考：s2 で β を引くため、s1 の段階で $(\gamma - a)$ を計算しておく必要がある。

Block 4.5: 計算結果の再構築とシフト演算

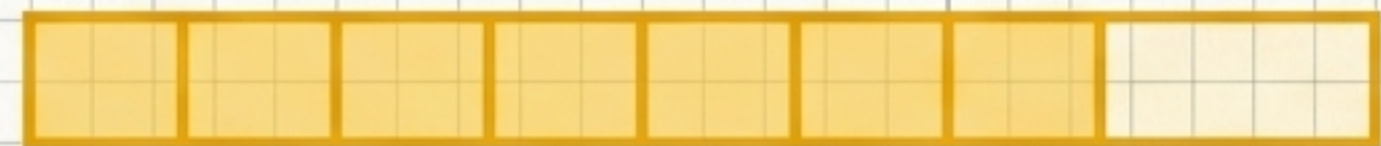
$$a \times 10^K + (\gamma - a - \beta) \times 10^{K/2} + \beta$$

a の配列



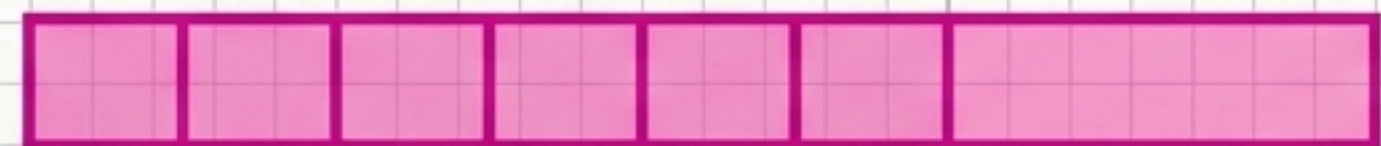
$\text{shift}(a, K) : 00\dots0$ (K個)

$(\gamma - a - \beta)$ の配列



$\text{shift}(\dots, K/2) : 00\dots0$ (K/2個)

β の配列



$\text{shift}(p, k)$ 関数：配列要素を上位へずらし、下位を0で埋めることで、乗算コストをかけずに桁の重み付けを実現する。

総合対照表：数学モデルとPythonコードの完全マッピング

数式記号	ツリーノード位置	Python変数	具体例（層2：46x134）
α	子ノード①（左）	<code>elements[cidx]</code>	48
β	子ノード②（中央）	<code>elements[cidx+1]</code>	84
γ	子ノード③（右）	<code>elements[cidx+2]</code>	260

※この対照表は、2023s問題における理論と実装の架け橋となるマスターキーです。

最終処理：繰り上がり (Carry) の一括適用

遅延評価された生の配列 (2桁の数値が混在)



10で割った商を上位へ加算
10で割った剰余を現在桁に残す



carry(p) 適用後：各要素が1桁に正規化された完全な多倍長整数

結論：数学的最適化とデータ構造の融合



計算量の劇的圧縮

乗算回数を削減するKaratsuba法により、多倍長乗算の計算コストを大幅に効率化。

データ構造の工夫

2次元的なツリー構造を1次元配列に平坦化し、キャリー処理を最後まで遅延させることで、メモリと実行速度を両立。

アルゴリズムの未来

本コードは分割統治法の美しい実例。さらに巨大な数にはToom-Cook法や高速フーリエ変換(FFT)へと発展する。