

2つのアプローチ：総当たり vs. 賢いスキップ

方法1: 単純な照合
(Naive)

方法2: 効率的な照合
(Boyer-Moore簡略版)

基本動作

不一致なら「1文字」だけ進める

不一致なら「表」を参照して
「大きく」進める

比較回数

テキスト長に比例して増大（非効率）

スキップにより劇的に減少（高効率）

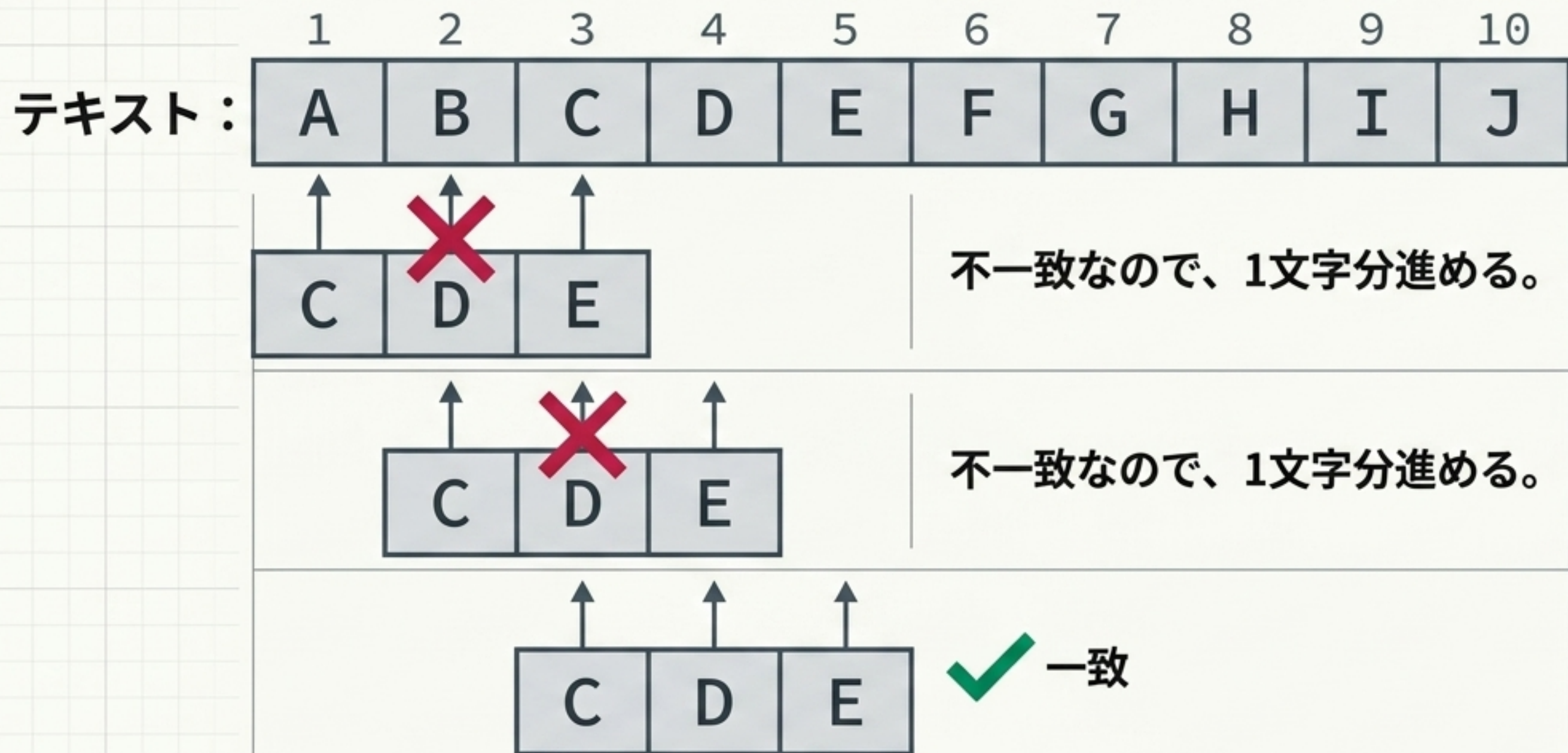
コードの
複雑さ

単純な2重ループ

3つの独立したブロック（表作成・比較・
スキップ）による高度な設計

本資料では、直感的な「方法1」から入り、アルゴリズムの工夫が光る
「方法2」の内部構造をコードレベルで解剖します。

方法1のメカニズム：1文字ずつ確実に進める



照合に失敗するたびに、無条件で開始位置を「1」だけズラす。
确实だが、すでに調べた文字の情報を全く活かしていない。

境界値の算出：[ア]と[イ]の導出

コードの役割

for i を 1 から [ア] まで

テキスト側の開始位置の限界

for j を 1 から [イ] まで

パターン内の何文字目か

視覚的な数式導出



[ア]の導出

$$11 - 3 + 1 = 9$$

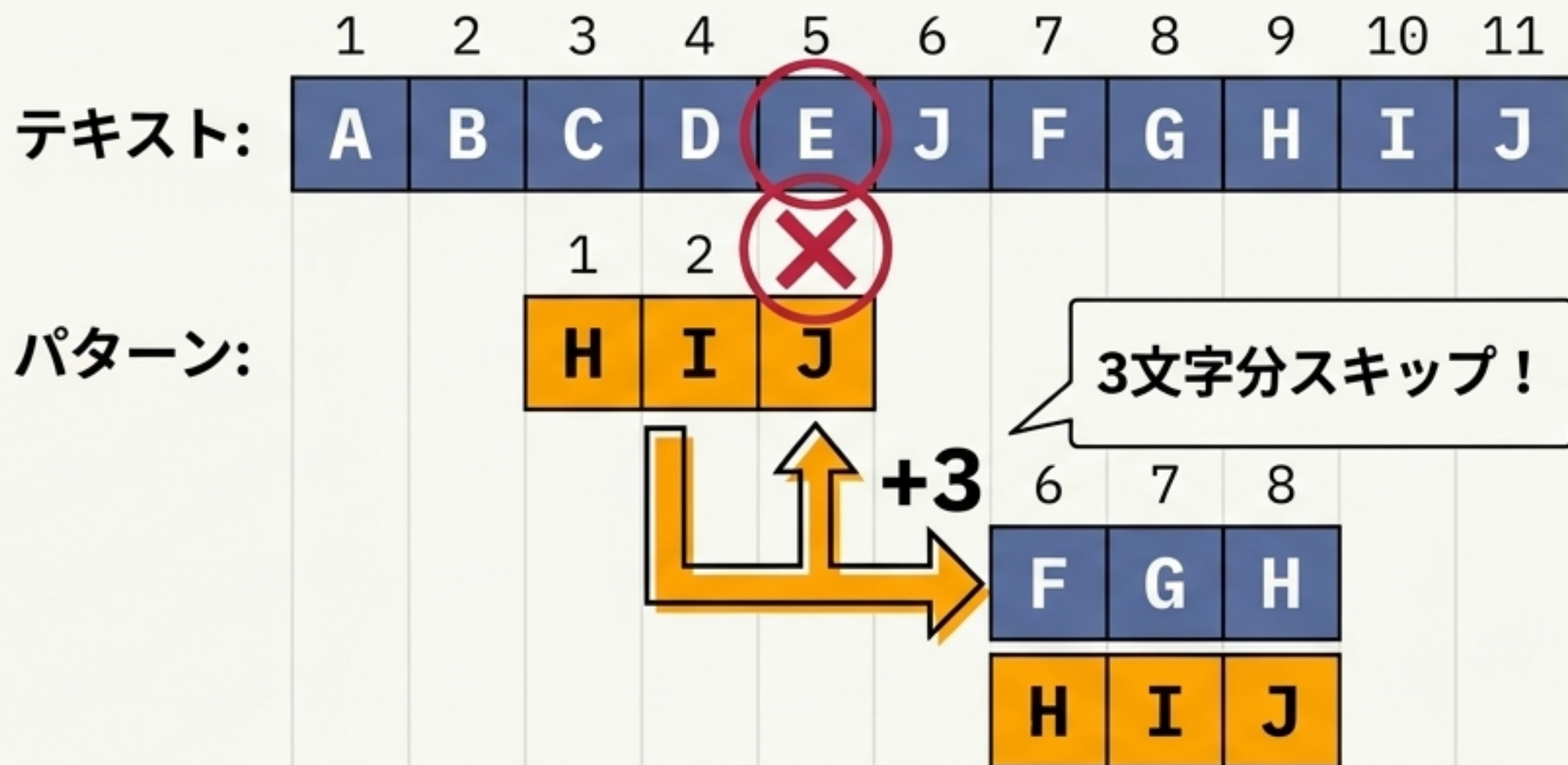
$$[ア] = T.length - P.length + 1$$

[イ]の導出

パターンの全文字を比較するため、
ループはパターンの長さ分だけ回る。

$$[イ] = P.length$$

発想の転換：失敗から「スキップ幅」を学習する



スキップ判定

判定文字「C」はパターン
「H I J」に含まれるか?

-> 含まれない!

アクション:
パターンの長さ (3文字)
分、一気にスキップ可能。

次の比較位置を「+1」ではなく、「不一致となった文字がパターン内のどこにあるか」から逆算する。これがスキップテーブルの基本原理。

スキップテーブルの構築：[エ] と [オ] の導出

スキップ数 = (パターンの長さ P.length) - (末尾から最も近い一致文字の位置)

判定文字	H	I	P	O	T	A	M	U	S	その他の文字
スキップ数	[エ]	9	[オ]	5	4	3	2	1	11	11

[エ] 判定文字「H」の場合
位置: 1文字目のみ存在。

計算: $11 - 1 = 10$

HIPOPOOTAMUS

答え: [エ] = 10

[オ] 判定文字「P」の場合
位置: 3文字目と 5文字目に存在。

ルール: 「末尾から最も近いもの（末尾は除く）」を採用 → 5文字目。

計算: $11 - 5 = 6$

HIP0POOTAMUS

答え: [オ] = 6

コードアーキテクチャ：効率的アルゴリズム（方法2）の全体像

```
function search2(T,P)
  for x を1から P.length まで1ずつ増やす
    C[x] ← "" // "" は空白文字を表す
    D[x] ← P.length
  endfor
  // (1): 表の作成
  for j を1から P.length-1 まで1ずつ増やす
    for k を1から j まで1ずつ増やす
      if C[k]と P[j]が等しい or C[k]と" "が等しい
        C[k] ← P[j]
        D[k] ← j
      endif
    endfor
  endfor // (1)の終わり

  // 文字列を照合する
  i ← 1
  while i が ア 以下である
    for j を1から イ まで1ずつ増やす
      if T[i+j-1]と P[j]が等しい
        if j と イ が等しい
          return i
        break
      endif
    endfor // (2)の終わり

    // (3): スキップ数の決定
    for k を1から P.length まで1ずつ増やす
      if C[k]と T[i+P.length-1]が等しい or C[k]と" "が等しい
        d ← D[k]
        break
      endif
    endfor
    i ← i+d
  endwhile
  return -1
endfunction
```

Block 1: (1) 表の作成

役割: 与えられたパターンから、各文字ごとの「スキップ数（配列D）」を事前計算する。

Block 2: (2) 文字の比較

役割: 現在の開始位置から、テキストとパターンが一致するか検証する。

Block 3: (3) スキップ数の決定

役割: 不一致だった場合、Block 1で作った表（配列D）を参照し、次の開始位置 i を更新する。

Block 1 解剖：スキップ数の数式化 [ウ] の導出

$D[k] \leftarrow [\text{ウ}] //$ スキップ数を代入

H (j=1) I (j=2) J (j=3)

長さ = 3

H (j=1) の場合、末尾までの距離は $3 - 1 = 2$ 。

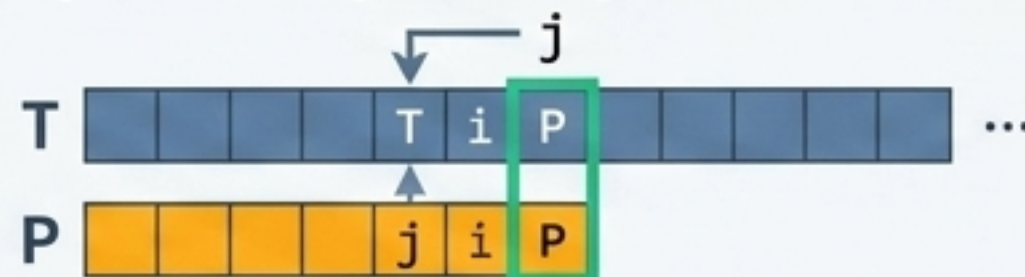
数式化: $[\text{ウ}] = P.\text{length} - j$

この事前計算があるからこそ、照合中のスキップが高速に行える。

Block 2 & 3 解剖：比較とダイナミックなジャンプ

```
while i が ア 以下である
// (2):文字の比較
for j を1から イ まで1ずつ増やす
    if T[i+j-1]と P[j]が等しい
        ...
    else
        break
    endif
endfor
// (3):スキップ数の決定
for k を1から P.length まで1ずつ増やす
    if C[k]と T[i+P.length-1]が等しい or C[k]と " "が等しい
        d ← D[k]
        break
    endif
endfor
i ← i+d
endwhile
endfunction
```

Step 1: if T[i+j-1] と P[j] が等しい (一致検証)



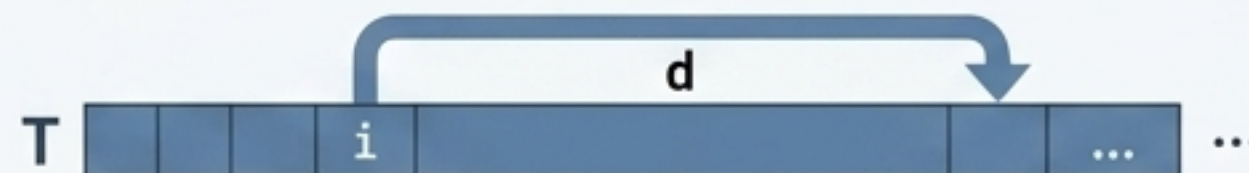
Step 2: 不一致なら break して Block 3 へ



Step 3: d ← D[k] (表からスキップ数を取得)

D	0	1	2	3	4	5	6	7	...	n
	1	2	3	4	5	6	7	d	→	0

Step 4: i ← i + d (大きくジャンプ)



方法1の $i++$ が、方法2では $i \leftarrow i + d$ に進化。無駄な比較を回避する。

効率性の証明：実行回数比較テスト

テキスト: **PICKLED_PEPPER**

パターン: **PEP**

挑戦者1: 方法1 (単純な照合)

測定変数: α (比較回数)

挑戦者2: 方法2 (効率的な照合)

測定変数: β (比較回数)

どちらのアルゴリズムが、より少ない「比較演算」で
'PEP' を見つけ出せるか？

α の実行回数：総当たりの現実

	P	I	C	K	K	L	E	D		P	E	P	P	E	R
i=1	P	✓	✗	I					→	P	E	P	P	E	R
i=2	P	→✗	I						→	P	E	P	P	E	R
i=3	P	I	→✗	C					→	P	E	P	P	E	R
i=4	P	I	C	→✗	K				→	P	E	P	P	E	R
i=5	P	I	C	K	→✗	L	✓		→	P	E	P	P	E	R
i=6	P	I	C	K		→✗	E	✓	→	P	E	P	P	E	R
i=7	P	I	C	K	L		→✗	D	✓	P	E	P	P	E	R
i=8	P	I	C	K	L	E	D	→✗	P	E	P	P	E	R	
i=9	P	I	C	K	L	E	E	D		P	E	P	P	E	R

i=1からi=8までの試行:
合計9回

i=9での試行: 3回

9 + 3 = 12

α の実行回数 = 12回

観察: 先頭の1文字目が不一致でも、律儀に毎回比較を行っている。

βの実行回数：Boyer-Moore法による大躍進

スキップ数

2	1	3	3
---	---	---	---

P	I	C	K	L	E	D	_	P	E	P	P	E	R
↓	↓												
P	E												
			↑										
			P	↑									
				P			↑						
						P							
								↑	↑	↑			
								P	E	P			

事前準備: PEPのスキップ表 (P=0, E=1, その他=3)

Row 1: PIC -> C (その他) 幅3 -> 比較2回

Row 2: KLE -> E (パターンの2文字目) 幅1 -> 比較1回

Row 3: LED -> D (その他) 幅3 -> 比較1回

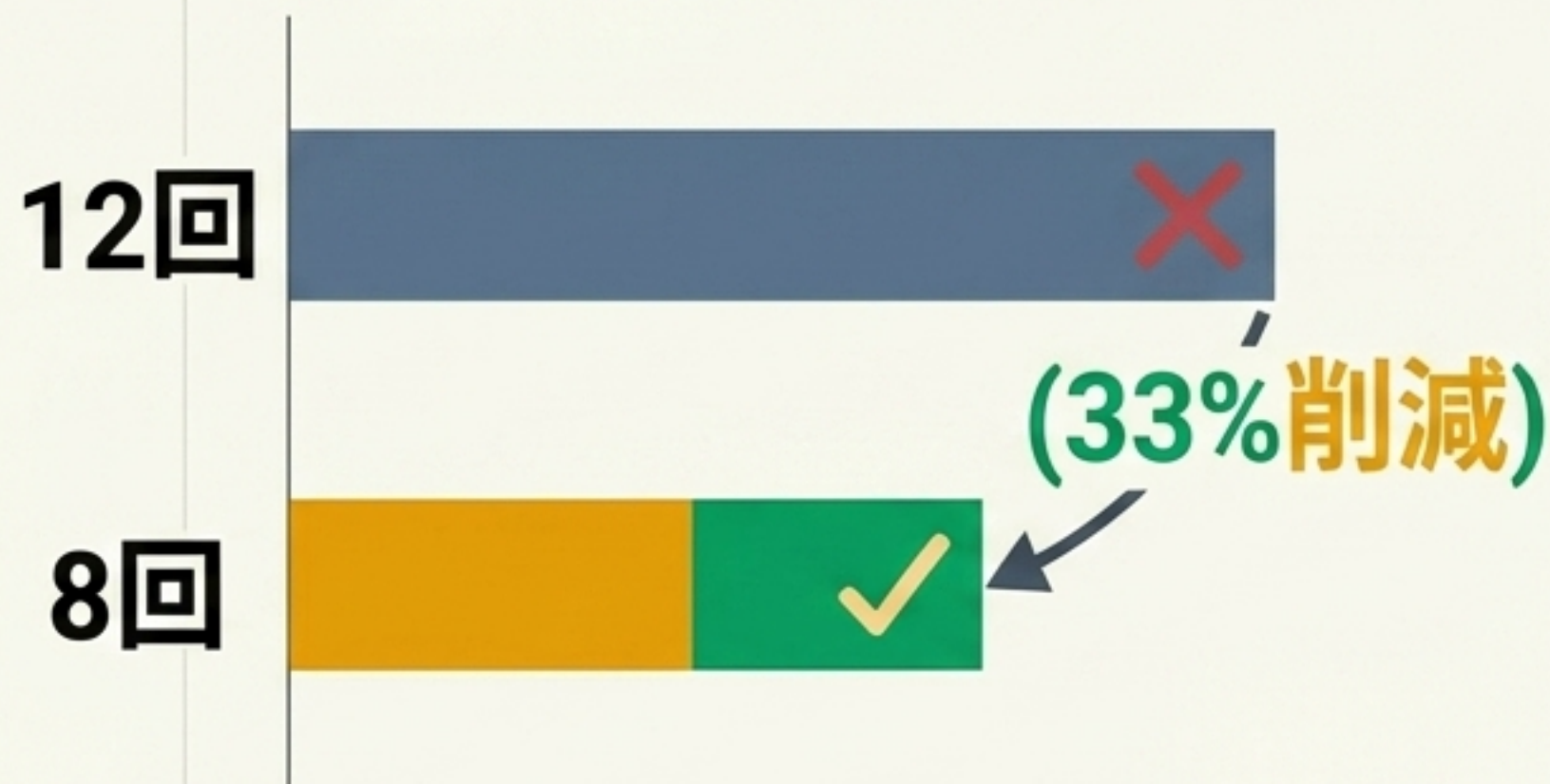
Row 4: _PE -> E (パターンの2文字目) 幅1 -> 比較1回

Row 5: PEP -> 一致 -> 比較3回

計算: $2 + 1 + 1 + 1 + 3 = 8$

βの実行回数 = 8回

結論：設計（アーキテクチャ）が性能を決める



情報への着目: 「合っているか」だけでなく「なぜ間違えたか」をデータとして活用。

事前計算の力: Block 1で作るスキップ表が、ループ処理の中核で絶大な威力を発揮。

計算量への影響: 長大なテキストになるほど、このアルゴリズムの真価が発揮される。

アルゴリズムの進化とは、コードを短くすることではなく、手元にあるデータをいかに賢く利用するかという「設計思想の進化」である。