

[<SYSTEM_ANALYSIS>] →

部分と問題の解説：力任せの探索 から枝刈りによる最適化まで

Pythonコードで読み解く探索アルゴリズムの設計図

← [OPTIMIZATION_PATH]

コアとなる課題：目標値に最も近い組合せを見つけ出す

入力 (Input):

指定された n 個の異なる自然数
(例: 10, 34, 55, 77)

目標 (Target):

指定された目標値 X (例: 100)

条件 (Constraints):

各数は1回しか選択できない。
合計は X を超えても下回ってもよい。

出力 (Output):

目標に最も近い組合せ
(例: 10, 34, 55 を選択し、合計99)

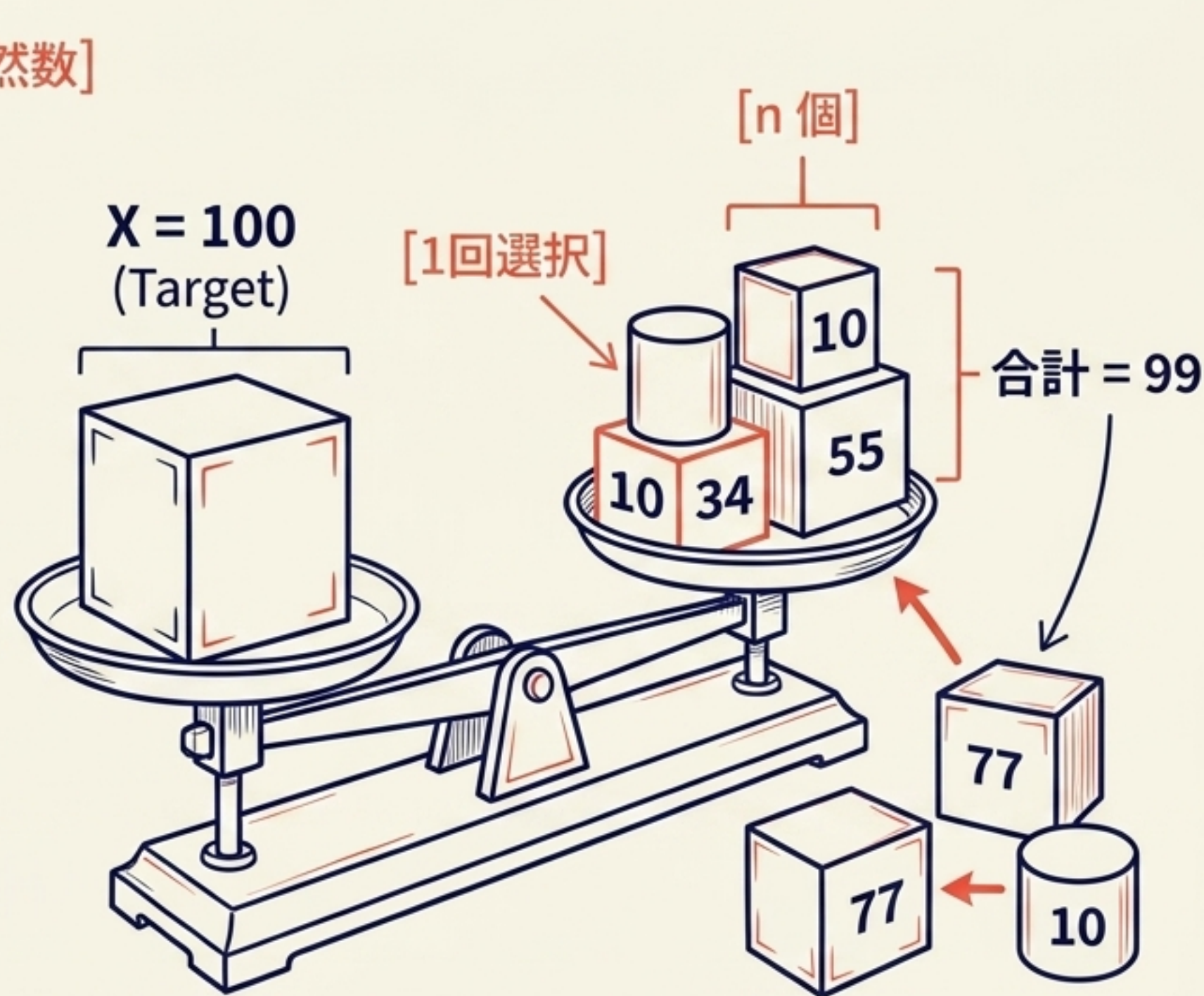
[自然数]

[n 個]

$X = 100$
(Target)

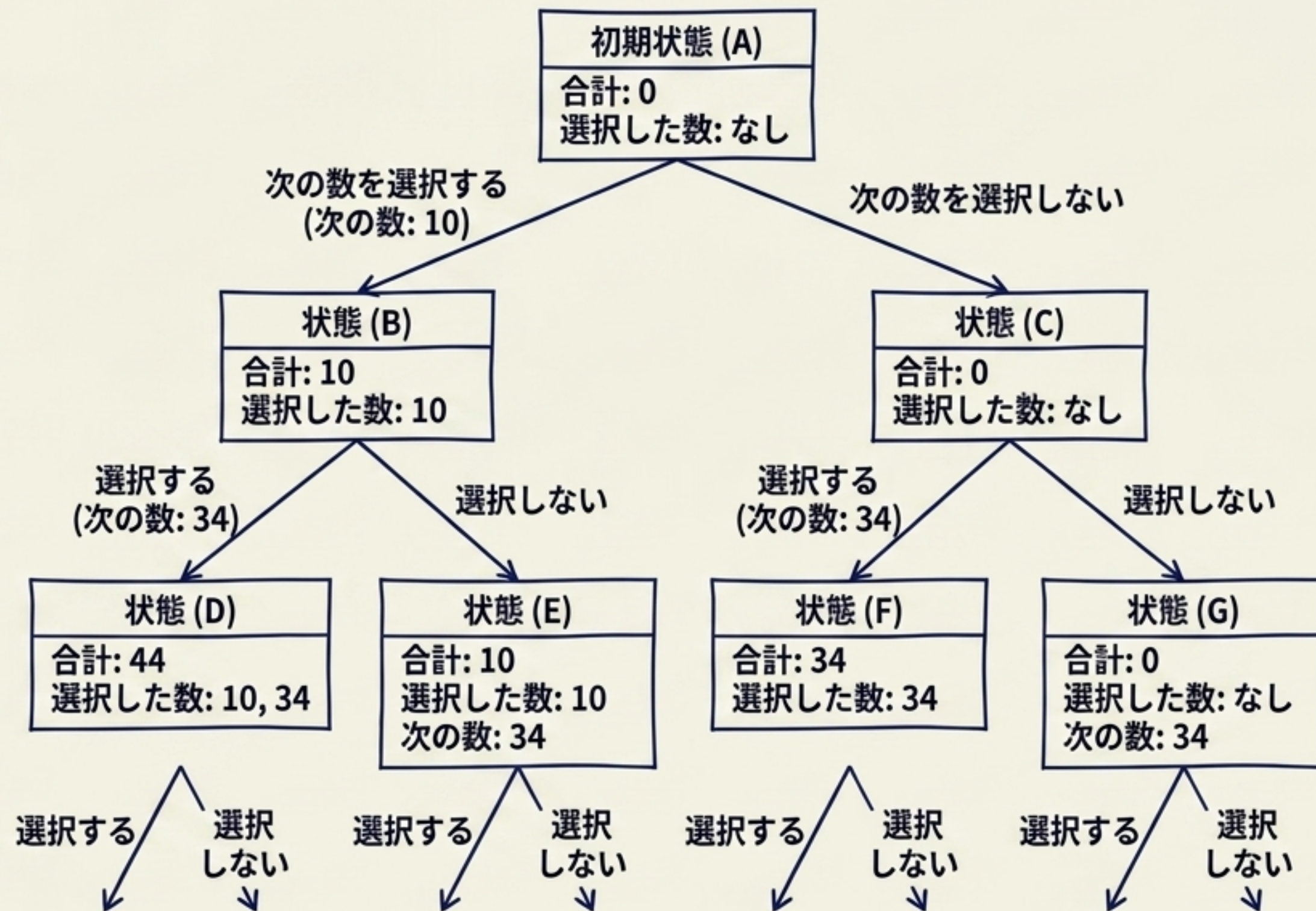
[1回選択]

合計 = 99



メンタルモデル：場合分けの状態空間木

- **ノード (Node):** 探索中の特定の状態（合計値、選択済みの数、次に見る数）を表す。
- **分岐 (Branching):** 各ステップで常に「選択する（左）」「選択しない（右）」の2つの未来が生成される。
- **網羅性 (Exhaustiveness):** 木の末端（葉）まで辿ることによって、全ての組合せパターンが網羅される。



コード解剖1：ベースライン・アルゴリズムの全体設計図

[MODULE 1: 初期データ (Initialization)]

- 配列: [0, 10, 34, 55, 77]
- 目標値: target = 100

[MODULE 2: 状態管理 (State Management)]

- Status クラス
- 各ノードのメモリ表現

[MODULE 3: 探索エンジン (Search Engine)]

- treeSearch() 関数
- キューを用いた状態生成と評価のループ

「状態」オブジェクトが探索エンジンに投入され、新たな「状態」が生成されるサイクル。

コード解剖2：状態（Status）のメモリ内表現

```
class Status:
    def __init__(self, total=0,
                  selected_numbers=None,
                  next_index=1):
        self.total = total
        self.selectedNumbers = selected_numbers if
        selected_numbers else []
        self.nextIndex = next_index
```

状態 (D)

[合計: 44]

[選択した数: [10, 34]]

[次の数の添字: 3]

コード解剖3：状態の分岐（生成ロジック）

現在の状態 (s)

ルートA：選択する

ルートB：選択しない

nextStatus1(s)

```
new_total = s.total + numbers[s.nextIndex]
new_selected = s.selectedNumbers +
[numbers[s.nextIndex]]
new_next_index = s.nextIndex + 1
```

- total に現在の数を加算。
- selectedNumbers に現在の数を追加。
- nextIndex を1つ進める。

nextStatus2(s)

```
new_next_index = s.nextIndex + 1
return Status(s.total, s.selectedNumbers,
new_next_index)
```

- total と selectedNumbers は維持。
- nextIndex を1つ進める。

コード解剖4：探索エンジンのループ処理

while queue: (キューが空になるまで反復)

キューから状態を取得 (Dequeue)

```
currentStatus = queue.popleft()
```



解答候補の評価 (Evaluate)

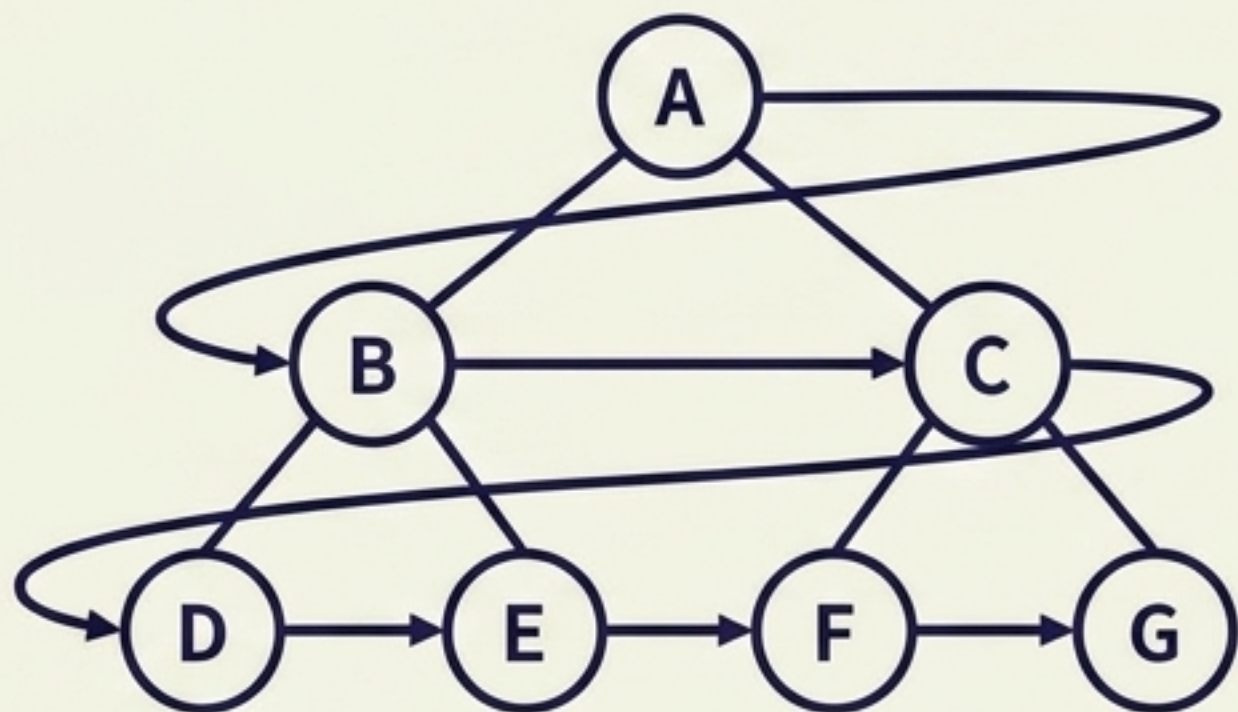
```
if abs(target - ansStatus.total) > abs(target - currentStatus.total):  
    ansStatus = currentStatus
```

子ノードの生成とキューへの追加 (Enqueue)

```
queue.append(nextStatus1(currentStatus))  
queue.append(nextStatus2(currentStatus))
```

エンジンの挙動：キュー（幅優先） vs スタック（深さ優先）

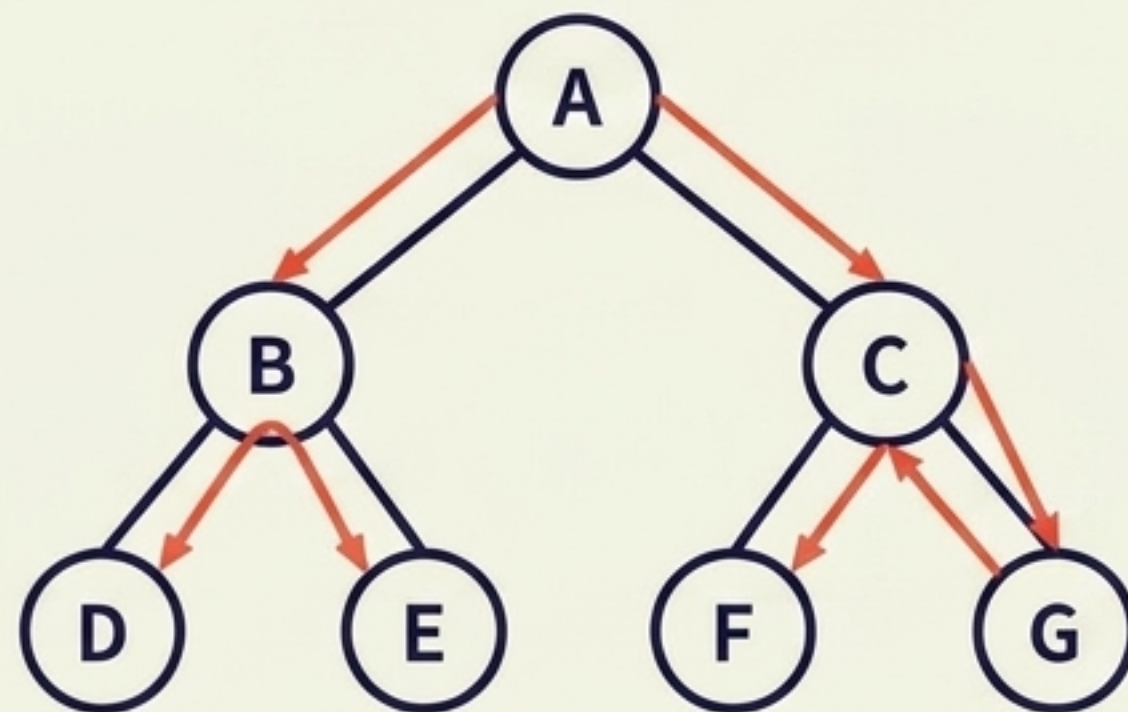
幅優先探索 (BFS) / Queue



探索順序: A -> B -> C -> D -> E -> F -> G

- ベースコードは `collections.deque()` をキューとして使用し、浅い階層から網羅的に探索する。

深さ優先探索 (DFS) / Stack



探索順序: A -> C -> G -> F -> B -> E -> D

- 最深部まで一気に潜る特性がある。

ボトルネック：組合せ爆発（Combinatorial Explosion）

完全2分木の性質：

- ・要素数が n 個のとき、木の深さは n 。
- ・最下層（葉）のノード数は 2^n 個。

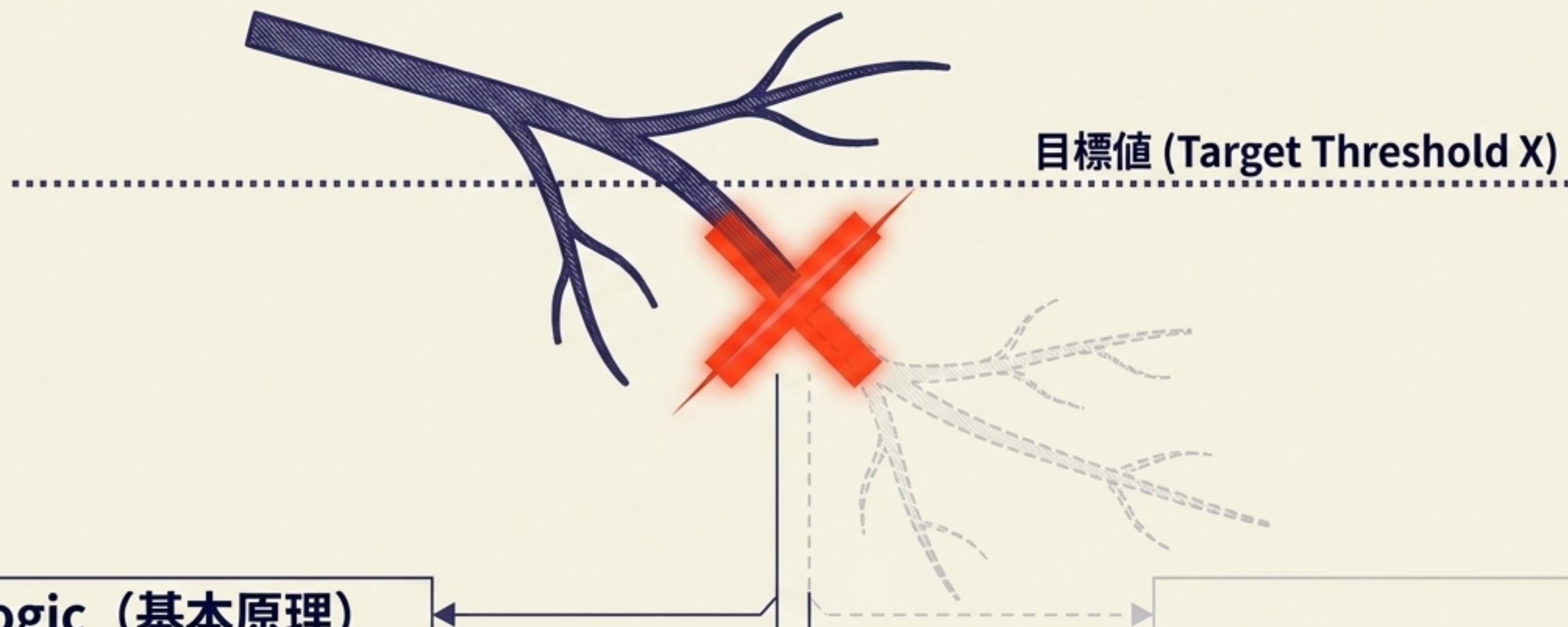
メモリと時間の壁

n が大きくなると、キューに同時格納される状態数が指数関数的に増大する。

最大メモリ使用量 $= 2^n \times m$

力任せの全探索（Brute Force）は破綻する。

最適化のコンセプト：「枝刈り (Pruning)」



Core Logic (基本原理)

Core Logic (基本原理)

「取得した状態」の合計が目標 X 以上になった場合、それ以降のステップでさらに数を選択しても、合計値は目標から遠ざかるだけである。

Result (効果)

決して「正解の候補」にならない未来の分岐を計算前に放棄（カット）し、探索回数とメモリ使用量を劇的に削減する。

コード解剖5：枝刈りロジックの実装

```
currentStatus = queue.popleft()
```

```
if currentStatus.total >= target:  
    nextFlag = 'N' # 探索中止
```

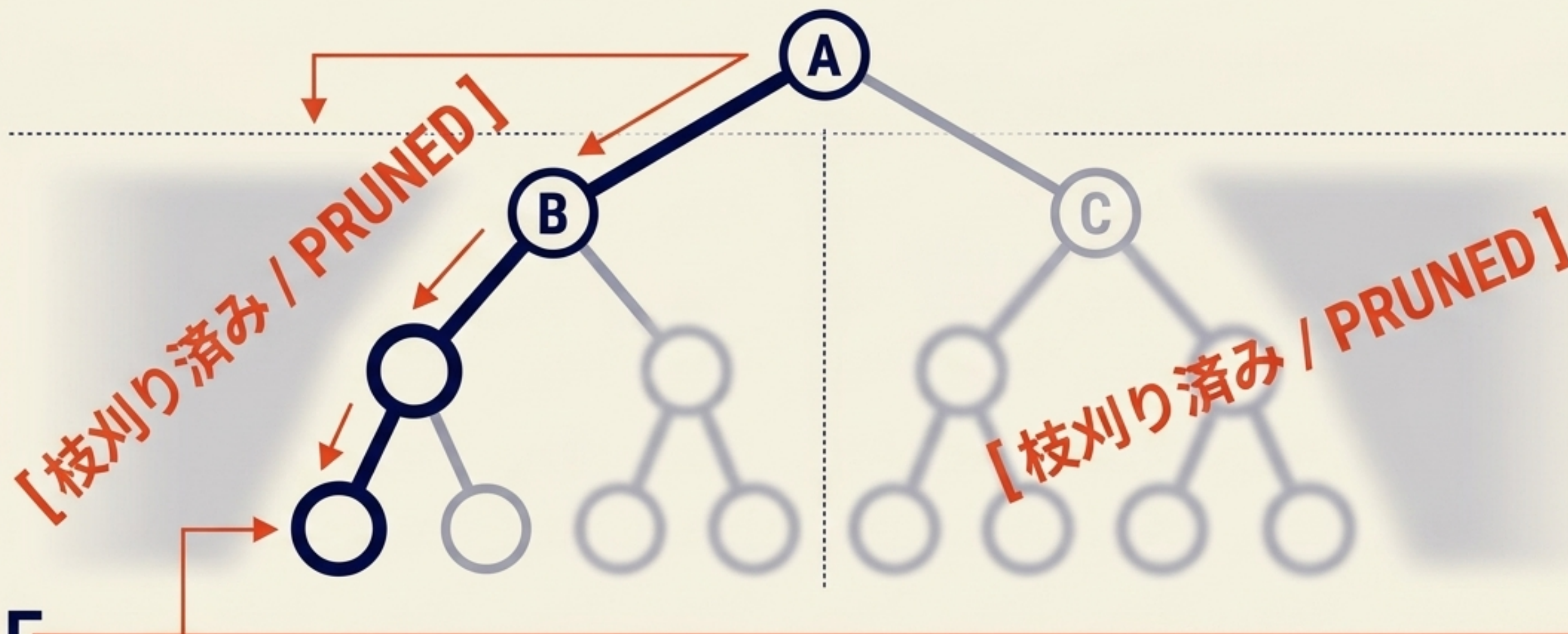
```
else:  
    nextFlag = 'Y' # 探索継続
```

```
if nextFlag == 'Y':  
    queue.append(nextStatus1(currentStatus))  
    queue.append(nextStatus2(currentStatus))
```

```
...(評価ロジック)...
```

ゲートキーパー機能：
フラグが 'N' になった場合、子
ノードの生成とキューへの追
加を完全にスキップする。

[枝刈りエンジンの可視化：無駄な探索空間の消滅]



- 無駄な子ノードの生成を防ぐことで、評価にかかるプロセッサ時間を節約。
- キューに格納される状態数が減少し、メモリ消費のボトルネックを解消。
- 探索空間が極めてシャープな経路に絞り込まれる。

高度な最適化：事前のデータソートがもたらす力

未処理データ (Mixed Data) 【差違原
注記、第2
タソアル、1】

[10]	20	70	80]
------	----	----	-----



ソート処理 (Sort)

降順ソート済み (Descending Data) 【≡≡≡】

[80]	70	20	10]
------	----	----	-----

最適化の戦略 (The Strategy):

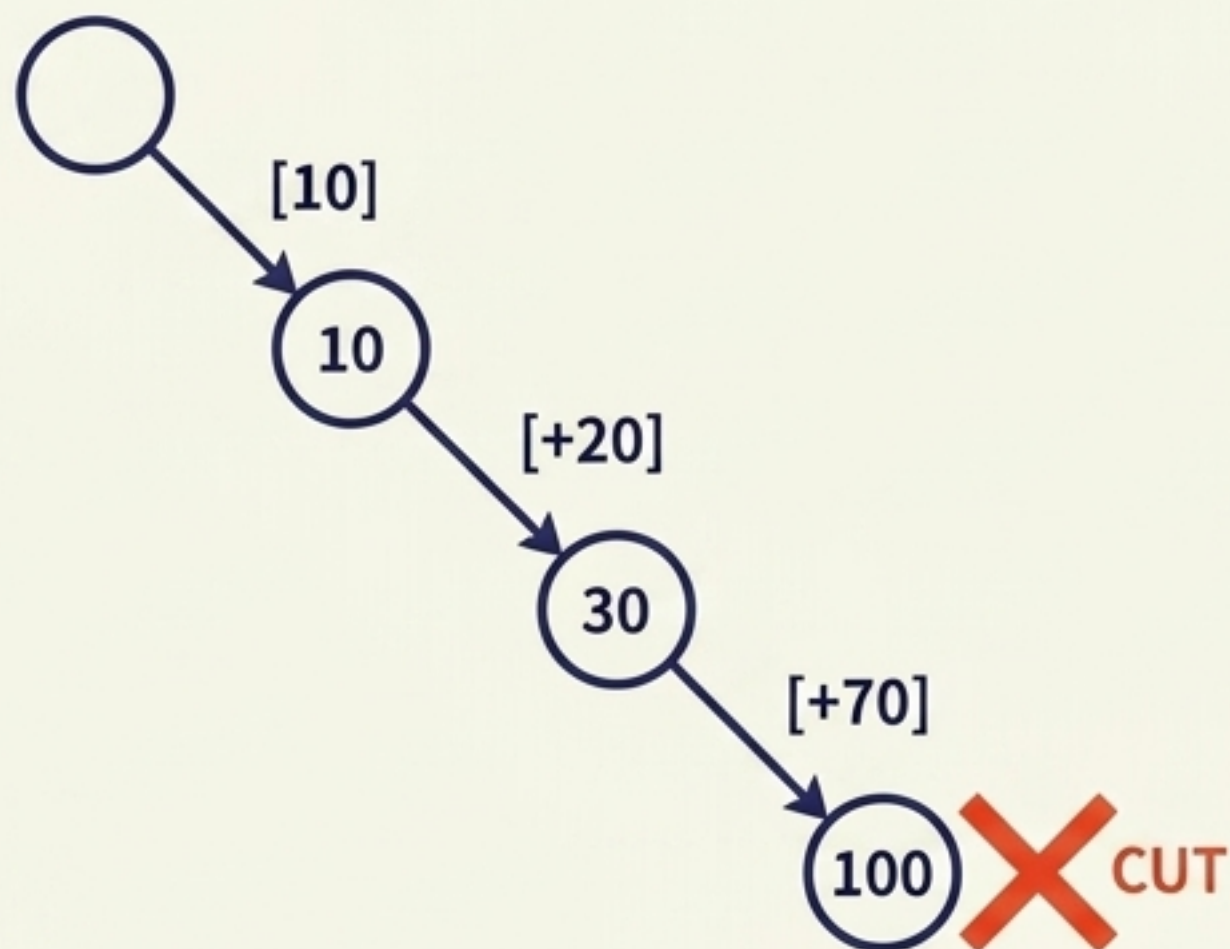
データを「大きい順（降順）」に並べ替えてから探索をスタートする。

目的 (The Goal):

木構造のより浅い階層で枝刈り条件 ($\text{total} \geq \text{target}$) を意図的に早期発動させる。

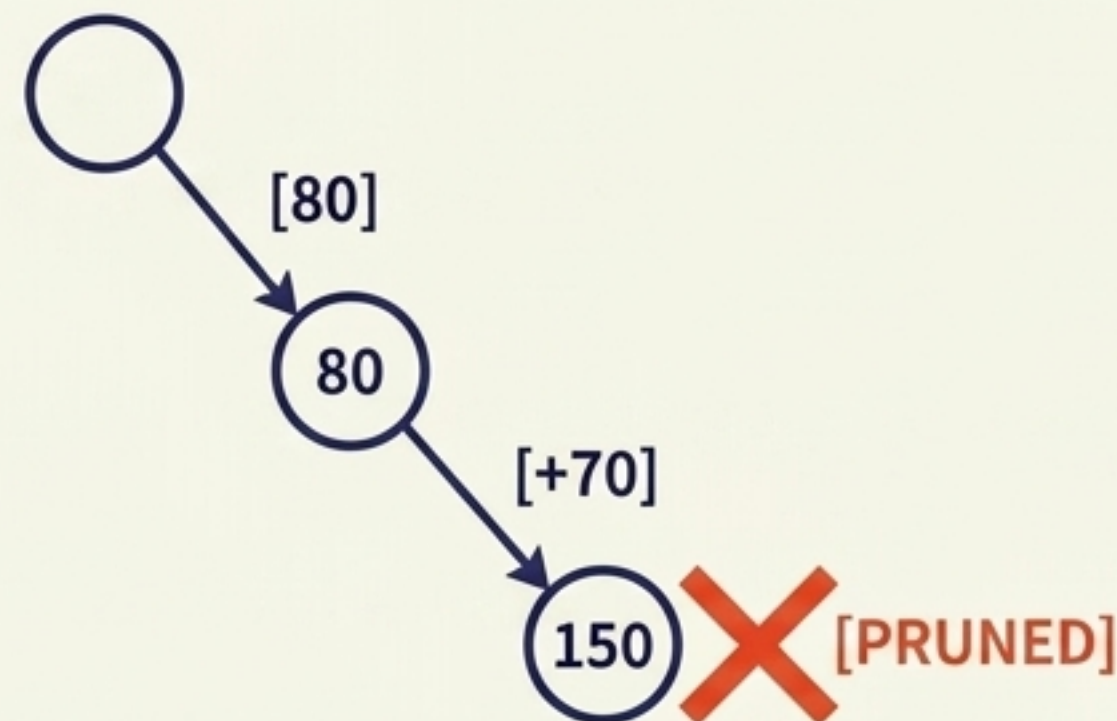
シナリオ比較：昇順 (Ascending) vs 降順 (Descending)

パターンA (昇順：10, 20, 70, 80)



- 目標100を超えるのに 3 レベル 深く潜る必要がある。
- 到達までに多くの無駄な分岐が生成されてしまう。

パターンB (降順：80, 70, 20, 10)



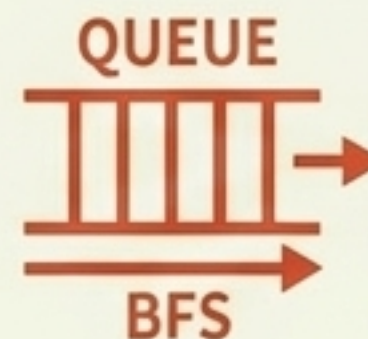
- わずか 2ステップ で目標100を突破し、探索中止フラグが立つ。
- 20と10を含む巨大な部分木を丸ごと一瞬でカットできる。

総括：部分和問題におけるアルゴリズム設計の要点



1. 課題 (The Challenge)

2^n の組合せ爆発によるメモリと計算時間の限界。



2. 基本構造 (Base Engine)

キュー (Queue) を利用した状態の幅優先探索 (BFS)。



3. コード最適化 (Logic Upgrade)

$\text{total} \geq \text{target}$ をトリガーとした枝刈り (Pruning) による探索空間のカット。



4. データ最適化 (Data Prep)

降順 (Descending) ソートによる、枝刈り条件の早期発動と最大効率化。