

全探索から「枝刈り (Pruning)」への進化

- [illegible]

課題の定義：部分和問題とは？

与えられた数の集合から、合計が「目標値（Target）」に最も近くなる組合せを見つける問題。

[10]

[34]

[55]

[77]

$$[10] + [34] + [55] = 99$$

Target: 100

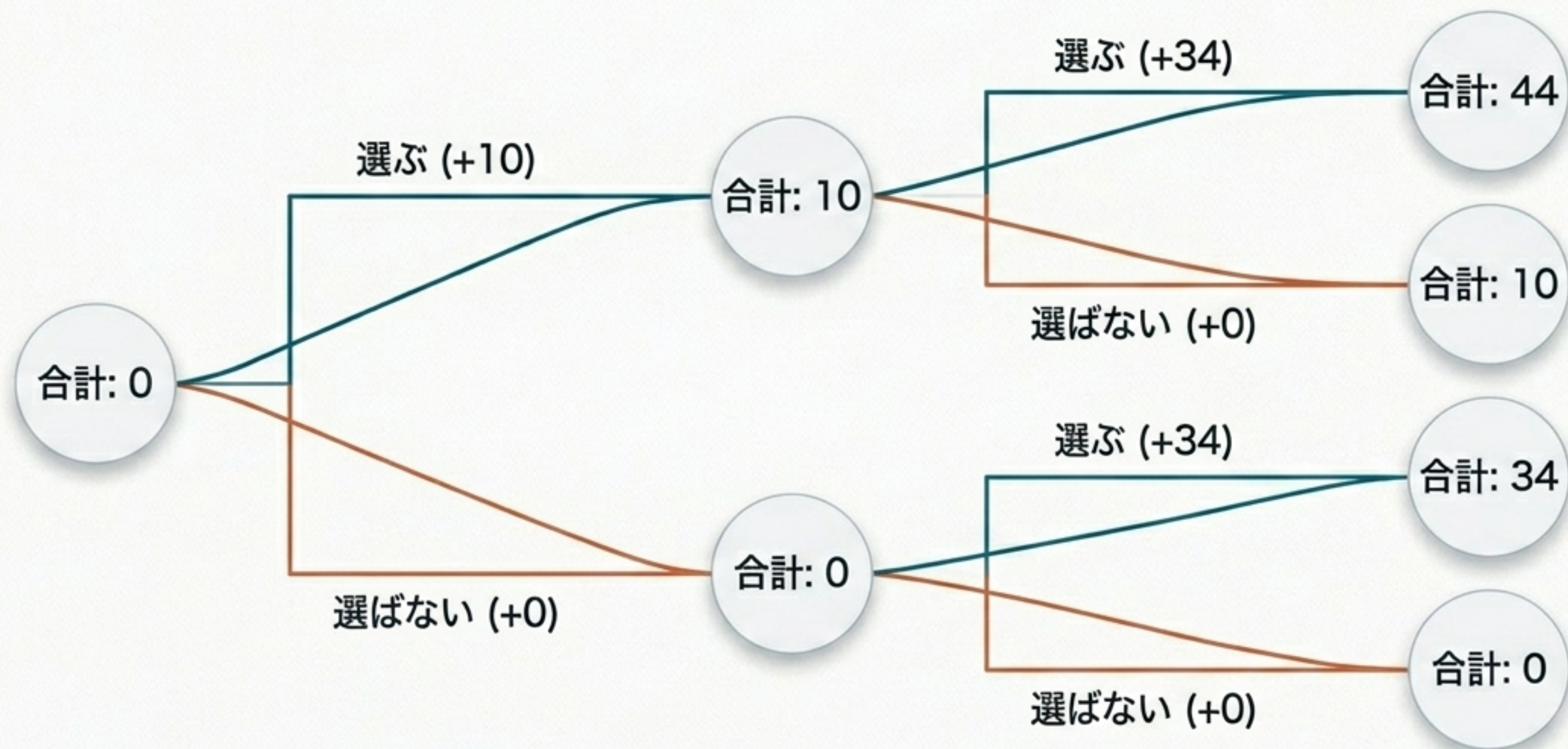
99

制約条件:

1. 各数字は1回しか使えない
2. 全ての組み合わせを網羅的に調べる必要がある
3. 目標値（100）を超過しても、不足してもいけない

アルゴリズムの基本概念：状態空間木 (State Space Tree)

答えを見つけるためのアプローチ。各数字について「選ぶ」か「選ばない」かの2択を繰り返し、木構造 (Tree) を展開して全通りを網羅する全探索 (Brute-force search) の手法。



実装アーキテクチャ（コードの全体像）

ベタ書きのスクリプトではなく、役割ごとに4つの論理コンポーネントに分割して実装。
これにより、後から「枝刈り」などの最適化を組み込みやすい堅牢な設計となる。



コード解剖①：初期データと状態管理（Data & Status）

探索の途中経過を記憶するためのクラス。木構造の「ひとつのノード」が保持すべき全情報（現在の合計、選んだ数、次に見るべき要素）をカプセル化する。

```
numbers = [0, 10, 34, 55, 77]
```

```
n = 4
```

```
target = 100
```

```
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
```

Memory Visualization

Status Object (Node)

total: 44

selectedNumbers: [10, 34]

nextIndex: 3
(次に見る要素: 55)

コード解剖②：状態遷移（Transition Functions）

現在の状態（親ノード）から、次の状態（子ノード）を生成する分岐関数。データを複製しながらインデックスを進め、木を「枝分かれ」させる。

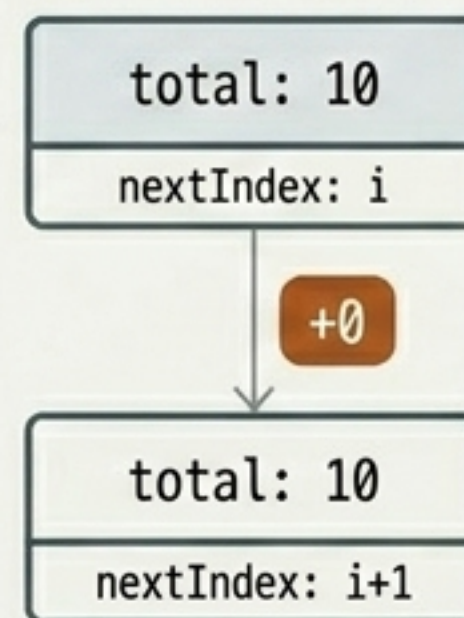
nextStatus1(s) – 選ぶルート

```
def nextStatus1(s):  
    new_total = s.total + numbers[s.nextIndex]  
    new_selected = s.selectedNumbers + [numbers[s.nextIndex]]  
    new_next_index = s.nextIndex + 1 if s.nextIndex < n else 0  
    return Status(new_total, new_selected, new_next_index)
```



nextStatus2(s) – 選ばないルート

```
def nextStatus2(s):  
    new_next_index = s.nextIndex + 1 if s.nextIndex < n else 0  
    return Status(s.total, s.selectedNumbers, new_next_index)
```

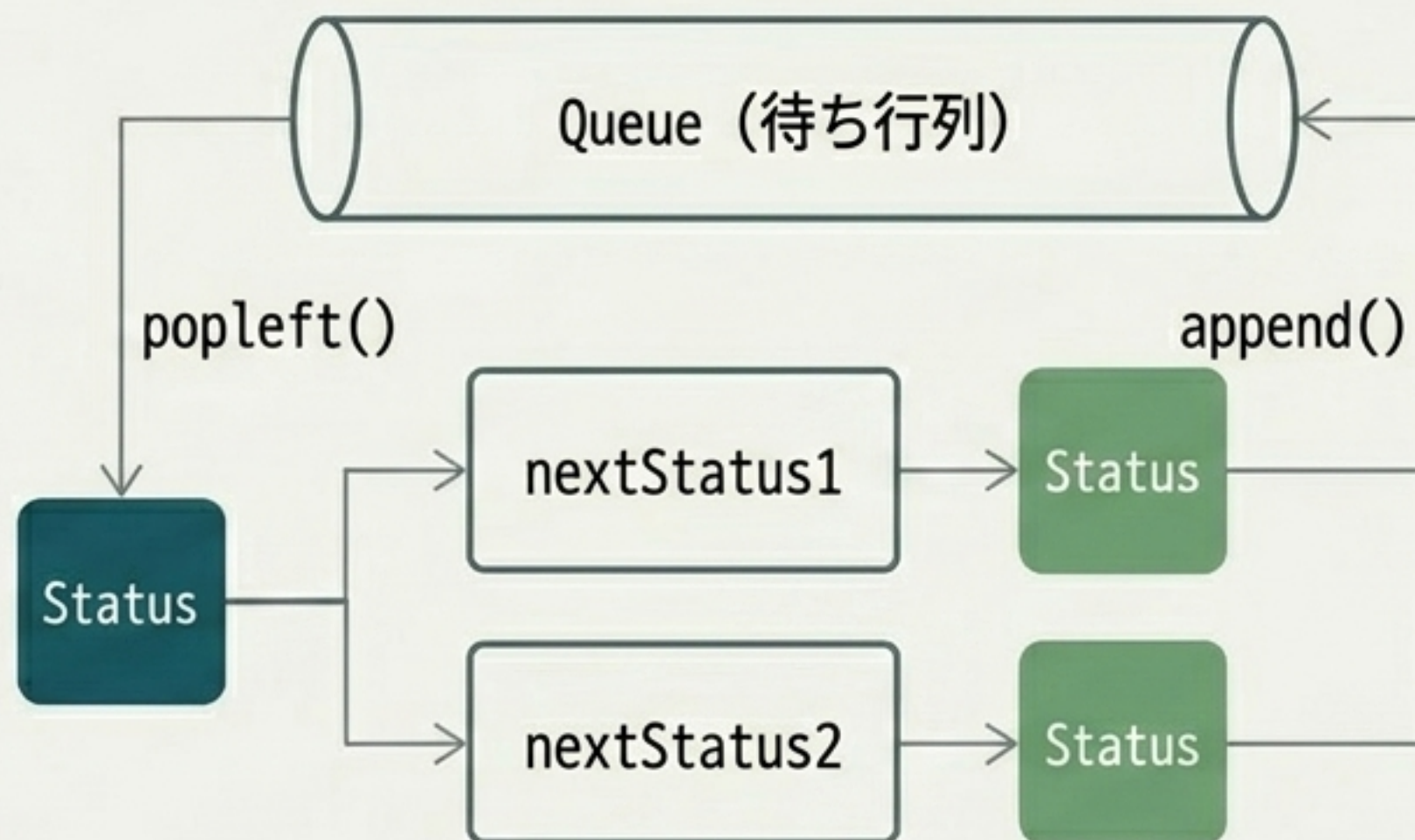


コード解剖③：探索処理（Tree Search Execution）

キュー（Queue）を用いた幅優先探索（BFS）の実装。

1. キューから状態を取り出す
2. 遷移関数で次状態（選ぶ・選ばない）を2つ作る
3. 新しい状態をキューの末尾に入れる
これをキューが空になるまで繰り返す。

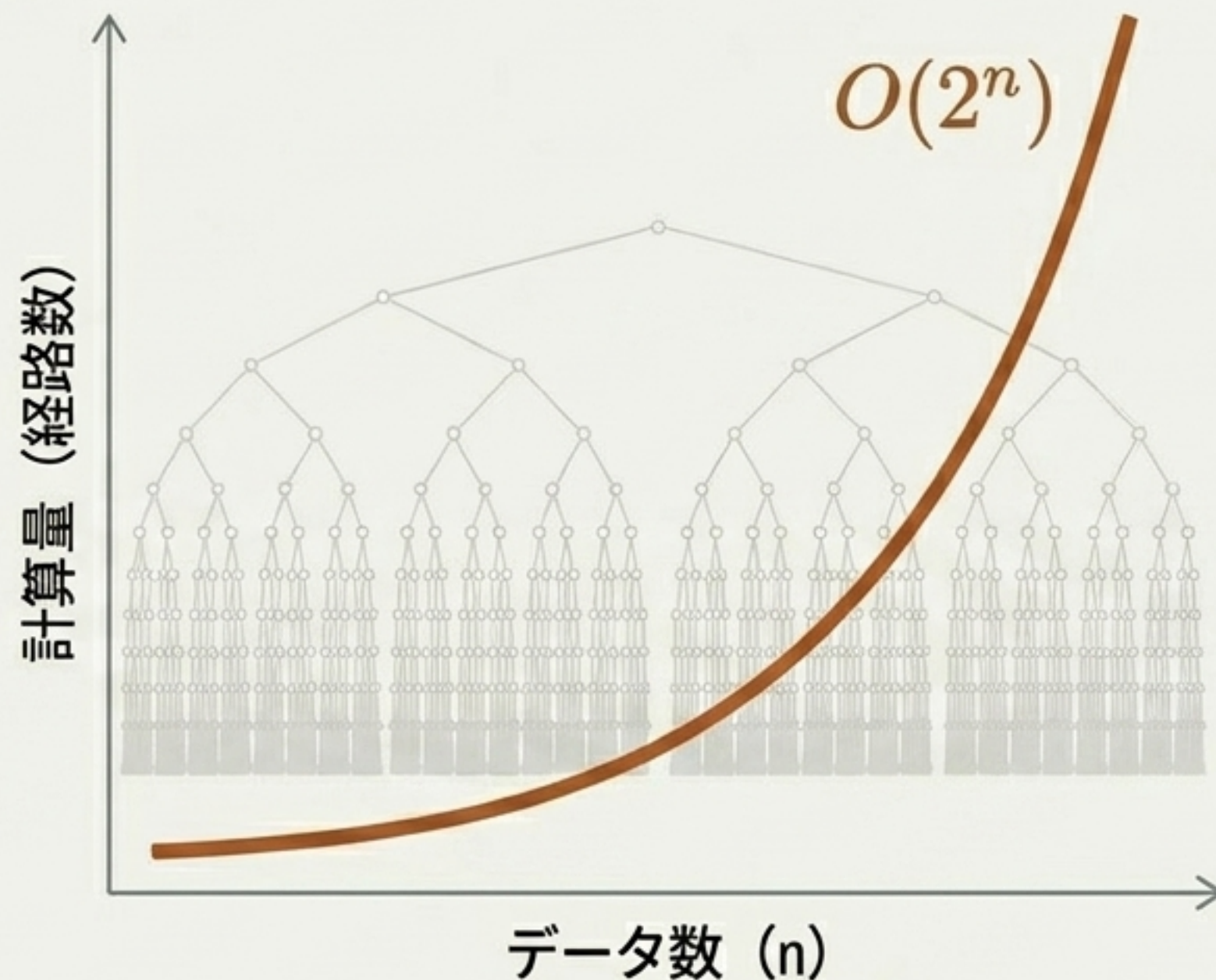
```
def treeSearch():  
    ansStatus = None  
    queue = collections.deque()  
    queue.append(Status())  
  
    while queue:  
        s = queue.popleft()  
  
        # 次の状態を生成してキューに追加  
        queue.append(nextStatus1(s))  
        queue.append(nextStatus2(s))
```



課題提起：全探索のボトルネック

愚直に全パターンを展開するアプローチの限界。無駄な探索ルート（明らかに目標から遠ざかるもの）も律儀に計算してしまうのが最大の弱点。

- データ数 $(n) = 4$ の場合：
 $2^4 = 16$ 経路（瞬時に完了）
- データ数 $(n) = 40$ の場合：
 $2^{40} \approx 1$ 兆 経路超
（処理が終わらない）

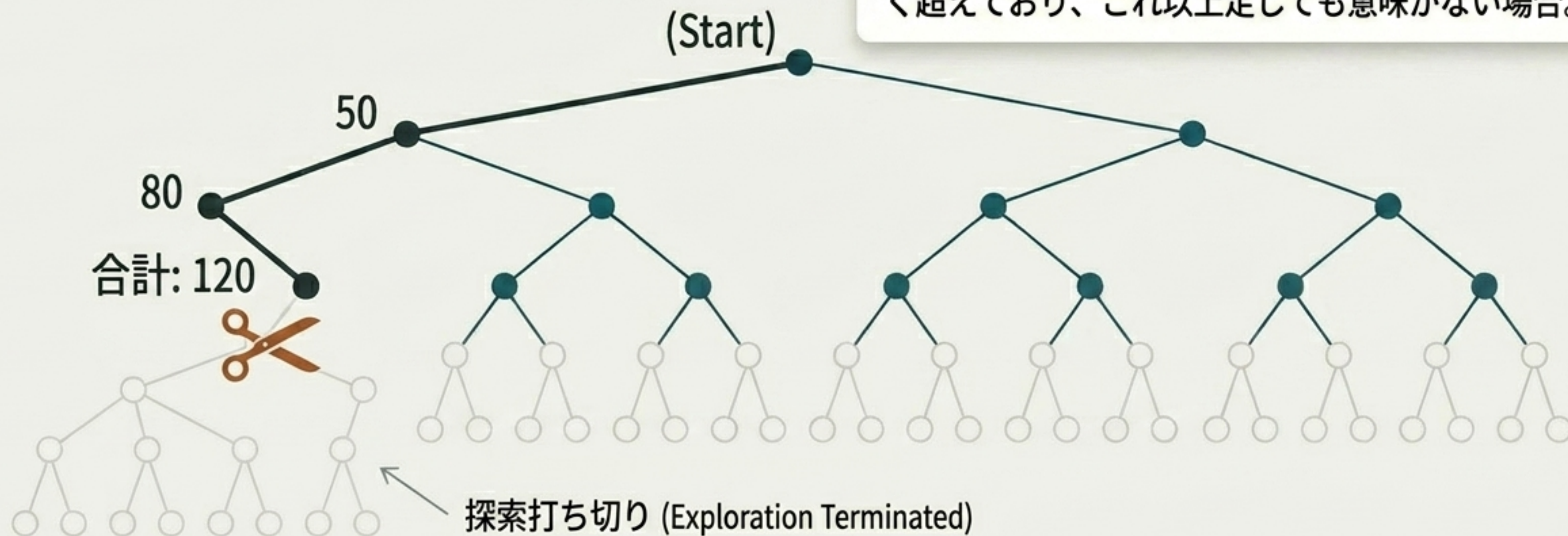


解決策：「枝刈り (Pruning)」の概念

「これ以上探索しても正解にたどり着かない」と判明した時点で、その先の分岐（木構造の展開）を打ち切るアルゴリズム設計手法。

枝刈りの条件例：

現在の合計値がすでに目標値 (Target=100) を大きく超えており、これ以上足しても意味がない場合。



コード解剖④：枝刈りの実装（treeSearchWithPruning）

キューから状態を取り出した直後に「条件判定」を追加するだけで、枝刈りは成立する。無駄な状態が生成されてキューに入るのを防ぎ、メモリ消費と計算時間を劇的に削減。

```
def treeSearchWithPruning():  
    ansStatus = None  
    queue = collections.deque()  
    queue.append(Status())  
  
    while queue:  
        s = queue.popleft()  
        if s.total > target:  
            continue # 枝刈り：目標値超過ならこの先の分岐を作らない  
  
        # 次の状態を生成してキューに追加  
        queue.append(nextStatus1(s))  
        queue.append(nextStatus2(s))
```



比較マトリックス：基本実装 vs 枝刈り実装

枝刈りは、アルゴリズムの「正確性」を一切損なうことなく、「リソース消費」だけを劇的に抑えるスマートな設計技法。

評価軸	基本実装 (treeSearch)	枝刈り実装 (treeSearchWithPruning)
探索アプローチ	網羅的（全探索）	選択的（最適化）
評価するノード数	最大（ $O(2^n)$ ）	大幅に削減される
計算時間	指数関数的に増大	条件次第で劇的に高速化
メモリ使用量	膨大（キューが巨大化）	最小限に抑えられる

まとめ (Synthesis & Insights)

アルゴリズム設計の王道プロセス：



1. データ・状態・遷移 の分割

複雑な処理も「今の状態」と
「次への変化」のクラス／関
数にモジュール化する。



2. 状態空間の 木構造化

「選ぶ」「選ばない」の分岐
として、問題を視覚的にマッ
ピングする。



3. 枝刈りによる 最適化

愚直な全探索を実装したの
ち、論理的な条件を用いて無
駄な探索ルートを断ち切る。