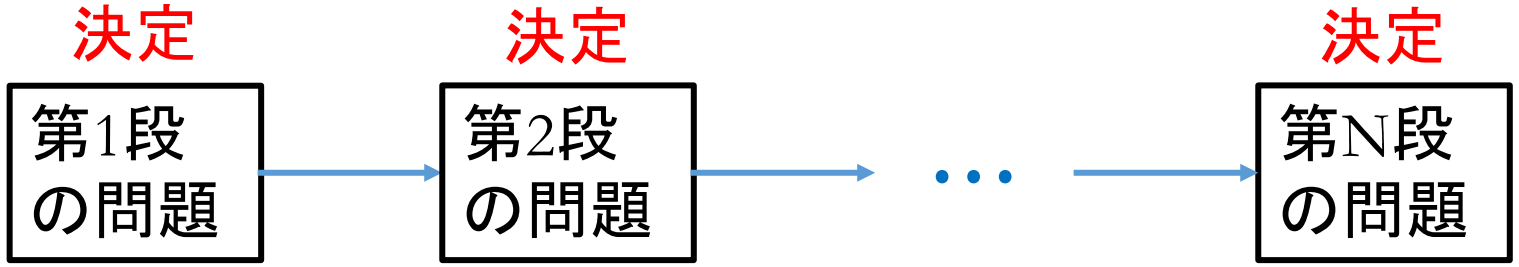
動的計画法（DP）

考え方・典型的なDP例

動的計画法（DP: Dynamic programming）

- 動的計画法（以降DPと略す）とは、与えられた問題を部分問題に分割し、部分問題の結果を記憶・利用しながら解いていく手法の総称
- 動的計画法は多段決定問題（multi-stage decision problem）としてとらえることもできる。つまり、N段階決定過程をN個の１段階決定過程の列に直すことにより、多段決定問題を逐次的に解く方法



DPで解く例

- フィボナッチ数列の計算（DP・メモ化再帰の考え方）
- 最大和問題（DPのやり方に慣れるため）
- 部分和問題・ナップサック
- グラフの最短経路問題（応用問題）
- 日本語形態素解析（応用問題）

フィボナッチ数列

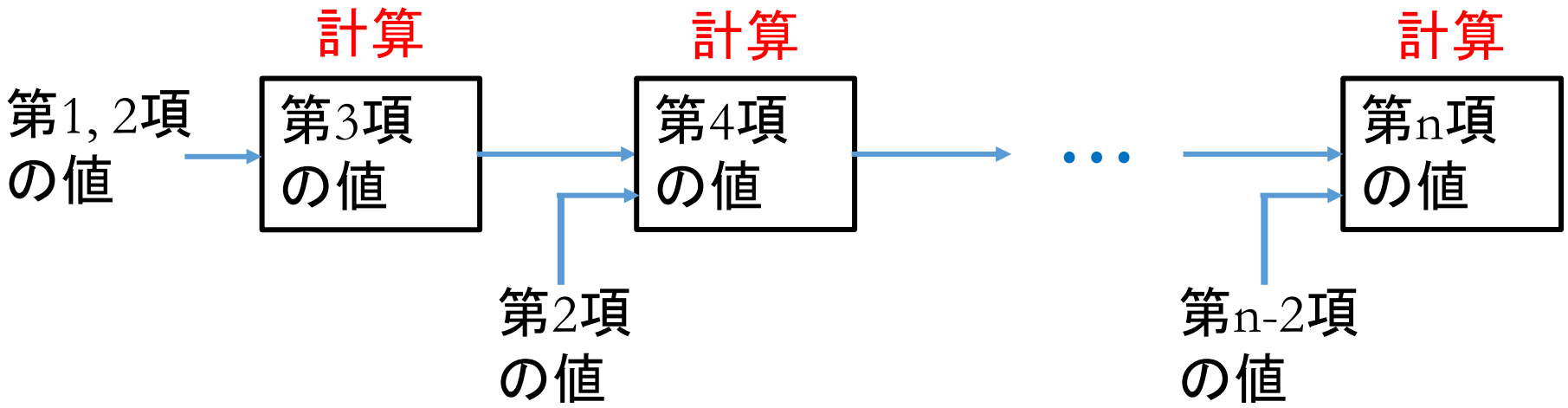
DP・メモ化再帰の考え方

フィボナッチ数列とは

- 前2つの数を足していくような並びの数列
  - 1, 1, 2, 3, 5, 8, 13, 21, ...
- 自然界に多く見られる数列
  - 木の枝分かれの様子
  - 花びらの数
  - 黄金比
    - 5/3=1.666, 8/5=1.6, 13/8=1.625, 21/13=1.615...
- 自然界以外の一般社会でも利用されている
  - 株式の相場予測
  - 人口変動モデル
  - 芸術作品
  - プログラミング・アルゴリズム

フィボナッチ数列の計算

- 第n項を決定（計算）するためにはまず第n-1, 第n-2項を決定、第n-1項を決定するためには第n-2, 第n-3を決定…
- つまり、多段決定問題



アルゴリズム

- 第n項の計算。n ≥ 3とする

a: 数列第1項から第n項を格納するための1次元整数配列

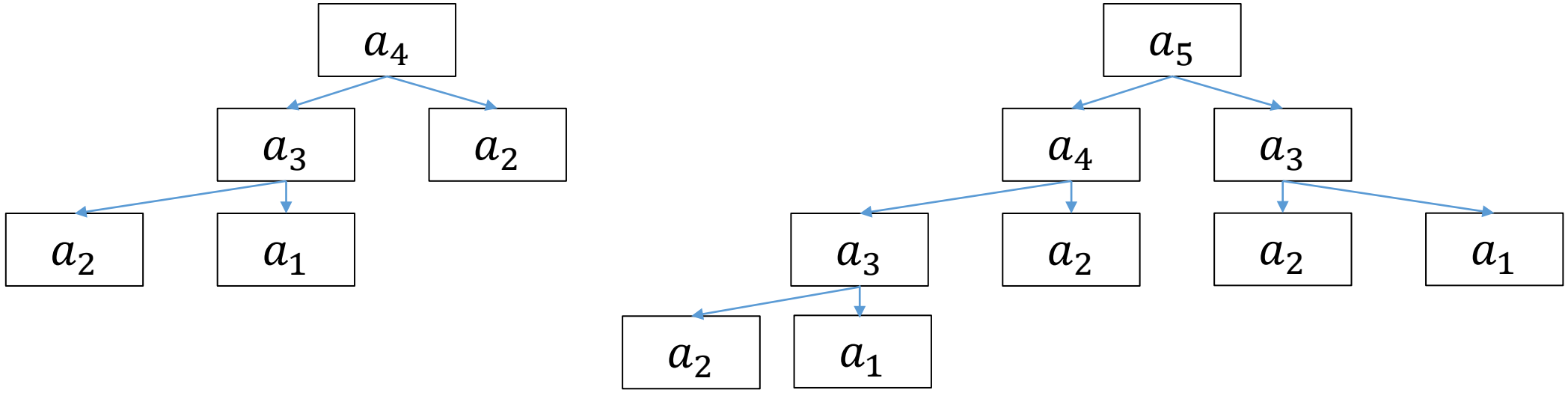
i : 繰り返し変数

1. 初期設定としてa[1]=1, a[2]=1とする
2. iを3からnまで以下の処理を繰り返す

    2.1 a[i]=a[i-1]+a[i-2]

再帰でもできる

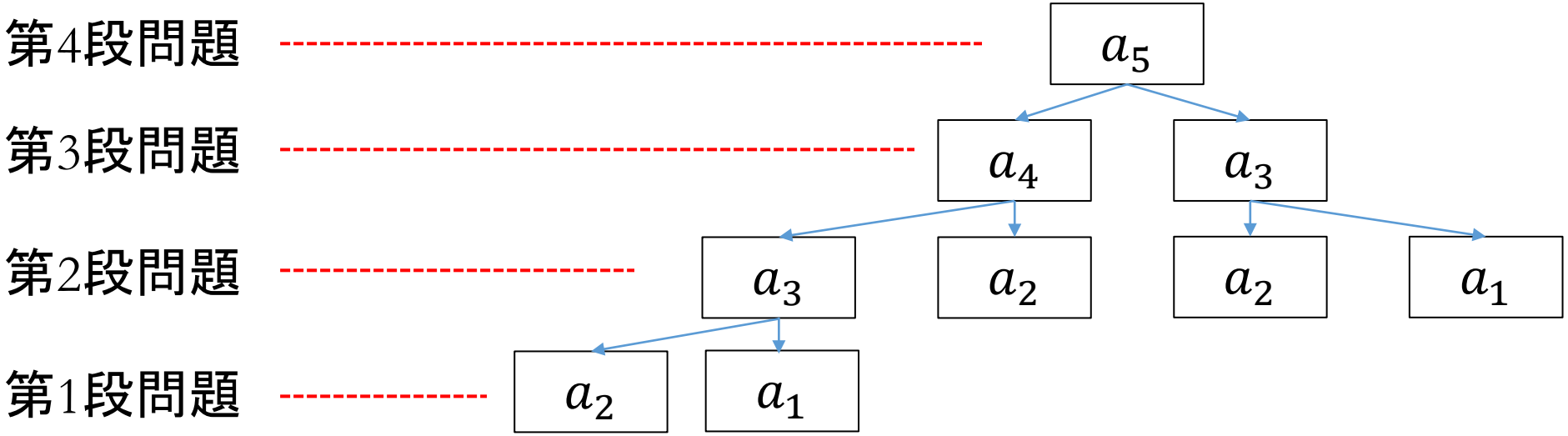
- 漸化式
  - $a_1 = 1, a_2 = 1, a_n = a_{n-1} + a_{n-2}\ (n = 3, \dots)$
- n項の計算は部分問題に分割して解ける



- n項の計算は深さ優先的に行うことができる…再帰

再帰でもできる

- 「多段決定問題」である



再帰プログラム

```
#include <stdio.h>

int n=50;

//answer 12586269025 overflows when
//using int or unsigned int

long int a(int i)
{
  if(i<=2) return 1;
  return a(i-1)+a(i-2);
}
```

```
int main(void)
{
  printf("%ld¥n", a(n));
  return 0;
}
```

演習：再帰の展開

- n=4とした場合、上記プログラムの実行結果が3であることについて、（これまでやってきた）再帰の展開で自分自身を納得させてください
- ヒント：
  - return a(i-1)+a(i-2)はa(i-1)を処理し、a(i-2)を処理し、その両方の結果を足してreturnする、という意味

何が問題？

- 上記プログラムを作成し、以下のように実行してみよう

  time ./a.out

- 結果が出るまで何分かかった？

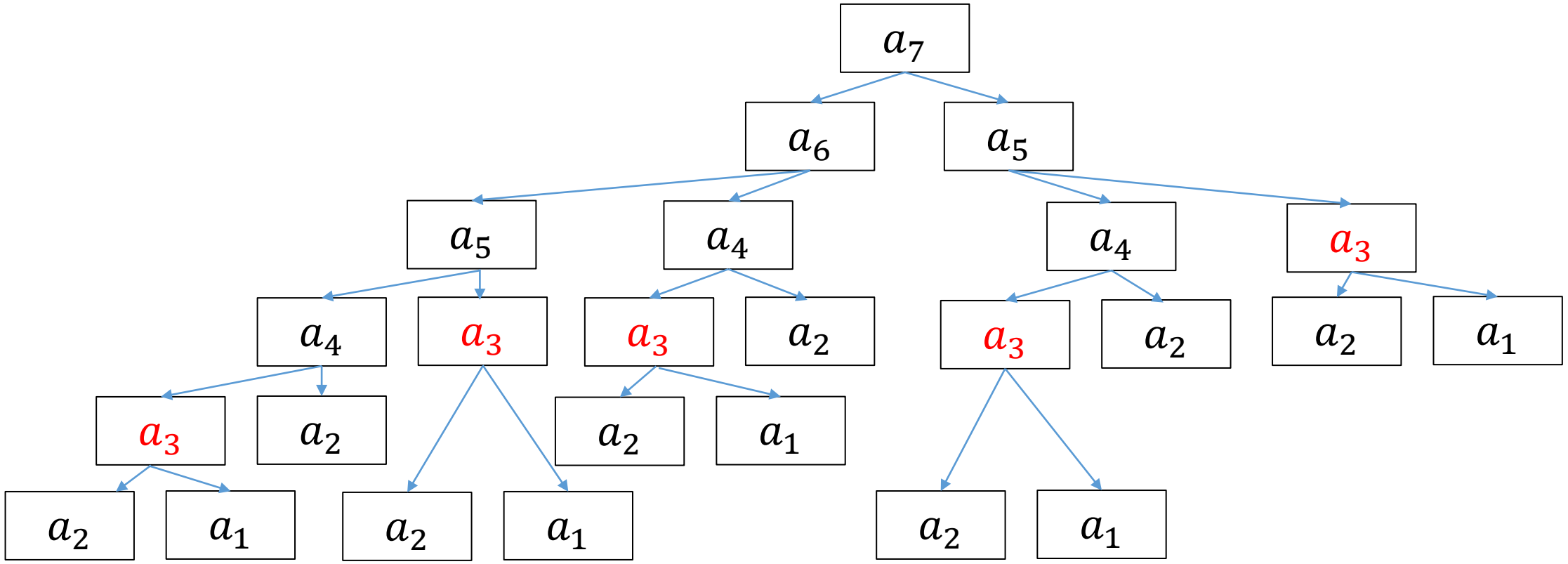
何が問題？

- n=50の項の計算は41秒も！

```
time ./a.out
12586269025
real    0m40.988s
user    0m40.987s
sys     0m0.000s
```

- 原因を調べると
  - n=50: a(3)を5億回以上も！呼び出されて実行されている

なぜ同じ関数が重複して呼び出されるか？



メモ化・メモ化再帰

- 途中結果を「メモ」するようにすれば問題解決
- メモ化（Memoization）
  - プログラム高速化のための最適化技法の一種。関数呼び出しの結果を記憶し、同じ入力があった場合記憶されている結果を返す
- 再帰にメモ化を用いることをメモ化再帰（Memoized recursion）
- メモ化再帰であれば必ずDPである
- DPは必ずしも再帰で実現しないといけないというわけではない

メモ化再帰のプログラム

```
#include <stdio.h>

int n=50;
long int memo[50]; // 0で初期化される

long int a(int i)
{
  if(i<=2) return 1;
  if(memo[i] != 0) return memo[i];
  memo[i]=a(i-1)+a(i-2);
  return memo[i];
}
```

```
int main(void)
{
  printf("%ld¥n", a(n));
  return 0;
}
```

どう変わった？

- 上記プログラムを作成し、以下のように実行してみよう

  time ./a.out

- 結果が出るまで何分かかった？

どう変わったか？

- n=50の項の計算はわずか0.002秒！

```
time ./a.out
12586269025
real    0m0.002s
user    0m0.002s
sys     0m0.000s
```

- 原因を調べると
  - n=50の場合
    - a(3)...a(48)まではそれぞれ呼び出し2回実行1回！
    - a(49)は呼び出し1回実行1回！
  - なぜかを考えてみよう

最大和問題

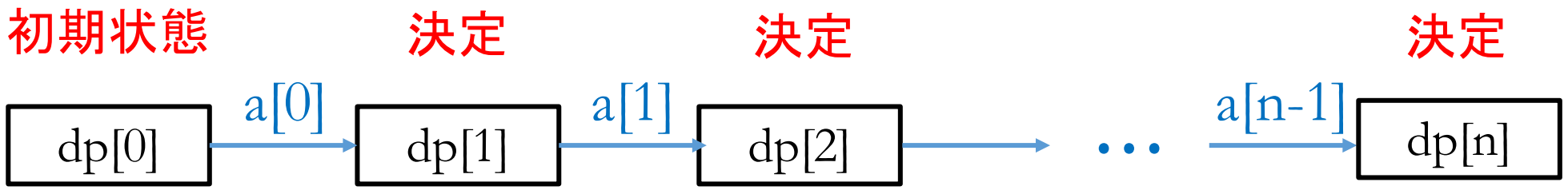
DPのやり方に慣れよう！

最大和問題とは

- n個の整数が格納されている配列aが与えられる。これらの整数から何個かの整数を選んで総和をとったときの、総和の最大値を求める問題。ただし、何も選ばない場合の総和は0とする
- 例
  - n=5, a[]={2, -6, 3, 1, -9, 4}　　答え：10（2, 3, 1, 4を選べばよい）
  - n=2, a[]={-9, -16}　　答え：0（何も選ばない）
- DPの考え方・解き方に慣れるための例。「正の値すべてを選ぶ」という方法で解けるので、DPを使う必要はまったくない

解き方

- 多段決定問題。各段の決定した最大値を整数配列dpで記憶
- dp[n]が結果



何も選んでいない状態dp[0]=0

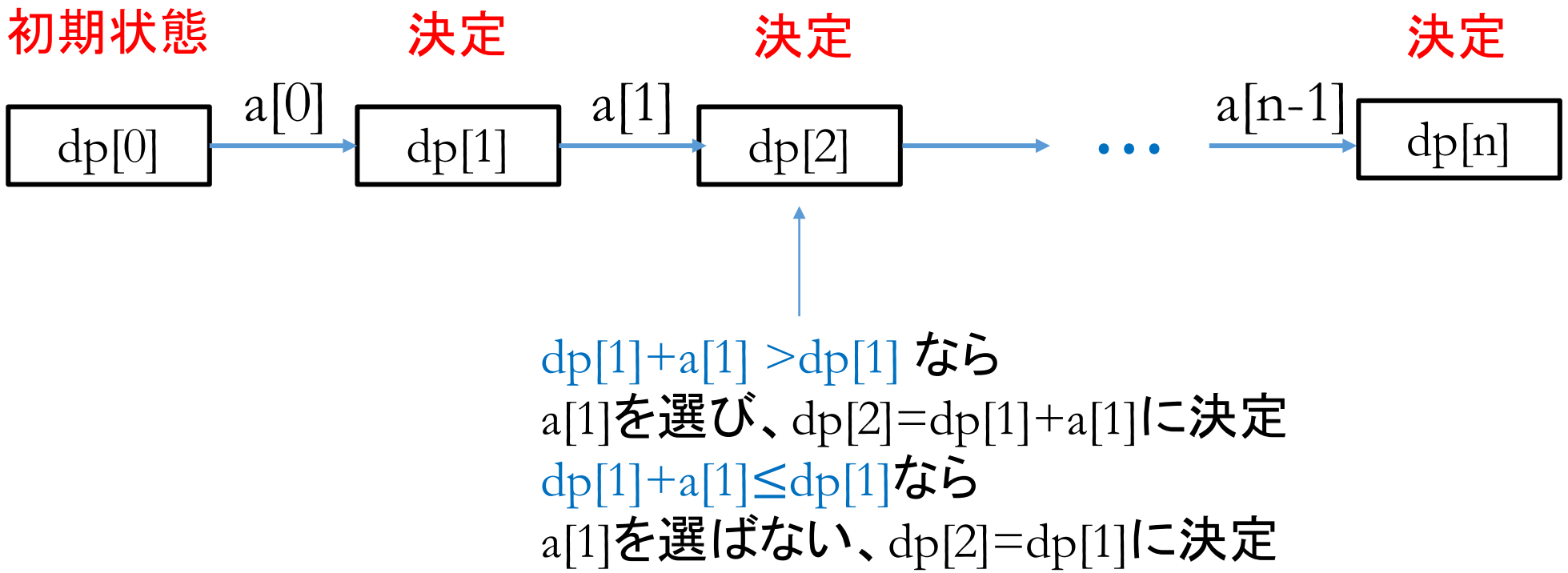
a[0]を選ぶか選ばないかをdp[1]が最大となるように決定し、dp[1]を決定する

a[1]を選ぶか選ばないかをdp[2]が最大となるように決定し、dp[2]を決定する

a[n-1]を選ぶか選ばないかをdp[n]が最大となるように決定し、dp[n]を決定する

解き方

- ここで重要なのは、dp[i+1]を決定するとき、dp[i]を使うこと。つまり
- dp[i+1]はi番目までのすべての整数（dp[i]にa[i]）によって決まる最大値



アルゴリズム

入力：データの数n, 1次元整数配列：a
出力：最大値
補助：1次元整数配列dp: 各段の最大値を格納
　　i: 段番号

1. 初期設定dp[0]=0
2. iを0からn-1まで以下の処理を繰り返す
　2.1 dp[i]+a[i]の値とdp[i]の値とを比較し、大きい方をdp[i+1]に

dp[n]が最大値となる

プログラム

```
#include <stdio.h>

#define N 101

int max(int x, int y)
{
  if(x<y) x=y;
  return x;
}
```

```
int main(void)
{
  int dp[N], n=10, a[]={1, -2, 3, -5, 4, -6, 6, 7, -10, 8};

  dp[0]=0;
  for(int i=0; i<n; i++){
    ?1=max(dp[i], ?2);
  }
  printf("maximum sum: %d¥n", dp[n]);

  return 0;
}
```

manaba小テスト：05-1

- 8分
- 6点

部分和問題

本格的なDP問題

部分和問題とは

- n個の正の整数が格納されている配列aと正の整数Aが与えられる。これらの整数を選んで総和がAになるようにすることが可能かを判定する問題。ナップサック問題によく似た有名な問題

- 例
  - n=5 a[]={7, 5, 9, 15, 11}
  - A=19　答え：No
  - A=27　答え：Yes　(7, 9, 11)

dpの定義

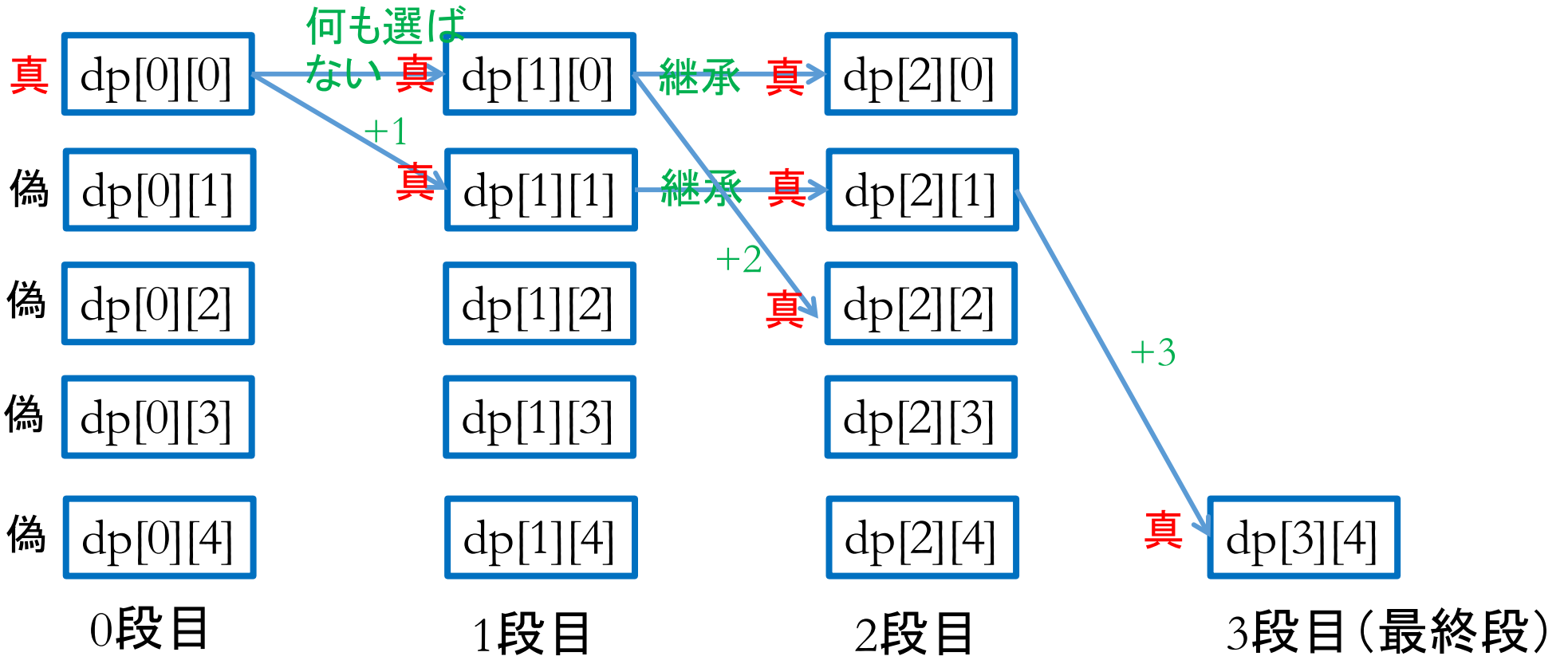
- 各段の決定値を格納する2次元整数配列dpを用意し、以下のように定義する
- dp[i+1][j]: 第i+1段において、(i番目までの整数から)選んだ整数の総和がjとなることが可能か不可能か(論理値:真/偽:1/0)
- i=0, 1, ..., n-1　　j=0, 1, ..., A
- 何も選んでいないときdp[0][0]=1(真) dp[0][j]=0 (j ≠ 0, 偽)
- dp[n][A]の値(1/0, 真/偽)が解となる

例

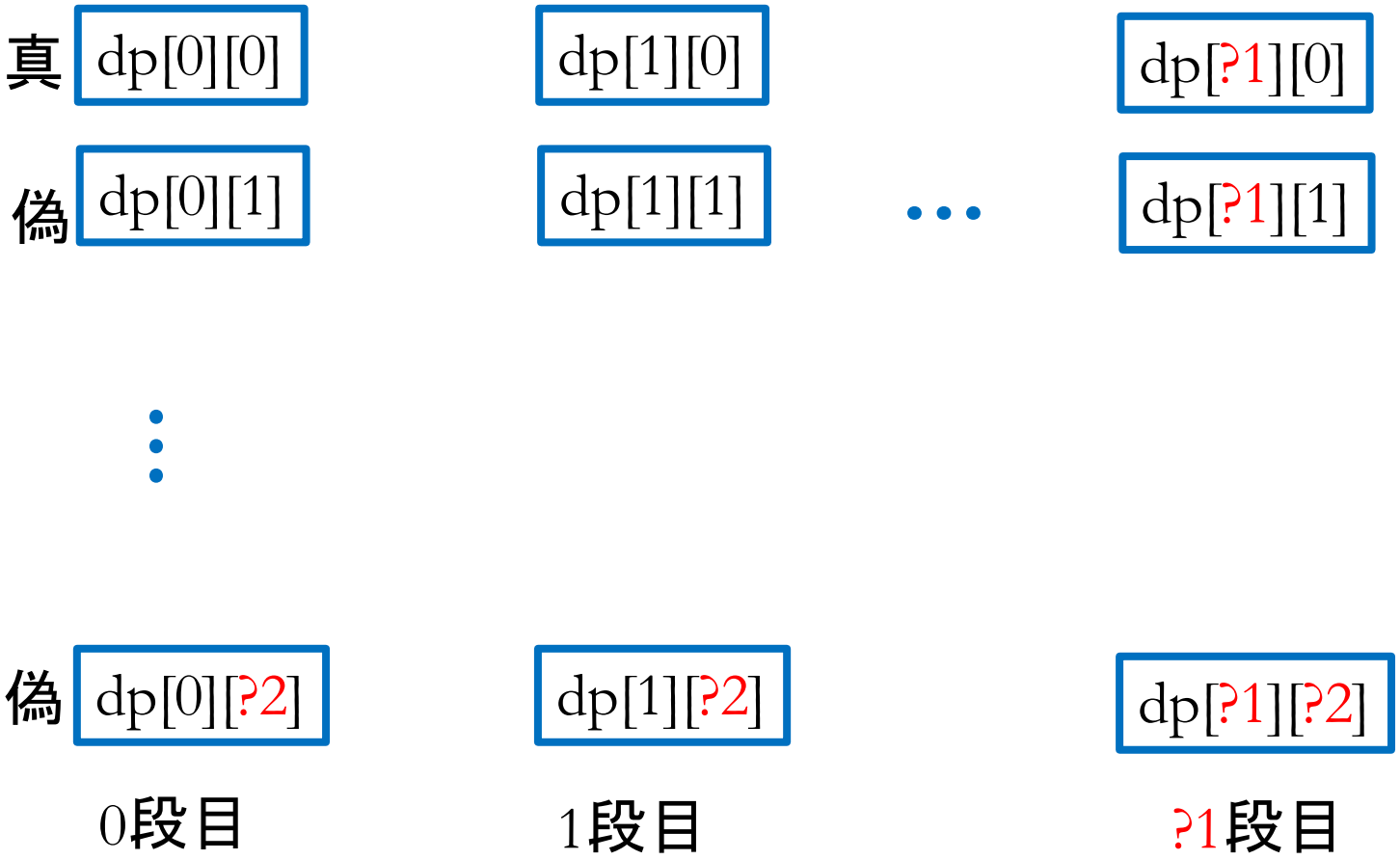
- a={1, 2, 3}, A=4
- n=3, dp[3][4]の論理値を求めればよい

例

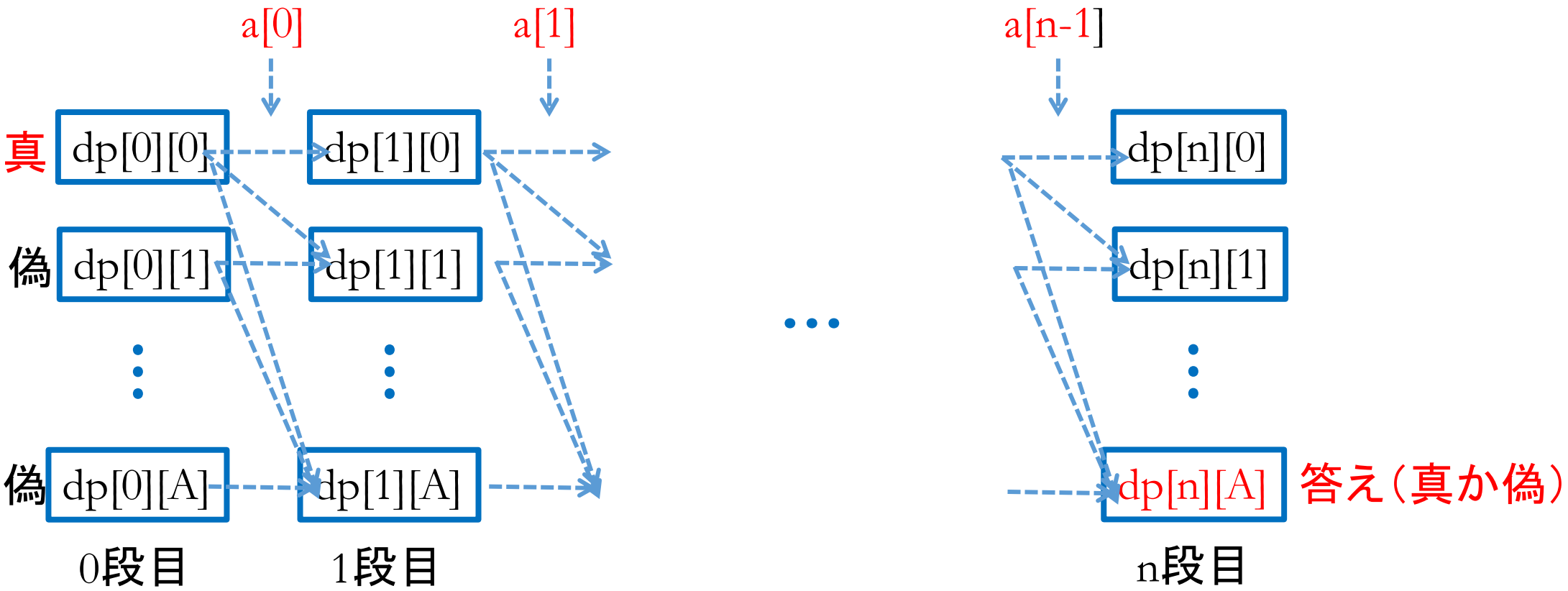
- a={1, 2, 3}, A=4
- n=3, dp[3][4]の論理値を求めればよい



manaba小テスト：05-2（5分，4点）

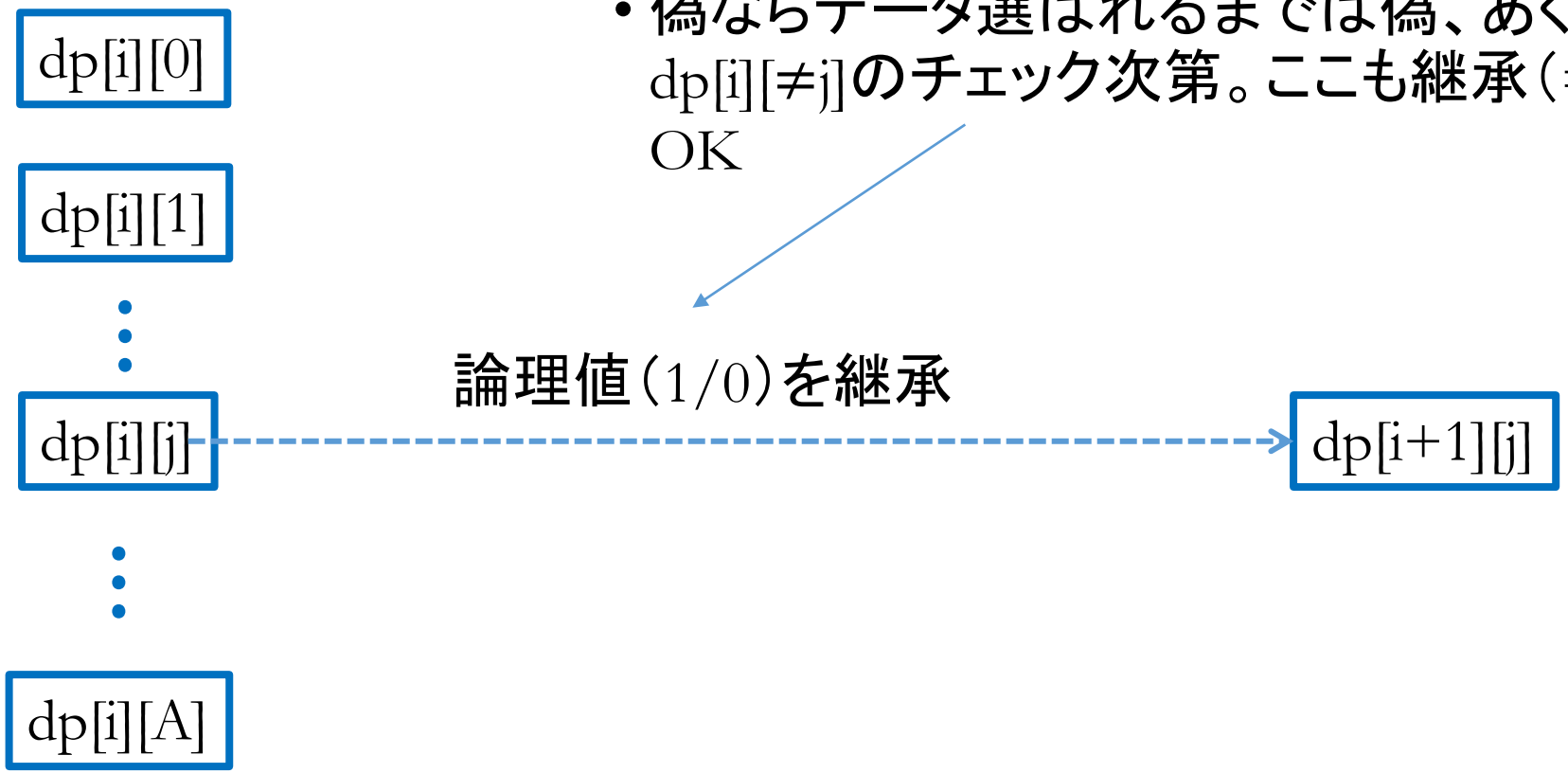


全段

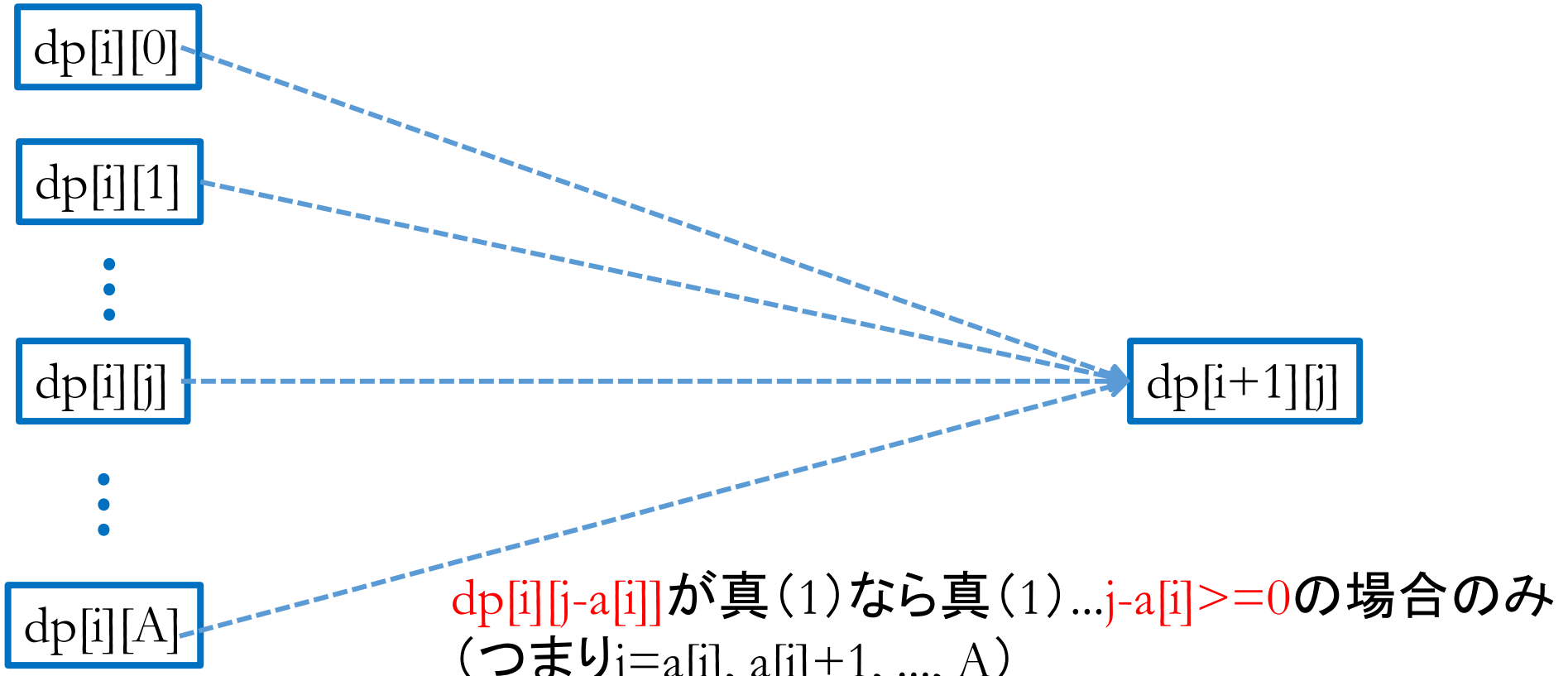

a[0]
a[1]
a[n-1]
真
dp[0][0]
dp[1][0]
dp[n][0]
偽
dp[0][1]
dp[1][1]
dp[n][1]
偽
dp[0][A]
dp[1][A]
dp[n][A]
答え（真か偽）
0段目
1段目
n段目

前後段の関係

- 真ならどんな場合でも真。継承でOK
- 偽ならデータ選ばれるまでは偽、あくまでもdp[i][≠j]のチェック次第。ここも継承（=偽）でOK

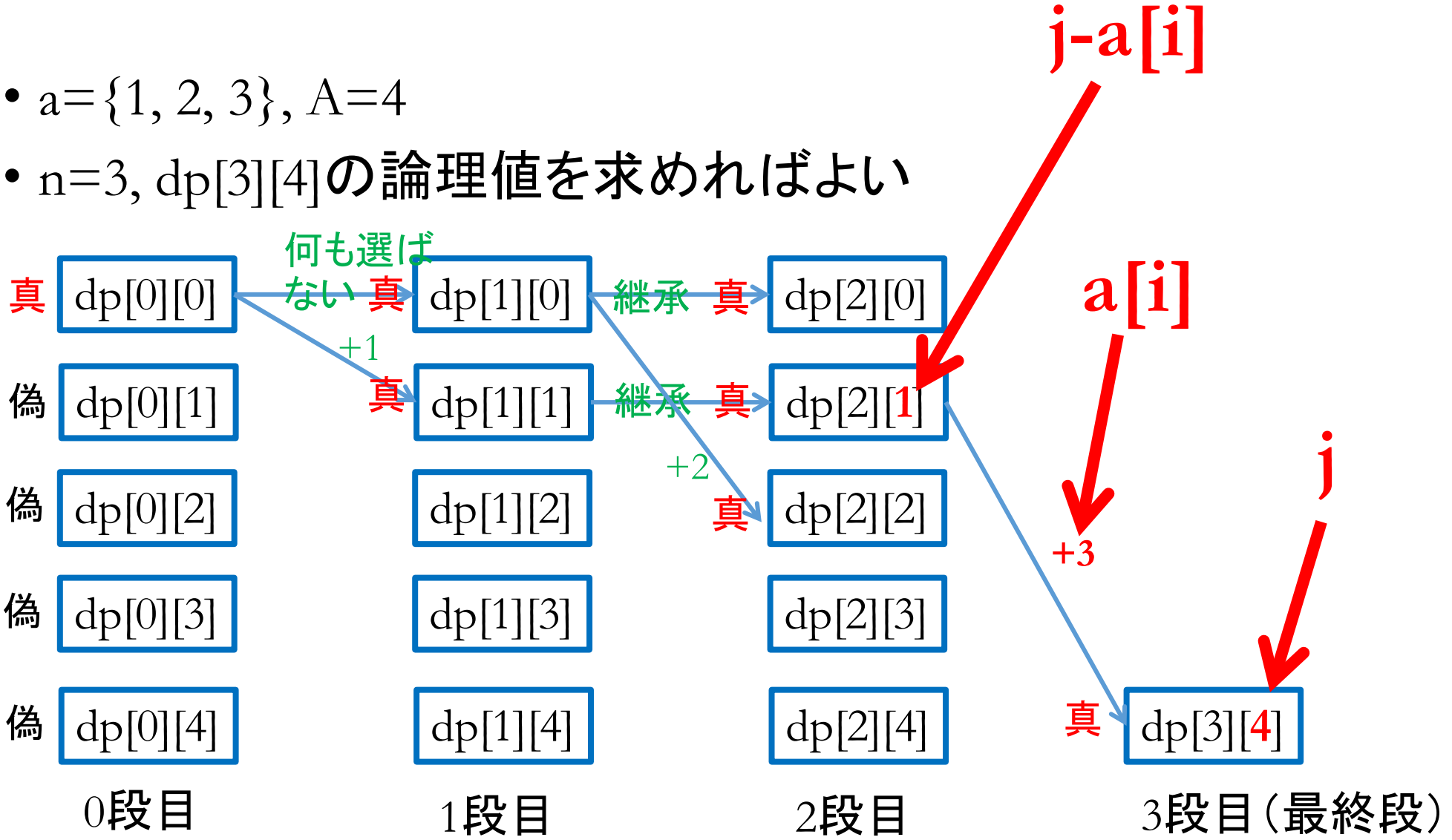


前後段の関係


dp[i][0]
dp[i][1]
dp[i][j]
dp[i][A]
dp[i+1][j]
dp[i][j-a[i]]が真(1)なら真(1)...j-a[i]>=0の場合のみ
(つまりj=a[i], a[i]+1, ..., A)

例（再掲）

- a={1, 2, 3}, A=4
- n=3, dp[3][4]の論理値を求めればよい



アルゴリズム

入力：データの数n, 1次元正の整数配列a, 答え用正の整数A
出力：論理値1, 0
補助：2次元整数配列dp: 各段の答えが0,1,...,Aの場合の論理値(偽：0, 真：1)
　　i: 段番号, j: 答え0, 1, ..., A

1. 初期設定dp[i][j]=0 (i=0,...,n) dp[0][0]=1
2. iを0からn-1まで, jを0からAまで以下の処理を繰り返す
   2.1 dp[i+1][j]はdp[i][j]から論理値継承
   2.2 j>=a[i]の場合 dp[i][j-a[i]]が真であれば dp[i+1][j]も真とする

dp[n][A]の論理値が答えとなる

実行結果：a[]={7, 5, 9, 15, 11}

```
A: 27
i:1     j: 7
i:2     j: 5 7 12
i:3     j: 5 7 9 12 14 16 21
i:4     j: 5 7 9 12 14 15 16 20 21 22 24 27
i:5     j: 5 7 9 11 12 14 15 16 18 20 21 22 23 24 25 26 27
selected numbers
11 9 7
Yes
```

```
A: 17
i:1     j: 7
i:2     j: 5 7 12
i:3     j: 5 7 9 12 14 16
i:4     j: 5 7 9 12 14 15 16
i:5     j: 5 7 9 11 12 14 15 16
selected numbers


No
```