問題解法における動的計画法の研究

－アルゴリズム実技検定の問題解法を通して－

瓜生　隆弘

A Study of a Solution to a Problem Using Dynamic Programming

- From Practical Algorithm Skill Test –

Takahiro Uryu

Abstract

In this paper,I verified the usefulness of Dynamic Programming and other techniques in problem solving. As an example,I took up the question O of the PAST test conducted in December 2019.It has been found that applying Dynamic Programming is useful when it is difficult to reach the correct answer because it takes a lot of time to implement the processing procedure as intended,as in the problem discussed this paper.However,even if the Dynamic Programming programming is applied,if the processing procedure includes multiple loops,the correct answer may not be reached depending on the constraints.

In that case,it was confirmed that the calculation time could be reduced by searching for and implementing an idea to reduce the number of loops.

Keywords: Dynamic Programming, Coordinate compression, Memoization recursive function, Algorithm

１．はじめに

アルゴリズム実技検定（ＰＡＳＴ）とは、競技プログラミングコンテストを運営するＡｔＣｏｄｅｒ株式会社が実施する検定で、プログラミングスキルが問われる問題が与えられ、制限時間５時間以内にその解法をプログラミングしてオンラインで提出するものである。問題は１５問与えられ、１００点満点で採点される。獲得した点数に応じて、エントリー、初級、中級、上級、エキスパートの５段階でランクが認定される。

本稿では、２０１９年１２月に実施された第一回アルゴリズム実技検定のO問題を対象とし、その解法として筆者が適用した動的計画法および、その他の技法についての有用性を検証し、動的計画法とループ処理を削減するアイデアの実装が計算時間の削減に効果があることを確認した。

２．座標圧縮

制約から分かるように、さいころの目は小さい数字から順番に使用されるとは限らない。ただし、同じ数は使用されない。この問題では、目の大小関係だけが吟味されるので、目に使用された数を新たに１から順次割り当て直す座標圧縮の技法が適用できる。

プログラミング言語c++には連想記憶クラスのmap（写像、mapping）があり、#include<map>を宣言することで利用可能となる。連想配列クラスとは、検索可能なキーとそのキーに対応する値をペアで保持し、キーを指定することで対応する値を高速に取り出すことができる仕組みを持つクラスのことである。

【code1】の記述により、入力されたさいころの目のデータを座標圧縮して、新たに配列b[i][j]に格納した。

３．動的計画法（Dynamic Programming）による解法

動的計画法は計算量を減らす設計技法のひとつとして広く知られており、トップダウン方式とボトムアップ方

式に大別できる。トップダウン方式は、一度計算した結果を記録しておき、あとで利用する方式で、メモ化再帰とも呼ばれる技法である。ボトムアップ方式は分割統治法、または漸化式ループとも呼ばれる技法である。

例えばフィボナッチ数列を求める場合、定義通りの素朴な計算法を採用すると【code2】に示したようなプログラムとなる。筆者の環境で【code2】を用いてフィボナッチ数列の第40項までを計算すると、6,683ms（6.683秒）かかった。【code3】に示すメモ化再帰の手法を採用したトップダウン方式のプログラムを用いて同様の計算を実行すると、処理時間は21ms（0.021秒）であった。また、【code4】に示すボトムアップ方式のプログラムを用いた場合の処理時間も同じく20ms（0.020秒）であった。

素朴な計算法を採用すると、フィボナッチ数列の処理時間は極端に増える。例えば【code2】を用いてフィボナッチ数列の第45項までを計算すると、74,334ms（1分14秒程度）かかる。一方、メモ化再帰の手法を採用したトップダウン方式【code3】では23ms（0.023秒）、ボトムアップ方式【code4】の計算時間も同じく23ms（0.023秒）であった。このように動的計画法を適用すると問題解法の処理時間を大幅に削減できることがわかる。

今回、検証の対象として取り上げたO問題においては、dp[j]を「jがでた後、のこり何回振れるかの期待値」とし、dp[j]をjの降順に計算することによってトップダウン方式の動的計画法を適用できる。【code5】に題意通りの処理手順を実装した。このコードによって正解を得ることができる。ただし、【code5】には▲1から▲3で示すようにループ処理が3重になる部分が含まれ、効率的とは言えない。制約にあるように最大30,000個のさいころがある場合、計算回数は少なくとも▲1（目の総数）×▲2（さいころの数）×▲3（目的の目の場所の探索）=(30,000×6)×(30,000)×（α）=54億×α 回以上が見込まれる。この計算量は1回あたりの処理に1マイクロ秒かかるとして6日以上の計算時間が必要で、パソコンで正解を得るには計算量を減らす工夫が必須となる。

４．計算量削減の工夫

最初に▲2のループを削るアイデアを検討した。すべてのさいころに対して、目的の目があるかどうかを全検索している部分である。この部分を改良して、それぞれの目がどのさいころにあるのかを管理しておけば、該当するさいころだけを検索の対象とすることができる。つまり、さいころの数をnとすれば、この事前処理により計算量を1/nに削減できる。そこで【code6】に示すような処理を追加した。具体的には、それぞれの目がどのさいころにあるかの情報を格納する連想記憶クラスのインスタンスmpを準備し、さいころの目jを検索キーとしてjの含まれるさいころの値が簡単に示せるように工夫した。

さらに、目的の目の場所を探索する手順の改良を検討した。▲3ではさいころ毎にjより大きな目を探索し、その期待値を加算することにより、それらのうちの最大値からdp[j]を求めたが、さいころ毎に期待値を保持する変数を準備すれば、j+1の目がひとつしかないことから、加算が必要なさいころはひとつなので、更新回数を1回に減らすことができる。

【code7】でわかるように、計算回数は▲6（目の数）×▲7（特定のさいころ）×α=(30,000×6)×1×αとなり、計算時間を最大30,000分の1に削減することができた。

５．まとめ

今回取り上げた問題のように、題意通りの処理手順を実装すると計算時間がかかり、正答にたどり着けない場合、動的計画法の適用が有用であることがわかった。ただし、動的計画法を適用しても処理手順に多重ループが含まれれば、制約によっては正答にたどり着けない場合がある。その場合、ループの数を減らすアイデアを模索して実装することで、計算時間を削減できることが確認できた。

今回筆者が作成したプログラムのヘッダー部分を【code0】に示した。このコードのAtCoderでの実行時間は357msであった。

６．参考文献

渡部有隆「アルゴリズムとデータ構造」マイナビ出版、2015年

AtCoder「アルゴリズム実技検定」（最終閲覧日2020年7月10日） https://past.atcoder.jp/

第一回PAST O問題　解説（最終閲覧日2020年7月10日）
https://www.youtube.com/watch?v=c-T4WOPPdII&list=PLLeJZg4opYKaru-yFYYQmp4GAg4Ewyg8I&index=16&t=0s

７．参考資料

筆者の環境：ハードウエア

ノートパソコン　DELL ALIENWARE 17R5（2019年製）

プロセッサ i9-8950HK CPU @2.90GHz

メモリ　32.0GB

64ビットオペレーションシステム x64ベース

ソフトウエア

Microsoft Visual Studio Community 2019 Version 16.6.3

Visual C++ 2019

【code0】

```
// PAST 0問題
// さいころの期待値　c++
#include<iostream> // cin cout などの基本的な入出力機能を利用することを宣言する
#include <map> // 連想記憶クラスを利用することを宣言する
#algorithm> // max min sort などを関数を使うことを宣言する
#define rep(i,n) for(int i=0;i<(n);++i)
using namespace std;
double dp[180010]={}; // 動的計画法　メモ化で使用
int a[30030][6]={}; // さいころ
int b[30030][6]={}; // 座標圧縮後のさいころ
```

【code1】座標圧縮

```
//          入力データ                                    座標圧縮後
//  さいころ0 { 12,237,374,111,247,234 }                { 4,13,15,11,14,12 }
//  さいころ1 { 857,27,98,65,83,90 }      →  写像  →    { 17,5,10,7,8,9 }
//  さいころ2 { 764,60,999,11,7,4 }        (mapping)     { 16,6,18,3,2,1 }

int main(){
  int n;
  cin >> n;
  map<int,int>dice; // 座標圧縮の作業用連想記憶クラス dice をつくる
  rep(i,n) rep(j,n){
                cin>>a[i][j]; // 各さいころに書かれている目の数を受け取る
                dice[a[i][j]]=0; // 読み込んだ目の数をキーにして連想記憶クラスdiceに格納する。
                                 // 対応する値は取敢えず0とする。
                }
  int num=1;
  for(auto x:dice){
        dice[x.first]=num; // 検索キーの昇順に記憶されているので、順番に対応する値を変更する。
        num++;
        }
  // 入力された目の数（検索キーx.first）と、昇順に付けかえた数値（値x.second）が対応している。
  rep(i,n) rep(j,n){
        b[i][j]=dice[a[i][j]];
        }  // i番目のさいころの目jは座標圧縮されてb[i][j]に保存された。
```

```
// Fibonacci number  フィボナッチ数列
//
// 1+1=2
//   1+2=3
//     2+3=5
//       3+5=8
//         5+8=13
//           8+13=21
//             13+21=34
//               21+34=55
```

【code2】定義通りの再帰によるフィボナッチ数列の計算法

```
int fib(int n) {
        if (n <= 1) return 1;
        return fib(n - 1) + fib(n - 2);
}
```

【code3】メモ化再帰によるフィボナッチ数列の計算法

```
bool flag[100]; // false:未計算 true:計算済
int dp[100]; // 計算結果のメモ領域
int fib2(int n) {
        if (n <= 1) return 1;
        if (flag[n] == true) return dp[n];
        dp[n] = fib2(n - 1) + fib2(n - 2);
        flag[n] = true;
        return dp[n];
}
```

【code4】ボトムアップ方式の動的計画法によるフィボナッチ数列の計算法

```
int fib3(int n) {
        if (n == 0) return 1;
        int f3 = 1;
        int f2 = 1;
        int f1 = 1;
        for (int i = 0; i < n-1; i++) {
                f3 = f1 + f2;
                f1 = f2;
                f2 = f3;
        }
        return f2;
```

【code5】題意通りの処理手順（ナイーブな方法）

```
▲1 for(int j=num-1;j>=1;j--){ // num-1:座標圧縮後の目の最大値
    dp[j]=0.0; // jが出た後、あと何回振れるかの期待値の最大値
    ▲2 rep(i,n){ // n:さいころの数
        double cur=1.0;
      ▲3 rep(k,6) if(b[i][k]>j) cur+=dp[b[i][j]]/6.0; // jよりも大きな目があれば期待値を加算
        dp[j]=max(dp[j],cur); // 期待値の最大値をdp[j]に保存
```

```
        }
    }
    double mx=0.0;
    rep(i,n){ // n:さいころの数
        double ans=0.0;
        rep(j,6) ans+=dp[b[i][j]]; // ans:各さいころの期待値
        mx=max(mx,ans); // mx/6.0が一番期待値の大きなさいころの期待値
    }
    printf("%2.14lf¥n",1.0+mx/6.0);
    return 0;
```

【code6】各目がどのさいころに含まれるのかを管理する部分を【code1】に追加する

```
    map<int,int>mp; // ▲4  追記
  // 入力された目の数（検索キーx.first）と、昇順に付けかえた数値（値x.second）が対応している。
  rep(i,n) rep(j,n){
        b[i][j]=dice[a[i][j]];
        mp[b[i][j]]=i; // ▲5  追記
        }  // i番目のさいころの目jは座標圧縮されてb[i][j]に保存された。
           // b[i][j]がi番目のさいころにあることをmpで管理する
```

【code7】各さいころの期待値を保持する変数vを準備し、更新回数を1回に削減する

```
double v[n] // n:さいころの数
rep(i,n)v[i]=1.0;
double mx_v=0.0;
dp[num-1]=1.0; // num-1:座標圧縮後の目の最大値
▲6 for(int j=num-2;j>=0;j++){
      ▲7 int i=mp[j+1]; // j+1の目のあるさいころはi、さいころiのみ更新すればよい
        rep(k,6){
        if(b[i][k]==j+1){
                v[i]+=dp[j+1]/6.0;
                break;
        }
      }
     if(mx_v<v[i]) mx_v=v[i];
     dp[j]=mx_v;
    }
```

第一回アルゴリズム実技検定（ＰＡＳＴ）O問題

問題文

偏りのない6面さいころがN個あり、i番目のさいころ(1≦i≦N)のj番目の面(1≦j≦6)には整数Aijが書かれている。高橋君は、1個のさいころを選んで1回振る、という操作を繰り返す。ただし、2回目以降の操作で、前回の操作で出た目より小さいか同じ目が出てしまったら、操作をやめる。各回にどのさいころを振るかは、前回に出た目を見てから選ぶことができる。

高橋君は、できるだけさいころを多く振りたいと考えている。操作が行われる回数の期待値が最大化されるような選択が行われたときの操作回数の期待値を求めよ。

制約

・1≦N≦30,000

・1≦Aij≦1,000,000,000
・Aij はすべて異なる。
・入力中の値はすべて整数である。

出力

操作回数の期待値を表す実数を出力せよ。
ジャッジの出力との絶対誤差または相対誤差が 0.000001 以下であれば正解と判定される。

入力例１

2
1 2 3 4 5 6
7 8 9 10 11 12

出力例１

3.64925355954377

入力例２

3
12 237 374 111 247 234
857 27 98 65 83 90
764 60 999 11 7 4

出力例２

3.42188884244970

Problem Statement

We have N unbiased six-sided dice. The j-th face (1≤j≤6) of the i-th die (1≤i≤N) has an integer Aij written on it.
Takahashi repeats the following operation: choose a die and toss it. However, in the second and subsequent operations, if the die shows a number that is less than or equal to the number shown in the previous operation, the process ends. The die to toss each time can be chosen after seeing the previous number shown.
Takahashi wants to toss a die as many times as possible. Find the expected number of operations done when the choices are made to maximize this number.

Constraints

・1≤N≤30000
・1≤Aij≤1,000,000,000
・Aij are all different.
・All values in input are integers.

(出典　https://atcoder.jp/contests/past201912-open/tasks/past201912_o)