アルゴリズムとデータ構造 プログラミング演習

森 立平 mori@c.titech.ac.jp

2017 年 6 月 27 日

今日の目標

- 動的計画法（整数入出力の関数の計算，最適解の発見などは含まない）のプログラムを書けるようになる．

今日の主な課題 (提出締切は授業終了時)

1. 分割数のプログラムを書く．

今日のワークフロー

1. この資料をよく読み動的計画法の考え方，トップダウン法，ボトムアップ法について理解する．
2. 3章に書いてある課題に取り組む．

1 動的計画法とは

1.1 分割統治法から動的計画法へ

動的計画法 (dynamic programming) は元の問題から小さな問題を作り出し，それらを解くことで元の問題を解く再帰アルゴリズムの枠組みである．フィボナッチ数列の例を考えてみよう．フィボナッチ数列は次のように定義される．

$$f_1 = f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad n \geq 3.$$

この再帰的な関係をそのままアルゴリズムにしてみよう．C 言語で書くと次のようなプログラムになる（この関数の引数は必ず 1 以上の整数でなくてはならない）．

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

このアルゴリズムの計算量 $T(n)$ を「f_n を計算するためにアルゴリズムの中で実行する加算の回数」と定義すると，$T(n)$ は次の漸化式を満たす．

$$T(1) = T(2) = 0$$
$$T(n) = T(n-1) + T(n-2) + 1 \qquad n \geq 2.$$

この式を良く見ると $T(n) + 1 = f_n$ なので，漸近的に $T(n)$ は n 番目のフィボナッチ数 f_n と同じくらい大きい．フィボナッチ数は指数関数的に大きい (だいたい $\left(\frac{1+\sqrt{5}}{2}\right)^n$) のでこのアルゴリズムはとても効率が悪い．どうしてこの計算の効率が悪いのかもう少し具体的に考えてみよう．例えば，f_{10} を計算するために f_9 と f_8 を計算する必要があるが，f_9 を計算するためにも f_8 を計算する必要がある．ここで f_8 の計算を 2 回することになるがこれは明らかに無駄である．動的計画法の基本的な考え方は「**計算結果を配列に保存しておいて無駄な計算の重複を避ける**」ということである．この考え方に基づいてフィボナッチ数列を計算するための動的計画法を設計すると次のようになる．

```
//十分大きなサイズの配列(99番目までのフィボナッチ数が計算できる)
//この配列はグローバル変数なので0で初期化されている
int dp[100];

//main関数の中で dp[1] = dp[2] = 1 と代入しておく
int fib(int n){
//計算済みの場合は配列の値を返す
  if(dp[n] > 0) return dp[n];
  dp[n] = fib(n - 1) + fib(n - 2);
  return dp[n];
}
```

このプログラムの中では各 $i \in \{3, 4, \ldots, n\}$ について fib(i) の計算は 1 回ずつしか行なわれない．そのため fib(n) を計算するために実行される加算の回数は $n-2$ である．このように再帰的な関数呼び出しの中で計算結果を配列に保存していく方法を「**トップダウン法** (もしくはメモ化再帰法)」と呼ぶ．

一方で，小学生に「10 番目のフィボナッチ数を計算しなさい」と言ったら，こんな複雑な計算方法はとらないであろう．おそらく，小学生は紙に 1 1 2 3 5 8 13 ... と順番に書いていくはずだ．この方法は明らかに $n-2$ 回の加算で n 番目のフィボナッチ数を計算している．この小学生のアルゴリズムを C 言語にすると以下のようになる．

```
//十分大きなサイズの配列(99番目までのフィボナッチ数が計算できる)
int dp[100];

int fib(int n){
  int i;
  dp[1] = dp[2] = 1;
  for(i = 3; i <= n; i++){
    dp[i] = dp[i-1] + dp[i-2];
  }
  return dp[n];
}
```

このように再帰呼び出しを使わないで順番に配列を埋めていく方法を「**ボトムアップ法**」と呼ぶ．このフィボナッチ数列の例ではそれまでに計算したフィボナッチ数のうち最新の 2 つだけ覚えておけばよいので，配列を使わずにすますこともできる（そのような工夫が一般の動的計画法で可能であるとは限らない）．なお，余談だがフィボナッチ数は $O(\log n)$ 回の加算と乗算で計算することもできる（上のアルゴリズムが小学生のアルゴリズムなら，これは大学生のアルゴリズムといったところか）．「トップダウン法」と「ボトムアップ法」は基本的にはどちらが優れているということはないが，両方の方法でアルゴリズムを構築し，プログラムが書けることが望ましい．以下ではより複雑な問題に対する動的計画法の手法について説明する．

1.2 再帰的な関係式の導出

トップダウン法とボトムアップ法のどちらの手法でも，まず最初に関数が満たす再帰的な関係式を導出しなければならない．フィボナッチ数の場合は定義がそのまま再帰的な関係式になっていたが，一般的にはそうとは限らない．一般的には以下に見るよう「**補助問題**」を定義する必要がある．自然数 n に対する分割数 $p(n)$ を「n を自然数の和として表現する方法の個数（順番の違いは同一視する）」と定義する．例えば $n=5$ の場合，

$$1+1+1+1+1, \quad 1+1+1+2, \quad 1+1+3,$$
$$1+4, \quad 5, \quad 1+2+2, \quad 2+3$$

と 7 通りの表し方があるので $p(5)=7$ である．この分割数を計算するアルゴリズムを考えよう．フィボナッチ数と違って分割数は再帰的に定義されていないし，そのような関係式を持つとも思えない．そこで元の問題に対する「**補助問題** (もしくは補助関数)」を定義する．分割数の場

合，補助関数として $q(k,n) :=$「k 以下の数だけを使った n の分割の総数」 と定義する．すると

$$\begin{aligned} &q(0,0) = 1, && \\ &q(0,n) = 0, && n \geq 1 \\ &q(k,n) = q(n,n), && k > n \geq 0 \\ &q(k,n) = q(k-1,n) + q(k,n-k), && n \geq k \geq 1 \end{aligned}$$

という再帰的な関係式が成立する．また $q(n,n) = p(n)$ である．そこで，以下ではこの補助関数 $q(k,n)$ をフィボナッチ数の例のように効率的に計算する方法を考える．

一般的に補助問題を定義するときには

(1) 補助問題の解から元の問題が効率良く解ける．

(2) 補助問題の解の間に再帰的な関係式が成立する．

という 2 つの条件を満たすようにする必要がある．フィボナッチ数の例に見たように，一般的に再帰的な関係式をそのまま再帰アルゴリズムにするととても効率が悪い．しかし，計算結果を配列に保存しておくことで効率化できる．**配列に計算結果を保存しておいて無駄な再計算を避ける**というのが動的計画法の考え方である．

実のところ，補助問題を定義するのが動的計画法で最も難しいところである．ひとたび補助問題が定義できれば，あとは機械的に補助問題の解を「トップダウン法」もしくは「ボトムアップ法」で計算できる．

1.3 トップダウン法 (メモ化再帰法)

トップダウン法は極めて単純な方法である．フィボナッチ数列の定義に従ったプログラムのように，分割数の補助関数 $q(k,n)$ の再帰的な関係をそのままプログラムにすると次のようになる (この関数の 2 つの引数は必ず 0 以上の整数でなくてはいけない)．

```
int q(int k, int n){
  if(k == 0 && n == 0) return 1;
  else if(k == 0) return 0;
  else if(k > n) return q(n, n);
  else return q(k - 1, n) + q(k, n - k);
}
```

もちろんこれはとても効率が悪いプログラムである．そこで，一度計算した値を覚えておいて再計算を避けるようにしよう．二次元配列 dp を用意し，dp[k][n] に $q(k,n)$ の値を保存することにしよう．計算済みかどうか区別するために，未計算の「しるし」として最初に dp には -1 を代入しておこう (フィボナッチ数列の場合は 0 が未計算の「しるし」であったことに注意せよ)．C 言語プログラムは次のようになる．

```
//十分大きなサイズの配列（main 関数の中で -1 を全ての要素に代入しておく）
int dp[100][100];

int q(int k, int n){
//計算済みの場合は配列の値を返す
  if(dp[k][n] != -1) return dp[k][n];
//そうでない場合は計算結果を配列に保存しておく
  if(k == 0 && n == 0) dp[k][n] = 1;
  else if(k == 0) dp[k][n] = 0;
  else if(k > n) dp[k][n] = q(n, n);
  else dp[k][n] = q(k - 1, n) + q(k, n - k);
  return dp[k][n];
}
```

元のプログラムと比較すると変更点は

- 関数の先頭で計算済みかどうかチェックする．

- 計算済みでない場合は，再帰的な関係に従って解を計算し，return する代わりに dp[k][n] に代入する．

- 最後に dp[k][n] を return.

の 3 つだけである．上の C 言語プログラムには書かれていないが main 関数の中で配列の中身を全て-1 で初期化する必要がある．もう少し工夫して $k = 0$ の場合の解をあらかじめ main 関数の中で保存しておくと，C 言語プログラム全体は次のようになる．

```
#include <stdio.h>

//計算する分割数の入力の最大値
#define N 1000

//十分大きなサイズの配列
int dp[N+1][N+1];

int q(int k, int n){
//計算済みの場合は配列の値を返す
  if(dp[k][n] != -1) return dp[k][n];
//そうでない場合は計算結果を配列に保存しておく
  if(k > n) dp[k][n] = q(n, n);
  else dp[k][n] = q(k - 1, n) + q(k, n - k);
  return dp[k][n];
}

int main(){
  int k, n, i;

  scanf("%d", &n);

// q(0,0) = 1 なので配列に解を保存しておく
  dp[0][0] = 1;

// q(0,n) = 0 なので配列に解を保存しておく(省略可)
  for(i = 1; i <= n; i++){
    dp[0][i] = 0;
  }

/*
  それ以外の場合は未計算なので，
  その「しるし」として -1 を代入しておく
*/
  for(k = 1; k <= n; k++){
    for(i = 0; i <= n; i++){
      dp[k][i] = -1;
    }
  }

  printf("%d\n", q(n, n));

  return 0;
}
```

特に説明が必要無いくらい単純ではないだろうか？「補助問題を定義して再帰的な関係式さえ立てられればトップダウン法は簡単に書けるぜ」というようになってほしい．

1.4 ボトムアップ法

ボトムアップ法はトップダウン法よりやや洗練された方法である．分割数の補助関数 $q(k,n)$ を計算するためには，同じ補助関数の異なる引数に対する値 $q(k',n')$ が必要であるが，このとき**常に** $k' \leq k$ **と** $n' \leq n$ **が成り立つ**．よって，もし $q(k',n')$ の値が全ての $k' \leq k$，$n' \leq n$，$(k',n') \neq (k,n)$ について得られていれば，$q(k,n)$ は効率的に計算ができる．よって小さな k, n から順番に $q(k,n)$ を計算して配列に結果を代入していけばよい．$q(k,n)$ **を計算するときには既に** $q(k',n')$ **は全ての** $k' \leq k$，$n' \leq n$，$(k',n') \neq (k,n)$ **について計算済みなので**効率的に $q(k,n)$ を計算することができる．

```
#include <stdio.h>

//計算する分割数の入力の最大値
#define N 1000

//十分大きなサイズの配列
int dp[N+1][N+1];

int main(){
  int k, n, i;
  scanf("%d", &n);

// q(0,0) = 1 なので配列に解を保存しておく
  dp[0][0] = 1;

// q(0,n) = 0 なので配列に解を保存しておく(省略可)
  for(i = 1; i <= n; i++){
    dp[0][i] = 0;
  }

  for(k = 1; k <= n; k++){
    for(i = 0; i < k; i++){
      dp[k][i] = dp[i][i];
    }
    for(i = k; i <= n; i++){
      dp[k][i] = dp[k - 1][i] + dp[k][i - k];
    }
  }

  printf("%d\n", dp[n][n]);

  return 0;
}
```

配列を効率的に埋める上で必要なことは $q(k,n)$ **を計算するときには既に** $q(k',n')$ **は全ての** $k' \leq k$，$n' \leq n$，$(k',n') \neq (k,n)$ **について計算済みである**ということだけである．よって二次元配列を埋めていく順番は違う順番でもよい（上のプログラムでは二次元配列の行を順番に埋めていくが，列を順番に埋めていくのでもよい）．

ボトムアップ法はトップダウン法ほど単純ではないが，ボトムアップ法も書けるようにしておくことが望ましい．一般的にボトムアップ法はトップダウン法よりプログラムが高速であることが多い（ただし時間計算量のオーダーは変わらない）．また，ボトムアップ法は工夫をすることで**空間計算量**（使用するメモリ量のこと）を改善できる場合がある．例えばフィボナッチ数の場合，最新の 2 つの値だけ覚えておけばよいので空間計算量を $O(n)$ から $O(1)$ に改善できる．同様に分割数の場合も工夫をすることで，空間計算量を $O(n^2)$ から $O(n)$ に改善できることを1.6 章で説明する．このような改善はトップダウン法では実現できない．

1.5 動的計画法の時間計算量

動的計画法の時間計算量はほとんどの場合「配列の要素を 1 つ埋めるのにかかる計算量 × 配列のサイズ」で計算できる．分割数の例の場合，$p(n)$ を計算するためには $q(n,n)$ を計算する必要があり，配列のサイズは n^2 である．ひとつの要素を計算するのに高々 1 回の加算をするので，分割数を計算する時間計算量は $O(n^2)$ である．

1.6 動的計画法の空間計算量（後半は発展的な内容）

アルゴリズムの中で使用するメモリの量のことを**空間計算量**と呼ぶ．動的計画法の空間計算量は使用する配列のサイズである．ボトムアップ法は工夫することで**空間計算量**を改善することができる場合がある．フィボナッチ数の場合，最新の 2 つの値だけ覚えておけばよいので空間計算量を $O(n)$ から $O(1)$ に改善できた．分割数の場合も以下のような工夫をすることで，空間計算量を $O(n^2)$ から $O(n)$ に改善できる（発展的な内容なので理解しなくてもよい）．補助関数 $q(k,n)$ が満たす再帰的な関係式を次のように考える (3 行目のみ変更されている)．

$$
\begin{aligned}
&q(0,0) = 1, && \\
&q(0,n) = 0, && n \geq 1 \\
&q(k,n) = q(k-1,n), && k > n \geq 0 \\
&q(k,n) = q(k-1,n) + q(k,n-k), && n \geq k \geq 1
\end{aligned}
$$

すると，$q(k,n)$ を計算するときには $q(k-1,n)$ と $q(k,n-k)$ の値だけ分かればよい．ここで $q(k,n)$ を計算するために参照する $q(k',n')$ の第一引数 k' は常に k もしくは $k-1$ なので二次元配列のうち $q(k-1,\cdot)$ と $q(k,\cdot)$ にあたる 2 行だけを覚えておけばよい．さらに，$q(k,n)$ の計算の中で一つ上の行 $q(k-1,\cdot)$ で参照するのは $q(k-1,n)$ だけなので，行を直接書き換えていくことができ，記憶するのを 1 行だけにできる．C 言語プログラムは以下のようになる．

```
#include <stdio.h>

//計算する分割数の入力の最大値
#define N 1000

//十分大きなサイズの配列
int dp[N+1];

int main(){
  int k, n, i;
  scanf("%d", &n);

// q(0,0) = 1 なので配列に解を保存しておく
  dp[0] = 1;

// q(0,n) = 0 なので配列に解を保存しておく(省略可)
  for(i = 1; i <= n; i++){
    dp[i] = 0;
  }

  for(k = 1; k <= n; k++){
    for(i = k; i <= n; i++){
      dp[i] += dp[i - k];
    }
  }
  printf("%d\n", dp[n]);

  return 0;
}
```

解説はしない．とても短くて美しいプログラムだと思わないだろうか．

2 動的計画法についてのまとめ

- 動的計画法は，小さなサイズの問題を解くことで元の問題の解を計算するアルゴリズムの枠組みである．
- 動的計画法ではまず補助問題を定義する必要がある．ここで補助問題は (1) 補助問題の解から元の問題の解が効率的に計算できる，(2) 補助問題の解の間に再帰的な関係式が成立する，という 2 つの性質を満たしている必要がある．
- 補助問題の解を保存しておくための配列を用意しておき，その配列を参照することで無駄な再計算を避ける．
- トップダウン法は補助問題を計算する関数の先頭で計算済みかどうか配列をチェックして，計算済みであればその値を返す．計算済みでない場合は再帰呼び出しして解を計算し，その解を配列に代入してから解を返す．
- ボトムアップ法は補助問題の解の依存関係から，再帰呼出でなくループで配列を埋めていく．それぞれの配列の要素を埋めるのに必要な他の要素は既に計算済みであることが必要である．
- 動的計画法の時間計算量はほとんどの場合「配列の要素を 1 つ埋めるのにかかる計算量 × 配列のサイズ」で計算できる．
- 動的計画法の空間計算量のオーダーは配列のサイズのオーダーと等しい．ボトムアップ法の場合，空間計算量を改善できる場合がある．

今日はフィボナッチ数や分割数など入出力が整数の問題のみを扱ったが，もっと複雑な問題にも動的計画法は適用できる．レポート 1 に挙げた問題はその典型的な例である．いきなりレポート 1 に取りかかるのは難しいので，まずは次章の問題を考えよう．

3 今日の課題

関数 $p_1(n)$ を「自然数 n の相異なる自然数による分割の総数」と定義する．レポート 1 と同様に

(2) 関数 $p_1(n)$ の補助関数の定義

(3) 補助関数が満たす再帰的な関係式

(4) $p_1(n)$ の補助関数をボトムアップ法で計算するアルゴリズムの擬似コードによる説明

(6) 設計した動的計画法の時間計算量のオーダーとその説明

及び $p_1(200)$ の値を A4 用紙に記入して提出せよ．

C 言語についての注意

原則として変数を宣言して値を代入しする前に変数の値を参照しないようにすること．ただし，グローバル変数は必ず 0 で初期化されている（0 が代入されている）．この原稿の中では動的計画法のための配列はすべてグローバル変数であるので，配列に 0 を代入するところは省略できる．このことを理解した上でグローバル変数に 0 を代入せずに参照するのはよい．また，コンパイラ `clang` にオプション `-Wall` をつけてコンパイルするとコンパイル時に未初期化ローカル変数の参照を警告してくれる．