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

2016 年 6 月 28 日

今日の目標

- 動的計画法（整数論的な関数の計算，問題の入力は整数で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 (提出締切は金曜日の授業開始時)

1.

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. 4 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

分割統治法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った．動的計画法も基本的な考えは同じであるが，再計算を省略することでより広い範囲の問題を効率的に解くことができる．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．

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

さて，分割統治法の考え方に乗っ取って，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（この関数の入力は必ず 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．しかし，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，こんなバカ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．とても簡単なようだが，これが動的計画法の考え方である．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
int F[100];

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

このアルゴリズムでは今まで計算した値を覚えておくために配列を使っている (これは小学生が使う紙に相当する)．実際にはそれまでに計算したフィボナッチ数で最新の 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)．以下では一般的な動的計画法の手法について見ていく．動的計画法には「**トップダウン法**」と「**ボトムアップ法**」という二つの実現方法がある．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．上の小学生のアルゴリズムはボトムアップ法にあたる．

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

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

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，この補助関数をフィボナッチ数の小学生のアルゴリズムの例のように効率的に計算することを考えるのである．

一般的に補助問題を定義するときには

- 補助問題の解から元の問題が効率良く解ける．
- 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場合がある．例えばフィボナッチ数の再帰的な定義をそのまま再帰アルゴリズムにした場合，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．**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」で計算すればよい．

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．フィボナッチ数の定義に従ったプログラムのように，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（関数 min は別途定義する必要がある）．

```
int q(int k, int n){
  if(k == 0 && n == 0) return 1;
  else if(k == 0) return 0;
  else return q(k - 1, n) + q(min(k, n - 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 Q を用意し，Q[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 Q には -1 を代入しておこう．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 Q[100][100];

int q(int k, int n){
//計算済みの場合は配列の値を返す
  if(Q[k][n] != -1) return Q[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) Q[k][n] = 1;
  else if(k == 0) Q[k][n] = 0;
  else Q[k][n] = q(k - 1, n) + q(min(k, n - k), n - k);
  return Q[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 Q[k][n] に代入する．
- 最後に Q[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>

#define N 101

//十分大きなサイズの配列
int Q[N][N];

int min(int a, int b){
  if(a < b) return a;
  else return b;
}

int q(int k, int n){
```

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

int p(int n){
  return q(n, n);
}

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

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

// q(0,n) = 1 なので配列に解を保存しておく
  for(n = 1; n < N; n++){
    Q[0][n] = 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(n = 0; n < N; n++){
      Q[k][n] = -1;
    }
  }

  for(i = 0; i < N; i++){
    printf("%d: %d\n", i, p(i));
  }

  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>

#define N 101

//十分大きなサイズの配列
int Q[N][N];
```

```
int min(int a, int b){
  if(a < b) return a;
  else return b;
}

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

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

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

  for(k = 1; k < N; k++){
    for(n = k; n <= N; n++){
      Q[k][n] = Q[k - 1][n] + Q[min(k, n - k)][n - k];
    }
  }

  for(i = 0; i < N; i++){
    printf("%d: %d\n", i, Q[i][i]);
  }

  return 0;
}
```

ここで大事なのは $q(k,n)$ 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である．なので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 章で見る．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（実際には $k \leq n$ の場所だけ考えるので配列のサイズはこれの約半分と考えられる）．ひとつの要素を計算するのに 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．

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

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>

#define N 101

//十分大きなサイズの配列
int Q[N];

int main(){
  int k, n;

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

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

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

  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 動的計画法の例

これまでの動的計画法の例はフィボナッチ数，分割数と整数論的な関数の計算であった．ここでは

3 動的計画法についてのまとめ

- 動的計画法は分割統治法と同様，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．補助問題を解くのに必要な他のパラメータに対する補助問題の解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．

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．そのような問題の例としてレポートにもなっている「宝詰め込み問題」がある．そのような問題の場合，出力は最適値以外に最適解の場合もある．金曜日のプログラミング演習ではそのような問題の例を扱う．

4 今日の課題

関数 $p_1(n)$ を「自然数 n の相異なる整数による分割の総数」と定義する．関数 $c(n)$ を「n 円を 1 円玉，10 円玉，50 円玉，100 円玉，500 円玉を使って支払う方法の数（ぴったり n 円支払わないといけない)」と定義する．ここで，各硬貨は無限に持っているものとする．

1. 関数 $p_1(n)$ の補助関数を定義し，その補助関数が満たす再帰的な関係式を導出せよ．
2. $p_1(n)$ の補助関数をボトムアップ法で計算することで $p_1(n)$ を計算する C 言語プログラムを書け．
3. 関数 $c(n)$ の補助関数を定義し，その補助関数が満たす再帰的な関係式を導出せよ．
4. $c(n)$ の補助関数の再帰的な関係式をそのまま再帰アルゴルズムにして $c(n)$ を計算する C 言語プログラムを書け．
5. $c(n)$ の補助関数をトップダウン法で計算することで $c(n)$ を計算する C 言語プログラムを書け．
6. $c(n)$ の補助関数をボトムアップ法で計算することで $c(n)$ を計算する C 言語プログラムを書け．

発展的課題: