基礎プログラミング+演習 # 12– さまざまな型+動的計画法

久野 靖 (電気通信大学)

2017.12.11

今回は次の内容を取り上げます。

- C プログラムの全体構造、C のさまざまな型
- 動的計画法

1 前回演習問題の解説

1.1 演習 1 — 簡単な計算

これは簡単なのでほぼプログラムのみ示します。まず四則 (書式指定%g については後述)。

```
// sisoku1 --- add/sub/mul/div/mod
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  double x = atof(argv[1]), y = atof(argv[2]);
  printf("add: %g\n", x + y);
  printf("sub: %g\n", x - y);
  printf("mul: %g\n", x * y);
  printf("div: %g\n", x / y);
  return 0;
}
```

次の剰余ですが、C では剰余は整数どうしでしか許されません。また実行結果も Ruby と違いますね (ここでは省略)。

```
// jouyo1 --- test mod operation
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  int x = atoi(argv[1]), y = atoi(argv[2]);
  printf("mod: %d\n", x % y);
  return 0;
}
```

円錐はまあ簡単ですが、π をどこかで定義してないの? と思われたかも知れませんが、C 言語の標準としては「ありません」。[1]

[1] 正確にやると話が長くなりますが、直接書くか自分定義しましょう。ただし M_PI という名前は避けること。

```
// convol1 --- calculate corn volume from r and h
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  double r = atof(argv[1]), h = atof(argv[2]);
  printf("corn volume: %g\n", r*r*3.141593*h/3.0);
  return 0;
}
```

平方根は前回実行例をお見せしました。

```
// sqrt1 --- calculate square root
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[]) {
  if(argc != 2) { fprintf(stderr, "needs 1 arg.\n"); return 1; }
  double x = atof(argv[1]);
  printf("%.20g\n", sqrt(x));
  return 0;
}
```

1.2 演習 3〜5 — 3 つの方法による平方根の計算

いずれも上にある平方根の関数に置き換えられるべきなので、必要な定義と関数本体のみ示します。まず数え上げ。

```
// enum1 --- calculate sqare root using enumeration
#define N 1000000
double enumsqrt(double x);
(途中略)
double enumsqrt(double x) {
  double dx = x / N;
  for(int i = 0; i < N; ++i) {
    double t = i * dx;
    double y = t*t - x;
    if(y >= 0) { return t; }
  }
  return x;
}
```

実行例を示しますが、1000000 くらいではあまり精度よくないです。

```
% ./a.out 2
1.4142139999999998601
```

つぎに区間 2 分法ですが、回数が分かるように途中経過を印字することもできるようにします (コメントアウトしてあります)。

```
// binsearch1 --- calculate sqare root using binary search
#define EPSILON 1e-10
double binsqrt(double x);
(途中略)
double binsqrt(double x) {
  double a = 0.0, b = x;
  while(b - a > EPSILON) {
    double c = 0.5 * (a + b);
    double y = c*c - x;
//  printf("%.20g\n", c);
    if(y >= 0) { b = c; } else { a = c; }
  }
  return a;
}
```

コメントアウトを外して実行すると次のようになります。遅くはないですが、それなりに反復が必要です (35 回程度)。

```
% ./a.out 2
1
1.5
1.25
1.375
1.4375
1.40625
1.421875
1.4140625
1.41796875
(途中略)
1.4142135621514171362
1.414213562267832458
1.4142135623260401189
1.4142135623260401189
%
```

最後はニュートン法ですが、x0 が 1 つ前の r_i、x1 画次の r_{i+1} で、ループ内で計算につれて順ぐりにコピーしています。これらの値が減少していくことを前提に while の条件を書いているので、ループに入る前で x0 を n、x1 を $2n$ にしています。

```
// newton1 --- calculate sqare root using newton method
#define EPSILON 1e-10
double ntnsqrt(double x);
(途中略)
double ntnsqrt(double n) {
  double x0 = n, x1 = 2*n;
  while(x1 - x0 > EPSILON) {
    x1 = x0; x0 = 0.5*x1 + 0.5*n/x1;
//  printf("%.20g\n", x0);
  }
```

```
  return x0;
}
```

実行してみると、ずっと周回数が少なくて済むことがわかります。

```
% ./a.out 2
1.5
1.4166666666666665186
1.4142156862745096646
1.4142135623746898698
1.4142135623730949234
1.4142135623730949234
%
```

2 C言語のさまざまな型

2.1 C言語のファイルとその内容 exam

C プログラムは複数ファイルから構成できます (プログラムが大規模になると不可欠)。しかし、必要になったら学べば済むので、当面の間はファイル 1 つを前提に説明します。

次に片付けるべきことは、「#」で始まる行の扱いで、ここでは次の 2 つを説明します (1 番目は既出)。

```
#include <ファイル名>
#define 名前 文字列
```

1 番目はシステムの決まった場所からファイルを取り込んでくるのでした。新手の 2 番目は、指定した名前が現れたら指定文字列に置き換える機能で、プログラム冒頭で定数を設定するのに使えます。たとえば平方根を求める際に精度をプログラマが決めていましたが、これを「`#define EPSILON 0.00000001`」のように冒頭で定義し、あとは必要なところに `EPSILON` という名前を使うようにしておくと、後で変更するときに間違いがありません。

これらの「#」で始まる行は、C コンパイラが本体の処理に入る前に **C プリプロセサ** (C preprocessor) と呼ばれるプログラムが処理します。実際、**#include** も **#define** もあらかじめ取り込みや置き換えを行ってしまい、その結果をコンパイラに渡せば済むような仕事ですから。

これらが片付いたとして、あとは C のファイルに含まれるものは次の 3 種類です。

- **変数定義/変数宣言** — この 2 つは形が似ているので一緒に説明します。変数宣言というのは「どこかにこういう名前でこういう型の変数がありますよ」とコンパイラに「知らせる」ことです。そして変数定義はそれに加えて「ここにその変数を作ってください」という指示も含みます。変数定義の形は次のようになります。

  型名 変数名 [= 式] ;

  このような定義が関数の外に (ファイルの中に直接) あった場合、それは**広域変数** (global variable) つまりずっと存在していてどの関数からでもアクセスできる変数として用意されます。

  次に、上記のものの冒頭に **extern** というキーワードがついていると、「このような変数が後で (または別ファイルで) 定義されますよ」という変数宣言になります。ファイル 1 個なら順番を工夫すれば変数定義だけで済むので変数宣言は以後出てきません。

- **関数宣言** — **プロトタイプ宣言**とも呼び、「このような名前でこのような返値の型、パラメタの個数と型のものが後で (または別ファイルで) 定義されます」とコンパイラに教えます。形は次の通り (型名はとりあえず `int` や `double` などだと思ってください)。

```
型名 関数名(型名 変数名, 型名 変数名,...); --- 一般的な場合
型名 関数名(void); --- 引数が無い関数
型名 関数名(); --- パラメタの情報を省略
```

void というのは「無い」ことを表すキーワードで、先頭の型名のところにも書けます (その場合は「値を返さない関数」であることを示します)。

- **関数定義** — これまでは main の関数定義だけから成るプログラムだけ扱って来ましたが、必要に応じて関数をいくつでも定義できます。その形は次の通りです。

```
型名 関数名(型名 変数名, 型名 変数名,...)  {
      文…
}
```

ここからが大事なところです。関数定義の本体 (「文…」の部分) で関数を呼び出したりグローバル変数をアクセスするためには、その箇所よりも「前で」その関数やグローバル変数が定義または宣言されている必要があります (そうしないと型の検査ができません)。本科目では、変数定義を冒頭に集め、その後に main 以外のすべての関数の宣言を書いてから最後に個々の関数定義を置くことで、この条件を満たすようにします。

お話ばかりでは面白くないので、とりあえず例題としてなつかしい最大公約数を計算するものを作ってみました。

```
// gcd1 --- gcd by recursion
#include <stdlib.h>
#include <stdio.h>
int gcd(int x, int y);

int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  int x = atoi(argv[1]), y = atoi(argv[2]);
  printf("gcd(%d,%d) = %d\n", x, y, gcd(x, y));
  return 0;
}
int gcd(int x, int y) {
  if(x == y) {
    return x;
  } else if(x > y) {
    return gcd(x-y, y);
  } else {
    return gcd(x, y-x);
  }
}
```

動かしたようすは次のとおり。

```
% ./a.out 24 60
gcd(24,60) = 12
%
```

printf の使い方が大幅に変わっていて驚かれたかも知れません。説明しておきましょう。printf で第 1 引数として渡す文字列は**書式文字列** (format string) と呼ばれ、基本的にはそのまま出力されますが、その中の「特別な」指定があるとその箇所で指定に応じた埋め込みが行われます。

- `%%` — 「`%`」1 文字を出力する。
- `%d` — 対応する引数 (整数) を十進表現で出力する。
- `%x` — 対応する引数 (整数) を 16 進表現で出力する。
- `%o` — 対応する引数 (整数) を 8 進表現で出力する。
- `%b` — 対応する引数 (整数) を 2 進表現で出力する。(Ruby のみ)
- `%f` — 対応する引数 (実数) を小数点表現で出力する。
- `%e` — 対応する引数 (実数) を仮数+指数表現で出力する。
- `%g` — 対応する引数 (実数) を値に応じ小数点か仮数+指数表現で出力。

「対応する引数」というのは、書式文字列の後に指定した引数を 1 つずつ対応させます (図 1)。最後の `g` は、基本的には小数点表現したいけれど大きすぎる/小さすぎる値は指数表現に切替えてね、という指定になります。

さらに、「`%`」と指定文字 (`d` とか `f` とか) の間に「w」「$w.v$」「$.v$」という形の指定が追加できます。w はその値を出力する幅を指定し、出力をきれいに揃えるために使います。数値の表示が w より小さい場合は左に空白を詰めて指定幅にしますが、w の先頭に数字の「`0`」をつけると空白の代わりに 0 が詰められます。一方 v は実数のみで「小数点以下の表示桁数」です。そういうわけでようやく、「`"%.20g"`」の意味が説明できました。

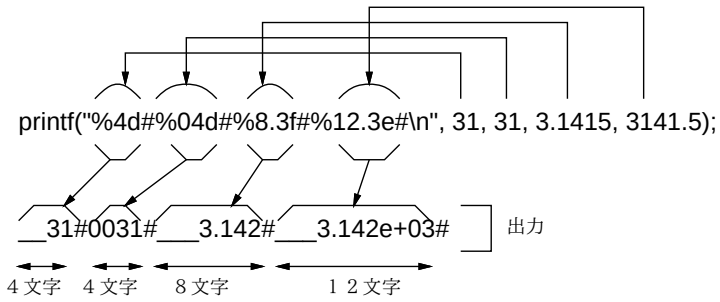

図 1: 書式文字列の機能

演習 1 上の例題を打ち込み (前回の例題等を適宜コピーして修正するのがよい) 動かせ。動いたら、次のものをやってみよ。再帰を使っても使わなくてもよいが、下請け関数は作ること。

a. 整数 n を入力し、n の階乗を表示する。
b. 整数 n を入力し、2^n を表示する。
c. 整数 n を入力し、n 番目のフィボナッチ数を表示する (1 番目から全部表示でもよい)。
d. 整数 n、r を入力し、組合せの数 ${}_nC_r$ を表示する。
e. 下請け関数を 1 個以上使ってうまく記述できる好きな計算を行う。

2.2 変数の存在期間と可視範囲

プログラムの構造について学んだので、それと関連する重要な話題である変数の存在期間ないし**エクステント** (extent) と可視範囲ないし**スコープ** (scope) について説明しておきます。前者は「その変数がどの範囲で存在しているか」、後者は「その変数がどの範囲からアクセスできるか」を意味します。たとえば次のようなプログラムの断片を考えてください。

```
int globx;
...
int sub(int a) {
  int b = 10;
  ...
  if(...) {
    ...
    int a = 20;
    ...
  }
  return a;
}
```

グローバル変数 globx の存在期間はプログラムの実行開始から終了までずっとです。いつでも使える変数なので当然ですね。これに対し、関数 sub のパラメタ a の存在期間は sub の実行開始から終了までの間です。また、その中の局所変数 b についても (先頭で定義していますから) 同じです。ということは、これらは sub が 2 回呼び出されたとすれば、2 回「存在を開始し」「存在を終了する」ことになります。ですから、2 回目に b に前回の値が残っていることは期待できません。if の内側の a はどうでしょう。これはブロックの途中で宣言されているので、「その箇所を実行した時点からブロックを出るまで」が存在期間です。

次に可視範囲ですが、globx は (ファイルの先頭で定義したとして) プログラムのどこからでもアクセスできますから、プログラム全体が可視範囲です。ただし、もし sub のパラメタ a か局所変数 b の名前が globx だったとしたら?! そのときは、sub の範囲内で globx と書くとそちらを意味しますから、sub の範囲内は可視範囲外になります。これを隠蔽ないし**シャドウ** (shadow) する、と言います。同様に、if の中の a の定義箇所以降ではパラメタ a はシャドウされています。ところが同じブロック内でも局所変数 a の定義より手前ではまだその局所変数は存在していないので、シャドウはなく、パラメタ a がアクセスできます。このシャドウの規則は整合性はあるのですが、実際にこういうプログラムを書くと勘違いを犯しやすいので、シャドウはできるだけ避けるのがよいでしょう。

2.3 アドレス型と入力 exam

C 言語の特徴的な機能の 1 つに、変数の**アドレス** (address、メモリ上の番地) を取得する機能があります。変数の場所を「指す」ことからアドレス値のことを**ポインタ** (pointer) とも呼びます。ポインタを扱うためには、次の 2 つの演算子を使います。

- $\&x$ — 変数 x のアドレスを取得して返す
- $*p$ — アドレス値 p にある変数をアクセス (**参照たどり**、dereference)

実際には変数には int 型のもの、double 型のものなど色々ありますから、ポインタ値も「int 型の変数を指すポインタ」など指す先の型で区分しないといけません。変数宣言において、変数名の直前に「*」を付けることでその変数はポインタ型であることを示します。例を見てみましょう。

```
// ptr1 --- pointer demonstration
#include <stdio.h>

int main(void) {
  int x = 10, *p;
  p = &x;
  printf("x = %d, %d, %d\n", x, ++(*p), x);
  return 0;
}
```

引数を使わないので main を引数無しとします。また atoi 等も使わないので stdlib.h も不要です。変数 x には 10 を入れ、p は「整数を指すポインタ」と宣言します。次の行で、x のアドレスを p に入れます。次は…3 つ数値を出力しますが、それは「x の値と、p の指す先の値を 1 増やした結果と、x の値」です。p は x を指しているので、(*p) は x と同じはずです。結果を見ましょう。

```
% ./a.out
x = 11, 11, 10
```

予想外でしたか? この処理形では引数を最後から順に計算するようです。C 言語の標準ではこのように引数の順序に依存するプログラムの結果は「不定」ということになっていますが、内実を伺ってみるのには面白い題材ですよね。

ポインタを使うと「変数の値をよそから書き換えられる」ので、C ではこれを入力に利用しています。具体的には「scanf("%d", アドレス);」で整数 (int)、「scanf("%lf", アドレス);」で実数 (double) の読み込みが行えます (もちろん正しい型の変数のアドレスを渡す)。やってみましょう。

```
// scanf1 --- scanf demonstration
#include <stdio.h>

int main(void) {
  int i; double x;
  printf("i? "); scanf("%d", &i);
  printf("x? "); scanf("%lf", &x);
  printf("i = %d, x = %f, x+i = %f\n", i, x, i+x);
  return 0;
}
```

実際に動かしてみましょう。

```
% ./a.out
i? 5
x? 3.1416
i = 5, x = 3.141600, x+i = 8.141600
```

しかしなぜ書式が"%lf"なのでしょう? それは、変数に複数のビット数のものがある関係です。たとえば sol の環境で、int は 32 ビットですが、もっとビット数が必要な場合は long という整数型も使えます (64 ビットになります)。これの入力や出力は"%ld"という書式指定を使います (long の l)。次に実数 double は 64 ビットですが、メモリを節約したい場合用にビット数の少ない実数型である float も使えます (32 ビットになります)。その入力に"%f"を使うので、それより長い double は"%lf"なのです。だったら出力は? と言われるでしょうけれど、C ではパラメタに渡す実数は標準で double を使うので、float の式を渡しても double に変換して渡されます。なので出力はどちらも"%f"なのです。

2.4 型変換とキャスト exam

変換 (conversion) という言葉が出てきました。Ruby でも整数と実数を混ぜて計算すると実数に自動変換されていましたが、C でも同様に整数は実数に自動変換が行われます。さらに上に述べたように、複数のビット数のものが混ざった場合は長いビット数に合わせられます。

では逆はどうでしょう? Ruby では実数を整数にするのに .to_i を使っていましたが、C ではメソッドではなく**キャスト** (cast) と呼ばれる専用の構文があります。その形は次の通りです。

(型名) 式

では例題を見てみましょう。

```
// cast1 --- cast demonstration
#include <stdio.h>

int main(void) {
  double x, y = 3.1416;
  printf("x? "); scanf("%lf", &x);
  printf("%f cast to int => %d\n", x, (int)x);
  printf("address of x, y => %lx, %lx\n", (long)&x, (long)&y);
  double *p =(double*)((long)&x - 8);
  (*p) = 2.71828;
  printf("y = %f\n", y);
  return 0;
}
```

実行結果を示します。

```
% ./a.out
x? 3.1416
3.141600 cast to int => 3
address of x, y => 7ffe0923c7b0, 7ffe0923c7a8
y = 2.718280
```

まず実数 x を読み込み、整数にキャストして出力します。次に、変数 x と y のアドレスを取得し、long にキャストして 16 進で出力します (sol ではアドレスは 64 ビットあるので long を使用する必要があります)。次に、double を指すポインタ変数 p を用意し、そこには「x のアドレスを long に変換し、8 引いて、double のポインタに再度変換して格納します。そして p の指す先を 2.71828 にすると、確かに y がその値になっています (最初 8 足していましたが、やってみると後に書いた変数の方がアドレスが小さいので修正しました)。

ただし、このようにアドレスを整数型に変換して計算というのは C 言語としては保証されていませんし、計算を間違って不正なアドレスをアクセスすると「Bus Error」「Segmententation Fault」などのエラーで強制終了されますから、注意してください (でも別に他人に迷惑を掛けることはないので、色々試すことは問題ないです)。

あと、型名の指定がよく分からなかったかも知れません。「p が整数を指すポインタ型」のとき、その宣言は「int *p」ですね。C では、その具体的な変数名 p を取り除いたもの、つまり「int*」が「整数を指すポインタ型」を意味する、という形で型名を指定するようになっています。

演習 2 上に出て来た 3 つの例題のうち好きなものを打ち込んで動かし、動作を確認しなさい。OK なら、以下のプログラムを作ってみなさい。

a. 実数値を次々に入力して、最後に 0.0 を入力すると合計を表示して終了するプログラム。
b. 最初にデータの個数を入力して、次にその個数だけ実数データを入力すると合計を表示して終了するプログラム。
c. 整数型の 2 つの変数の内容を交換する関数 iswap。もちろん 2 つの変数のアドレスを渡すことが想定されます。実際に動作していることを確認するコードも含むこと。
d. double 内部のビット表現がどうなっているか探究するプログラム。(ヒント: dobule を直接 long にキャストすると整数に変換されてしまうので、double 型の変数のアドレスを取得して long*にキャストし、そこから long 値を取り出し"%lx"でビット表示するとよい。)
e. グローバル変数やローカル変数、そして同じローカル変数でもブロックの中の変数がメモリ上でどのように配置されるかを探究するプログラム。とくに main → sub1 と呼んだ時と main → sub2 → sub1 と呼んだ時の違いに注意すること。

f. その他ポインタやアドレスを使った面白いと思うプログラム。

2.5 配列型とポインタ演算 exam

必要な概念が揃ったので、ここでようやく配列の説明をします。C では配列の宣言は「変数名 [用素数] で行います。そして添字を指定したアクセスのしかたは Ruby と同じです。例を見てみましょう。

```
// array1 --- array demonstration
#include <stdio.h>

int main(void) {
  int a[10] = {1,2,3,4,5,6,7,8,9,10};
  for(int i = 0; i < 10; ++i) {
    printf(" %d", a[i]);
  }
  printf("\n");
  return 0;
}
```

なお、配列 a の大きさは右辺の値の数から分かるので「a[] = ... でも OK です。では実行のようすです (for の初期化に宣言があるので-std=c99 指定必要)。

```
% gcc array1.c -std=c99
% ./a.out
 1 2 3 4 5 6 7 8 9 10
%
```

Ruby と全然おんなじで問題ないと思ったかも知れませんが、そうではないのです。C では基本的に、変数は存在範囲に入ったところで領域ができますが、上の中かっこで囲んだリストはそのときの「初期化」のための値です。ですから、実行の途中で好きな配列値を入れ直す等はできません。[2]

第 2 に、a[i] という**添字**つきの式が問題です。C では「T *p;」であるようなポインタ値に対して p[i](i は整数の式) という書き方ができ、その意味は*(p + i) と等しい、と定めています。

そして p + i とは? これは**ポインタ演算**と呼ばれ、p の指している要素から i 個ぶん先 (マイナスなら手前) の要素のアドレスとなります。1 つの要素のサイズは型 T によって違いますから、アドレスでいうと p に sizeof(T)*i を加えた値になります。ちなみに sizeof(T) は T 型の変数の占めるバイト数で、sol の環境では sizeof(int) == 4、sizeof(double) == 8 等となります。

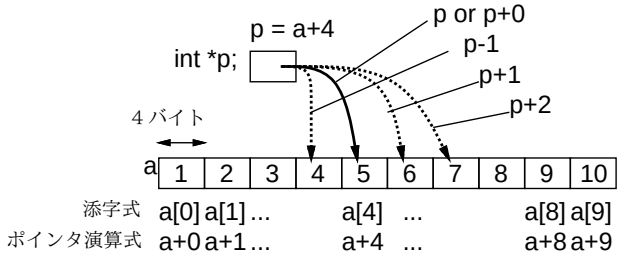

図 2: ポインタ演算と添字

そして、上の例での a のような配列名はその要素型のポインタ型であり、先頭要素のアドレスを表す、ということになっています。ですから、図 2 のように変数 p に a の途中を指させることでマイナスの添字を使ったりもできます。例を見ましょう。

[2] C99 では配列リテラルの機能が加わったのですが、そのリテラルの存在範囲も関数内なので結局あまり便利ではありませんし、C89 までと互換性がないのでここで説明している「初期化」のみ扱います。

```
// array2 --- pointer calc demonstration
#include <stdio.h>

int main(void) {
  int a[10] = {1,2,3,4,5,6,7,8,9,10};
  int *p = a + 4;                 // 5のところを指す
  for(int i = 5; i > -5; --i) {  // マイナスでもOK
    printf(" %d", p[i]);          // 10 9 ... 3 2 1
  }
  printf("\n");
  return 0;
}
```

実行例は略しますが、ちょっと面白いですね。しかし配列アクセスをこのように定義するということは、添字式が配列の範囲に入っていることをチェックすることを非常に難しくします (上の例でも a なら正しい添字の範囲は 0〜9 ですが p ならそれが-4〜5 になるわけです)。そういうわけで、C では配列の添字計算が間違っているとすぐに「Bus Error」「Segmentation Fault」といって実行が止まってしまい、どこが悪いか分からない、という事態になりやすいのです。

場所が分からない、では困るのでそれを調べる方法も書いておきます。まず、コンパイル時に-g オプションを指定します。次に直接実行するのでなく、**gdb** コマンド (デバッガ) に a.out を読ませて起動し、「run」コマンドで開始します。デバッガでは実行が中断するとデバッグ用コマンドを受け付けるので「bt」(backtrace) を指定してどこから呼ばれたどの位置という情報を表示します。たとえば、上の例で「--」を「++」にしてやってみます。

```
% gcc -g array2.c -std=c99      ← -g つけてコンパイル
% gdb a.out                     ← gdb を起動
...
(dgb) run                       ← 実行開始
.... 大量の出力 ...
0x000000000004005e7 in main () at array2.c: 8
8         printf(" %d", p[i]);  ← おかしい行が表示
(gdb) bt                        ← バックトレース表示
#0 0x000000000004005e7 in  main () at array2.c: 8
(dgb) q                         ← 終了
...
Quit anyway? (y or n) y         ← 終了して OK と返答
%
```

3 動的計画法

3.1 動的計画法とは

先にフィボナッチ数の計算を取り上げた時、次のような再帰的定義を示し、それをそのまま再帰関数にしたのでは遅すぎる、という説明をしました。遅すぎる理由は、この定義どおりだと 1 段階の再帰ごとに自分自身を 2 回呼び出し、同じパラメタに対する値を何回も重複して実行するからです。

$$fib(n) = \begin{cases} 1 & (n = 0 \; or \; n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

遅くなる理由である再計算を防ぐために、たとえば配列 `fib[`i`]` を用意して一度計算した値はそこに蓄え、2 回目からは計算しないでそれを持ってくる、という方法があります。一般に、関数を計算する時に、一度計算した結果を引数と一緒に覚えておいて、同じ引数に対しては覚えておいた値を返すようにすることを**メモ化** (memoization) と呼びます。

しかしそもそも、配列を使うのだったら、いちいち計算する代わりに、最大 30 番目までのフィボナッチ数だったら最初に順番に計算してしまい、それを参照するだけの方が分かりやすいはずです:

```
int fib[31] = { 1, 1 };
for(int i = 2; i <= 30; ++i) { fib[i] = fib[i-1]+fib[i-2] }
```

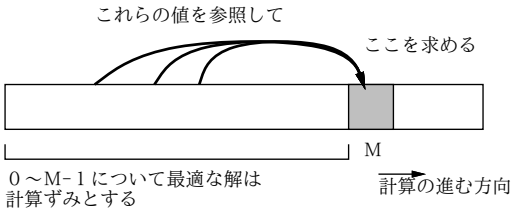

図 3: 動的計画法の考え方

これは図 3 のように、「値 $0 \sim M-1$ までについて解が求まっていれば値 M についての解もすぐ求まる」問題に対し、配列を用いて 0 から順に答えを埋めていくことで値 M に対する答えを求めていると言えます。一般にある問題に対して、その問題だけを解く代わりに小さい問題から順に全ての問題を答えを記録しつつ解くことで必要な解を求める手法のことを、**動的計画法** (dynamic programming, DP) と呼びます。なお、これは単なる 1 つの手法であり、特別に動的でも特別にプログラミングでも何でもありません。この手法を考案した人がそういう名前をつけた、というだけです。他の方法では計算量が多すぎて扱えない問題が動的計画法によって効率よく扱える場合も多くあります。

3.2 部屋割り問題

動的計画法の適用例として、次のような問題を考えてみます。

> 合宿で 1 泊料金が「1 人部屋:5,000 円、3 人部屋:12,000 円、7 人部屋:20,000 円」というホテルに泊まる。[3] 合計宿泊人数 n 人に対し、最も安い宿泊金額総計を求めよ。

この問題では、7 人部屋が非常に割安なので、7 人より少ない人数で泊まっても 7 人部屋を選んだほうがよい場合があり、最適な割り当てを求めるのは簡単ではありません。この問題に限って言えば、できるだけ多く 7 人部屋を使って、残った 1〜6 人の場合について全部の場合を検討すれば済みますが、17 人部屋とか 31 人部屋とかもあったとすると大変すぎます。

そこで動的計画法を用いる準備として、人数 n に対して最も安い値段を計算する関数 $roomprice$ を次のように定義します。

$$roomprice(n) = minimumof \begin{cases} roomprice(n-1) + 5000 \\ roomprice(n-3) + 12000 \\ roomprice(n-7) + 20000 \end{cases}$$

$minimumof$ というのは聞いたことがないと思いますが (今発明したものなので当然です)、右側の選択肢のうち一番小さい値を取る、という意味のつもりです。なお、$roomprice(n)$ は $n \leq 0$ のときは 0 であるものとします (泊まる人数が 0 以下ならお金は掛かりませんから)。

[3] 参加者は全員同性とし、各部屋には収容人数より少ない人数でも泊まれます。どの部屋も数は十分あるものとします。

なぜこれでいいかというと、n 人で泊まる時の最も安い方法は、「$n-1$ 人で泊まる時の最も安い場合に 1 人部屋を追加する」「$n-3$ 人で泊まる時の最も安い場合に 3 人部屋を追加する」「$n-7$ 人で泊まる時の最も安い場合に 7 人部屋を追加する」のうちのどれかではあるに決まっているからです (どれであるかは計算してみないと分かりませんが)。

では、これを C プログラムにしてみましょう。大きさ RMAX の配列 roomprice を作り、1〜$RMAX-1$ までの i について順次、3 つの場合の最小値を求めて入れて行きます。

```
// rooms1 --- room pricing with DP
#include <stdio.h>
#include <stdlib.h>
#define RMAX 1000

int roomprice[RMAX] = { 0, };
void initialize(void);
int room1(int i);

int main(int argc, char *argv[]) {
  initialize();
  for(int i = 1; i < argc; ++i) {
    int n = atoi(argv[i]);
    printf("room price for %d => %d\n", n, roomprice[n]);
  }
  return 0;
}
int room1(int i) {
  return (i < 0) ? 0 : roomprice[i];
}

void initialize(void) {
  for(int i = 1; i < RMAX; ++i) {
    int min = room1(i-1) + 5000;
    if(min > room1(i-3) + 12000) { min = room1(i-3) + 12000; }
    if(min > room1(i-7) + 20000) { min = room1(i-7) + 20000; }
    roomprice[i] = min;
  }
}
```

こんどはまたコマンド引数を使うようになっていますが、main は調べたい人数を並べていちどに受け取るようにしています。まず最初に initialize を呼び、この中で配列 roomprice の初期化を行います。そして i を 1〜argc-1 まで順に変化させながら、argv[i] の文字列を atoi で整数にして変数 n に入れ、その人数のときのコストを表示します。

initialize では先のアルゴリズムによって roomprice を順に初期化していますが、値を読み出す時には room1 という下請け関数を呼びます。この関数は、人数 i が負のときは 0 を返し、それ以外は roomprice[i] を返します。そうしないと配列の負の場所を参照してしまいますから。return の後ろにあるのは C 言語の 3 項演算子「条件 ? 式 : 式」で、機能は Ruby の if-then-else 式と同じです。では動かしてみましょう。

```
% ./a.out 10 11 12      ←調べたい人数をならべて指定
room price for 10 => 32000
```

```
room price for 11 => 37000
room price for 12 => 40000
%
```

なるほど、12 人だと 7 人部屋が 2 つの方が安いわけです。

しかし、何人部屋がいくつなのかも知りたいですよね? そのためには、次の定義による値 $roomsel(n)$ も一緒に計算すればよいのです。

$$roomsel(n) = \begin{cases} 1 & (roomprice(n-1) + 5000 \ is \ the \ smallest) \\ 3 & (roomprice(n-3) + 12000 \ is \ the \ smallest) \\ 7 & (roomprice(n-7) + 20000 \ is \ the \ smallest) \end{cases}$$

この関数は、n 人のときに「最後に選んだ最適な部屋人数」を返します。選んだ部屋のリストを得るには、たとえば $roomsel(10)$ が 3 だったら、さらに $roomsel(7)$ を調べ、というふうに次々に「逆向きに」たどって行く必要があります。このため、こちらの情報のことを「トレースバック情報」と呼びます。では、先のメソッドを改造してトレースバックを記録し、金額に続いて部屋のリストを (1 つの配列として) 並べて返すようにしてみます。

```
// rooms2 --- room pricing with DP w/ traceback
#include <stdio.h>
#include <stdlib.h>
#define RMAX 1000

int roomprice[RMAX] = { 0, };
int roomsel[RMAX] = { 0, };
void initialize(void);
int room1(int i);

int main(int argc, char *argv[]) {
  initialize();
  for(int i = 1; i < argc; ++i) {
    int n = atoi(argv[i]);
    printf("room price for %d => %d;", n, roomprice[n]);
    while(n > 0) { printf(" %d", roomsel[n]); n -= roomsel[n]; }
    printf("\n");
  }
  return 0;
}
int room1(int i) {
  return (i < 0) ? 0 : roomprice[i];
}

void initialize(void) {
  for(int i = 1; i < RMAX; ++i) {
    int min = room1(i-1) + 5000, sel = 1;
    if(min > room1(i-3) + 12000) { min = room1(i-3) + 12000; sel = 3; }
    if(min > room1(i-7) + 20000) { min = room1(i-7) + 20000; sel = 7; }
    roomprice[i] = min; roomsel[i] = sel;
  }
}
```

これを動かすと、今度はちゃんと部屋の選択が分かります。

```
% ./a.out 10 11 12
room price for 10 => 32000; 3 7
room price for 11 => 37000; 1 3 7
room price for 12 => 40000; 7 7
%
```

演習 3 上の例題を打ち込んでそのまま動かしなさい (最初はトレースバック無しの簡単な方を動かし、動いてからトレースバックを追加した方が楽だと思います)。動いたら、「13 人部屋 3 万円」「17 人部屋 4 万円」の選択肢を追加して動かしてみなさい。

演習 4 「釣り銭問題」も動的計画法が使える典型的な問題です。たとえば、米国だとコインの額面が「1 ¢」「5 ¢」「10 ¢」「25 ¢」の 4 種類なので、ある金額 (¢) を与えられたとき「何枚」コインがあれば済むかを決めるのはちょっと面倒です。これも、次のように考えると動的計画法で解けます (ただし $coins(0) = 0$ と定義します)。

$$coins(c) = minimumof \begin{cases} coins(c-1)+1 & (c \geq 1) \\ coins(c-5)+1 & (c \geq 5) \\ coins(c-10)+1 & (c \geq 10) \\ coins(c-25)+1 & (c \geq 25) \end{cases}$$

これに基づき、¢ 金額を与えたとき最小コイン枚数を答えるプログラムを作りなさい。できればさらにトレースバックを追加して、具体的なコインの組み合わせも答えるようにしなさい。

演習 5 整数の列が与えられた時、その中から (とびとびに) 後の方ほど値が大きくなるような部分列を選ぶとする。そのような部分列で最も長いもの (最長増加部分列、longest increasing subsequence) の長さ (できれば列自体も) を表示するプログラムを書きなさい。たとえば列が「1, 5, 7, 2, 6, 3, 4, 9」であれば、長さ「5」、列としては「1、2、3、4、9」が解となる。

ヒント: 動的計画法の場合配列を用意し、その i 番には i 番の値が最後にある部分列の最大長を入れていきます。列自体を表示するにはトレースバックが必要になります。

演習 6 動的計画法を使って何か面白いと思うプログラムを作りなさい。

本日の課題 12A

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語の入出力機能についてどのように思いましたか。

Q2. C 言語の型変換機能についてどのように思いましたか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 12B

「演習 1」～「演習 6」の (小) 課題から 1 つ以上を選択してプログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語の配列機能についてどう思いましたか。

Q2. 動的計画法を理解しましたか。またどのように思いましたか。

Q3. 課題に対する感想と今後の要望をお書きください。