プログラミング通論'19 ＃6 – 分割統治と再帰の除去

久野 靖 (電気通信大学)

2019.4.20

今回は次のことが目標となります。

- 分割統治アルゴリズムの考え方について知る。
- 再帰呼び出しを持つプログラムから再帰を除去する手法について知る。

1　分割統治アルゴリズム

1.1　分割統治アルゴリズムとは

前回から再帰を用いたコードを扱っていますが、再帰アルゴリズムの作り方には多様なものがあります。ここでは**分割統治** (divide and conquer) と呼ばれる手法について取り上げます。[1] 分割統治アルゴリズムの考え方は次のようなものです。

- 与えられた問題に直接取り組む代わりに、問題を 2 つとかさらに多くの「部分問題」に分割する。
- それぞれの部分問題は自分自身を再帰呼び出しして解く。
- その解いた結果を組み合わせることで元の問題の解を得る。

たとえばハノイの塔の問題を振り返ってみましょう (図 1)。元の問題は K 段の塔を (1 つずつ円盤を動かしながら)A から B にそっくり移動するというものですが、結構複雑です。

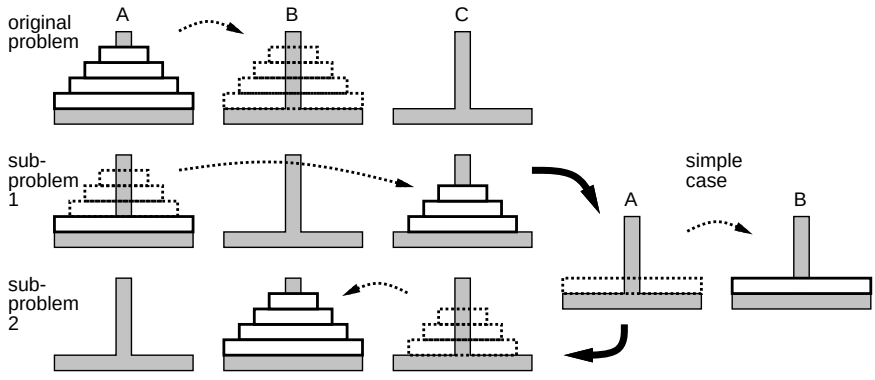

図 1: 分割統治手法としてのハノイの塔

そこで自分は円盤 K だけ担当することにして、1〜K-1 をまず C に移し (※)、その後上に何もなくなった K を A から B に移し、最後に C に移してあった 1〜K を B に移す (※) ことにするわけです。ここで※印の 2 つが分割された部分問題にあたります。そして全体の解を組み立てるとき、その 2 つと間の「K を A から B に移す」という簡単な作業を合わせればよいわけです。「部分問題に分割」の意味がお分かりになったでしょうか。

[1] 分割統治という言葉は、ローマ帝国が各地を統治するときに「統治される側をばらばらに分断したりその上で互いに反目させるなどしてうまく御した」という故事から来ていますが、ここではその意味からこの言葉を援用しています。

1.2 再帰による空間分割

分割のしかたは問題により様々です。次の例として、長方形 (や正方形) の領域内で曲線や直線で囲まれた図形の面積を求める問題を取り上げましょう。たとえば、2x2 の領域に図 2 のように四分の一円が配置されていたとして、その面積を求めれば π になるはずですね。

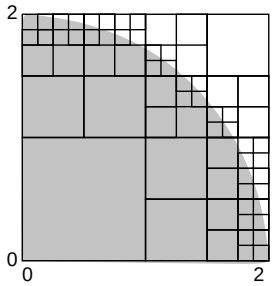

図 2: 空間分割により面積を求める

ただし、その問題は簡単ではありません。そこで、自分の領域を 4 分割してそれぞれについて自分自身を再帰呼び出しして、それぞれの領域で図形に囲まれた部分の面積を求めます。これが部分問題です。部分問題から自分の問題の解を作り出すのは簡単で、面積ですから単に足せばよいわけです。

しかしそれでは、どんどん領域が細かくなるだけで止まらなくなりますね? そこで (1) 領域の 4 隅がともに図形に含まれている/いないなら、領域全体の面積ないし 0 を返します。あるいはそうでない場合でも、(2) 分割が十分細かければ…たとえば領域の幅が δ 未満であれば、適当に近似した値を返します。これが「簡単な場合」ですね。このように、2 次元や 3 次元の領域を必要に応じて荒く/細かく分割して処理してゆく手法を**空間分割**と呼びます。

ではプログラムを見てみましょう。最初の関数 `inside_ciecle` は、渡された XY 座標が円の内部にあれば 1、無ければ 0 を返すものとします (それなら `bool` でもよさそうですが、結果を足し算したいので `int` としています)。そして面積を計算する関数 `area` ですが、パラメタとして X の最小と最大、Y の最小と最大を受け取ります。この長方形 (または正方形) の領域内で、さらに渡された図形の中にある面積を求めるわけです。

ですが、最後の「`int (*f)(double,double)`」とは? これは関数ポインタ型で、「実数を 2 つ受け取り整数を返す」関数へのポインタを受け取ります。

```
// area.c -- calcurate area of 2-D object
#include <stdio.h>
#define DELTA 0.001

int inside_circle(double x, double y) {
  return x*x+y*y <= 4.0;
}
double area(double x1, double x2, double y1, double y2,
            int (*f)(double,double)) {
 int b1 = (*f)(x1,y1), b2 = (*f)(x1,y2);
 int b3 = (*f)(x2,y1), b4 = (*f)(x2,y2), bn = b1+b2+b3+b4;
 if(x2-x1 < DELTA || bn == 0 || bn == 4) {
   return bn*0.25*(x2-x1)*(y2-y1);
 } else {
   double x3 = 0.5*(x1+x2), y3 = 0.5*(y1+y2);
   return area(x1, x3, y1, y3, f) + area(x1, x3, y3, y2, f) +
          area(x3, x2, y1, y3, f) + area(x3, x2, y3, y2, f);
```

```
  }
}
int main(void) {
  printf("%8.6f\n", area(0, 2, 0, 2, inside_circle));
  return 0;
}
```

そして area に入ってすぐのところで、領域の 4 隅について渡された関数を呼び出し、図形の内部か否かの 1/0 を求めます。その合計が 4 または 0 なら「完全に入っている/いない」単純な場合です。または、X 方向の幅が δ 未満のときも、これ以上細かく分けずに単純な場合として処理します。具体的に図形内部に入っている頂点の数が bn だとしたとき、領域の面積を s として $s \times \frac{bn}{4}$ で面積を近似します (全て入っている/いない場合もこれで対応できます)。一方、単純でない場合は X および Y の範囲の中間を求めて領域を 4 分割し、それぞれの領域内での面積を再帰で求めてから合計します。

main は簡単で、領域の範囲と inside_ciecle を渡して area を呼び出し、結果を表示するだけです。実行のようすを見ましょう。小数点以下 4 桁まで正しく求まっているようです。

```
% ./a.out
3.141577
```

演習 1 面積の計算を実際に動かして確認しなさい。OK なら次のことをやってみなさい。

a. もっと別の図形で面積を計算してみる。
b. DLETA の値を変化させることで、どれくらい精度が変わるか、試して検討する。できれば複数の図形の面積で試してみるとなおよい。
c. 実際にいくつの長方形 (または正方形) を計算しているのか、それは DELTA の値を変化させるとどのように変わるのかを試してみて検討する。
d. 上の例は 2 次元だったが、3 次元でも同じように考えて体積を空間分割で求めることができる。作成してみよ (たとえば八分の一球の体積を求めてみるなど)。
e. 空間分割を使った面白いプログラムを何か作ってみよ。

2 再帰の除去

2.1 再帰の除去とその必要性

再帰はさまざまな計算をコンパクトに書けるようにする有力な手法ですが、一方で再帰の数が多くなると実行時スタック領域を多く消費するという弱点も持っています。また、言語や環境によっては再帰呼び出しが使えなかったり使いづらいこともあります。

一般に、再帰を使って書かれたプログラムはすべて、再帰を使わないように書き換えることが可能です。これを**再帰の除去** (resursion elimination) と呼びます。除去のやりやすさは、プログラムの形によって違っています。以下で主要な手法について見て行きます。

2.2 末尾再帰の除去

末尾再帰 (tail recursion) とは、再帰呼び出しの形が「`return` 再帰呼び出し`;`」となっているもの、すなわち再帰呼び出しで返された値がそのまま自分の返値になるようなものを言います。たとえば前に取り上げた GCD の関数を見てみましょう。

```
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);
  }
}
```

見て分かるように、2 箇所ある再帰呼び出しはいずれも上記の形すなわち末尾再帰になっています。なぜ末尾再帰を問題にしているかというと、末尾再帰では再帰を呼ぶところでもう呼び側の関数のフレームは不要になっているからです (戻って来たら直ちに return するのでそれ以上変数やパラメタなどを使う余地がない)。ということは、再帰呼び出しをして新しいフレームを作ったりしなくても、単にパラメタを渡そうとしている実引数の値で書き換えて先頭に戻るだけでよいのです。実際に上のコードをそのように変更してみましょう。

```
int gcd2(int x, int y) {
  while(true) {
    if(x == y) {
      return x;
    } else if(x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
}
```

「先頭に戻る」ことを実現するため、本体全体を無限ループで囲みました。そして、再帰呼び出しの箇所ではパラメタ x や y に渡そうとしている式の値を代入しています (このコードでは片方は元の値のままなので何もしなくて済みます)。

このように末尾再帰をループに置き換えることは、呼び/戻りの手間も余分なスタックフレームの消費も削減でき、利点だらけです。そのため、言語処理系によっては、このような変形を自動的にやってくれます (残念ながら C コンパイラではほとんどないですが)。

演習 2 上の再帰除去版の GCD を動かして動作を確認しなさい。OK なら次の末尾再帰のみから成る関数でも同様に再帰除去してみなさい。元の版と両方動かして検討すること。

a. 非負整数を受け取り偶数か否かを返す関数 iseven。

```
bool iseven(int n) {
  if(n == 0) {
    return true;
  } else if(n == 1) {
    return false;
  } else {
    return iseven(n - 2);
  }
}
```

b. 2 つの非負整数を受け取りその和を返す関数 sum。

```
int sum(int a, int b) {
```

```
  if(a == 0) {
    return b;
  } else {
    return sum(a-1, b+1);
  }
}
```

c. 文字列を「abc」「bc」「c」「」のように 1 文字ずつ削りながら打ち出す関数 strtriangle。この例では打ち出すのが目的なので返値がなく return がないが、関数の末尾に来たらそこで return することになるのでこれまでと同様に考えられる。

```
void strtriangle(char *s) {
  printf("%s\n", s);
  if(*s == '\0') {
    // do nothing
  } else {
    strtriangle(s+1);
  }
}
```

d. その他自分の好きな末尾再帰のみから成る関数。

2.3 末尾再帰への変形

末尾再帰の除去は分かりましたが、現実には末尾再帰でない再帰呼び出しも多くあります。それらのうちで、自分自身を 1 回しか呼ばない再帰は末尾再帰に変形できます。たとえば階乗を見てみます。

```
int fact(int n) {
  if(n < 1) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

これは再帰呼び出しから返された値にさらに n を掛けて自分の値とするので、「そのままの値を返す」末尾再帰にはなっていません。しかしこれを、次のように変形したらどうでしょうか。

```
int fact1(int n, int r) {
  if(n < 1) {
    return r;
  } else {
    return fact1(n-1, r*n);
  }
}
int fact(int n) { return fact1(n, 1); }
```

再帰関数そのものは 1 つパラメタを追加し、呼び出し方が変わるのでそれを呼び出すための fact を別に用意しました。追加したパラメタ r は「最終結果を累積していくための」パラメタで、**累積引数** (accumulation parameter) と呼びます。

たとえば 5 の階乗であれば fact(5) → fact1(5, 1) → fact1(4, 5) → fact1(3, 20) → fact1(2, 60) → fact1(1, 120) → fact1(0, 120) → 120、のように累積引数を使って結果が計算されていきます。一般

に累積引数は、適切な初期値 (変換前の関数で単純なケースの値) を与えて呼び出し、そこに加算や乗算を行って最終結果ができあがり、最後に単純なケースでその値を返します。これで末尾再帰のみに変形できたので、あとは前と同じようにして再帰を除去できます。

演習 3 階乗の例題をひととおり動かして確認しなさい。OK なら、以下の末尾再帰でない 1 次元の (=自分を 1 回しか呼ばない) 再帰を累積引数を使って末尾再帰に変形し、さらに再帰を除去してみなさい。元の関数と変換した関数の両方を実行し確認すること。

a. 2 つの正の整数を受け取りその積を返す関数 `mul`。

```
int mul(int a, int b) {
  if(b == 0) {
    return 0;
  } else {
    return a + mul(a, b-1);
  }
}
```

b. 実数 x と非負整数 n を受け取り x^b を返す関数 `powx`。

```
int powx(double x, int n) {
  if(n < 1) {
    return 1.0;
  } else {
    return x * powx(x, n-1);
  }
}
```

c. 実数 x と非負整数 n を受け取り x^b を返す関数 `powx`(高速版)。

```
double powx(double x, int n) {
  if(n < 1) {
    return 1.0;
  } else if(n % 2 == 1) {
    return x * powx(x, n-1);
  } else {
    double y = powx(x, n / 2); return y*y;
  }
```

d. 実数 x と非負整数 n を受け取り $\sum_{i=0}^{n} \frac{1}{x+i}$ を返す関数 `calc`。

```
int calc(double x, int n) {
  if(n < 0) {
    return 0.0;
  } else {
    return 1/(x+i) + calc(x, n-1);
  }
}
```

e. その他自分の好きな 1 次元再帰の関数。

2.4 スタックを使ったコードへの書き換え

末尾再帰 (や末尾再帰への変形) により再帰が除去できるのは、1 次元の (1 つの呼び出しにつき自分を 1 回しか呼ばない) 再帰に限られます。そうでない場合はどうしたら良いのでしょうか。もともと言語処理系は再帰を実行時スタックを用いて実現しています。ですから、言語処理系の代わりに自分で「同じように」スタックを操作することで、再帰と同じ動作が「必ず」実現できます。とはいえ、あまり分かりやすくはないことが多いですが…

再びハノイの塔を取り上げましょう。再帰版のコードを再掲します。

```
void hanoi(int k, int x, int y, int z) { // (1)
  if(k == 1) {
    printf("move disc %d from %c to %c.\n", k, x, y);
  } else {
    hanoi(k-1, x, z, y); // (2)
    printf("move disc %d from %c to %c.\n", k, x, y);
    hanoi(k-1, z, y, x);
  }
}
```

(1)、(2) とコメントがついていますが、これは「どこから実行するか」が 2 通りあることに対応しています。そして、スタックには引数である k, x, y, z のほかにその「どこから実行」を区別する値 cont も積みます (ローカル変数があればそれも積むのですが、この例題ではローカル変数はありません)。上記の 5 つのフィールドを持つ構造体を要素とするスタックを使うのが自然ですが、ここでは既に作った istack を再利用することにして、5 つの値を逆順に積む下請け関数 p5 というのを作りました。[2] ではスタック版のコードを見てみましょう。

```
// hanoistack.c --- hanoi with stack
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"

void p5(istackp s, int a, int b, int c, int d, int e) {
  istack_push(s,e); istack_push(s,d);
  istack_push(s,c); istack_push(s,b); istack_push(s,a);
}
void hanoiloop(int k, int x, int y, int z) {
  istackp s = istack_new(100); p5(s, 1, k, x, y, z);
  while(!istack_isempty(s)) {
    int cont = istack_pop(s); k = istack_pop(s);
    x = istack_pop(s); y = istack_pop(s); z = istack_pop(s);
//  printf("%d %d %c %c %c\n", cont, k, x, y, z);
    if(cont == 1) {
      if(k == 1) {
        printf("move disk %d from %c to %c.\n", k, x, y);
      } else {
        p5(s, 2, k, x, y, z); p5(s, 1, k-1, x, z, y);
      }
```

[2] なぜ逆順かというと、スタックでは最後に積んだものが最初に出て来るため、降ろす時に自然な順番で取り出したいからです。

```
      } else { // cont == 2
        printf("move disk %d from %c to %c.\n", k, x, y);
        p5(s, 1, k-1, z, y, x);
      }
    }
  }
int main(int argc, char *argv[]) {
    hanoiloop(atoi(argv[1]), 'A', 'B', 'C'); return 0;
  }
```

hanoiloop がその変換した関数ですが、最初にスタックを用意し、実行箇所 1 と 4 つの引数を積みます。その後はスタックが空になるまで繰り返しです。

繰り返しの先頭で 5 つの値を取り出し変数に入れます (実行位置 cont 以外はパラメタの変数を再利用しています)。それから、実行位置によって 2 つに枝分かれします。まず (1) の方ですが、k が 1 のときはこれまで通り出力します。そうでないときが再帰呼び出しですが、再帰版を見るとまず hanoi(k-1, x, z, y) を読んで、戻って来たら出力ですね。変換後も同じに動作するためには、スタックの下にまず「戻って来たら (2) から実行」を積んで、それから「先頭から k-1, x, z, y のパラメタで実行」を積みます (後から積んだ方が先に出て来るので)。そして仕事はこれで終わりです。残りは (2) からの続きの実行ですが、それは出力したあと今度は「先頭から k-1, z, y, x」で実行ですね。

main はコマンド引数から円盤の数を取り出して hanoiloop を呼ぶだけです。実行のようすを示します (printf につけてあるコメントを外して毎回スタックから出て来る値を表示するようにしています)。図 3 にスタックの変化の様子を示しました。

```
% gcc8 hanoiloop.c istack.c
% ./a.out 3
1 3 A B C
1 2 A C B
1 1 A B C
move disk 1 from A to B.
2 2 A C B
move disk 2 from A to C.
1 1 B C A
move disk 1 from B to C.
2 3 A B C
move disk 3 from A to B.
1 2 C B A
1 1 C A B
move disk 1 from C to A.
2 2 C B A
move disk 2 from C to B.
1 1 A B C
move disk 1 from A to B.
```

2.5 返値がある場合のスタック変換

ハノイの塔では関数に返値がありませんでしたが、返値がある場合はどうしたらいいでしょうか。ここでは分かりやすさのため、返値を積むスタックを別に用意する方法で説明します。例題は組み合わせの数の再帰版です。

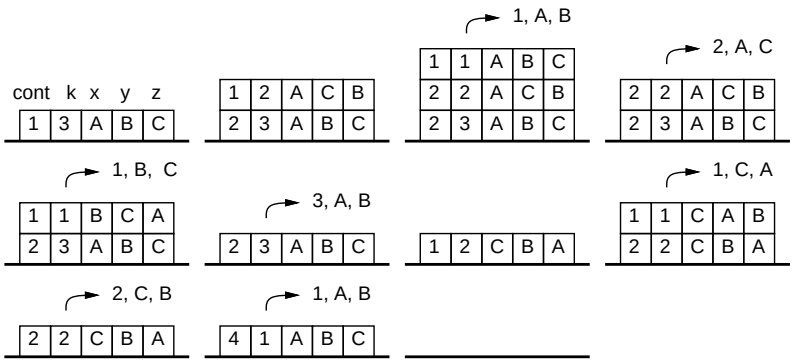

図 3: ハノイの塔のスタック版でのスタックの動作

```
int combr(int n, int r) { // (1)
  if(r == 0 || r == n) {
    return 1;
  } else {
    return combr(n-1, r) + combr(n-1, r-1); // (2)
  }
}
```

上述のように返値のスタック v を追加し、return ではそこに値を積むようにします。変数を積むスタックは実行位置も含め 3 つずつ値を積めばよいです。

```
// combstack.c --- combination with stack
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"

void p3(istackp s, int a, int b, int c) {
  istack_push(s,c); istack_push(s,b); istack_push(s,a);
}
int combloop(int n, int r) {
  int x, ret;
  istackp v = istack_new(100);
  istackp s = istack_new(100); p3(s, 1, n, r);
  while(!istack_isempty(s)) {
    int cont = istack_pop(s);
    n = istack_pop(s); r = istack_pop(s);
//  printf("%d %d %d\n", cont, n, r);
    if(cont == 1) {
      if(r == n || r == 0) {
        istack_push(v, 1);
      } else {
        p3(s, 2, 0, 0); p3(s, 1, n-1, r-1); p3(s, 1, n-1, r);
      }
    } else { // cont == 2
      istack_push(v, istack_pop(v)+istack_pop(v));
    }
```

```
    }
    return istack_pop(v);
  }
int main(int argc, char *argv[]) {
    printf("%d\n", combloop(atoi(argv[1]), atoi(argv[2]))); return 0;
  }
```

まず先頭 (1) から実行する場合ですが、単純なケースでは return だけですからスタック v に 1 を積むだけです。そうでない場合は、2 つの再帰を実行したあとでそれらの結果を足し算しますから、(2) からの実行を先に、2 つの再帰呼び出しぶんを後に積みます。再帰から戻った (2) ですが、スタック v から 2 つ値を取り出して足し、それを再度 v に積みます。そしてループが終わった時には結果がスタック v に載っているので、それを返します。

演習 4 hanoiloop と combloop を実行し、動作を確認しなさい。また両方についてそれぞれ、複数の (資料に載っているものとは別の) 実行例のスタックの変化の様子をまず自分で書き出し、次に実行してみて合っているか確認しなさい (printf につけているコメントを外して実行する)。

演習 5 次のような再帰を使った (返値を持たない) プログラムの再帰をスタックを用いて削除せよ。なお、再帰呼び出しごとに変化しないパラメタはスタックに載せないでよいことに注意。

a. 12 減少列のプログラム。参考までに再帰版をつける。

```
// decr12.c --- usage: ./a.out INTEGER
#include <stdio.h>
#include <stdlib.h>

void decr12(int n, int k, int *a) {
  if(n < 1) {
    // do nothing
  } else if(n == 1) {
    a[k] = 1;
    int i;
    for(i = 0; i <= k; ++i) { printf(" %d", a[i]); }
    printf("\n");
  } else {
    a[k] = n; decr12(n-1, k+1, a); decr12(n-2, k+1, a);
  }
}
int main(int argc, char *argv[]) {
  int buf[100]; decr12(atoi(argv[1]), 0, buf); return 0;
}
```

b. すべての文字の組み合わせ。参考までに再帰版をつける。

```
// allstr.c --- usage: ./a.out STRING LENGTH
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void allstr(int n, char *s, int len, int k, char *a) {
```

```
  if(len == 0) {
    a[k] = '\0'; printf("%s\n", a);
  } else {
    int i;
    for(i = 0; i < n; ++i) {
      a[k] = s[i]; allstr(n, s, len-1, k+1, a);
    }
  }
}
int main(int argc, char *argv[]) {
  char str[100];
  allstr(strlen(argv[1]), argv[1], atoi(argv[2]), 0, str);
  return 0;
}
```

c. 文字列のすべての順列。参考までに再帰版をつける。

```
// perm.c --- usage: ./a.out STRING
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void cswap(char *a, int i, int j) {
  char c = a[i]; a[i] = a[j]; a[j] = c;
}
void perm(int len, int k, char *a) {
  if(k == len) {
    printf("%s\n", a);
  } else {
    int i;
    for(i = k; i < len; ++i) {
      cswap(a, k, i); perm(len, k+1, a); cswap(a, k, i);
    }
  }
}
int main(int argc, char *argv[]) {
  char str[100];
  strcpy(str, argv[1]); perm(strlen(str), 0, str); return 0;
}
```

d. その他自分の好きな再帰プログラム。

演習 6 自分の好きな返値を持つ再帰プログラムのスタックを用いた再帰削除版を作成しなさい。

本日の課題 6A

「演習 1」〜「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 6a ファイル名`」で以下の内容を提出。

2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. 分割統治という概念について納得しましたか。
   Q2. 末尾再帰とはどういうものか理解しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 6B

「演習 1」〜「演習 6」(ただし 6A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:69 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 6b` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. 末尾再帰に対する再帰除去ができるようになりましたか。
   Q2. スタックを用いた再帰除去ができるようになりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。