プログラミング通論'19 ＃14 – 動的計画法

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。

- 再帰関数のメモ化と動的計画法の関係を理解する
- 2次元の動的計画法を使えるようになる

メモ化と動的計画法

再帰関数とメモ化

今回は以前やった再帰関数の中から、フィボナッチ数のコードを取り上げます。

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

このコードは再帰1段ごとに自分自身を2回呼ぶので、計算量が$O(2^n)$になってしまい、大変遅いです。しかしそもそも、遅い理由というのは、同じ引数に対する関数値を何回も重複して計算するためですよね(図1)。

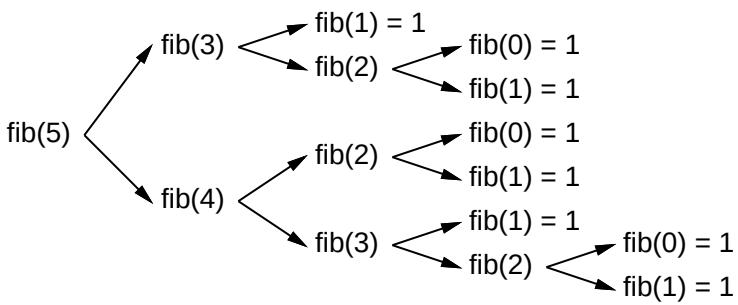


図 1: fib 再帰関数での計算の重複

再帰関数とメモ化 (2)

この関数では与えられた引数に対する結果は何回計算しても同じわけですから、「最初に計算したときにそこ結果を覚えておき」「2 回目からは覚えている結果をそのまま返す」ようにすれば、同じ引数に対する計算を何回も行わなくて済みます。このような処理を (結果をメモしておくことから) メモ化 (memoization) と呼びます。実際にやって見ましょう。

再帰関数とメモ化 (3)

```
// fibmemo.c --- calcurate fib with memoization.
#include <stdio.h>
#include <stdlib.h>
#define ARRSIZE 100
int fib(int n) {
  if(n <= 1) { return 1; }
  return fib(n-1) + fib(n-2);
}
int fib1(int n) {
  static int memo[ARRSIZE];
  if(memo[n] != 0) { return memo[n]; }
  int r = 1;
  if(n > 1) { r = fib1(n-1) + fib1(n-2); }
  memo[n] = r; return r;
}
int main(int argc, char *argv[]) {
```

```
  int n = atoi(argv[1]);
  printf("fib(%d) = %d\n", n, fib1(n));
  return 0;
}
```

再帰関数とメモ化 (4)

これまでのfibの方は比較用に入れてあります (時間を比較するときに利用してください)。メモ化を行った方はfib1です。その先頭でメモ用の配列を宣言していますが、この配列は値を保管するので「ずっと存在していてくれないと」困ります (プログラムの実行中全体が存在期間)。そのような場合はローカル変数でも冒頭にstatic をつけます。もちろん配列を関数の外に置いてもそうなりますが、この場合関数の中だけで使うのでこの方がお行儀がよいです。なお、C言語ではグローバル変数およびstatic変数は内容が0で初期化されることになっています。

次に、渡された引数nの位置のmemoを調べ、0でなければ既に計算した結果が入っているので、直ちにそれを返します。これがメモ化の「後半 (値を使う方)」になります。続いて、変数rに関数の値を計算しますが、これは直接returnで返してしまうとメモ配列に書き込むタイミングが無いためです。最後に、引数nの位置にrを書き込むのがメモ化の「前半 (値を入れる方)」で、そのあとそのrを返して終わります。実行例は当り前なので省略します。

再帰関数とメモ化 (5)

演習 1 例題を動かし、正しく計算されていることを確認しなさい。メモ化する前の版も動かし、結果と時間を比較すること。納得したら、次の関数のメモ化版を作り、元の版と比較しなさい (あくまでもここに示した遅い再帰関数をもとにメモ化すること)。なお、最後のもののように引数が 2 つある場合は、メモ配列も 2 次元配列にする必要があることに注意。

a. 3 の n 乗

```
int pow3(n) {
  if(n <= 0) { return 1; }
  return pow3(n-1) + pow3(n-1) + pow3(n-1);
}
```

再帰関数とメモ化 (6)

b. n の階乗

```
int fact(int n) {
  if(n <= 1) { return 1; }
  int r = 0;
  for(int i = 0; i < n; ++i) { r += fact(n-1); }
  return r;
}
```

c. ${}_nC_r$

```
int comb(int n, int r) {
  if(r == 0 || r == n) { return 1; }
  return comb(n, r-1) + comb(n-1, r-1);
}
```

メモ化から動的計画法へ

前節では再帰関数が呼ばれた時にそのパラメタに対する値を配列memoに保管していました。しかしfibの場合でいうと、ある値nに対する計算を行うためには結局$0 \sim n-1$すべてについて値を計算する必要があるので、最初からこれらを「小さい順に」計算してしまう方が簡単になります。これが動的計画法(**dynamic programming, DP**)の考え方です。言い替えると、動的計画法とは次のような手法です。

> ある問題に対する解が、その問題より小さいサイズの部分問題に基づいて求められる(1)場合に、小さいサイズの問題から順に解を記録しながら解く(2)ことで元の問題に対する解を求める。

ここで(1)は分割統治手法であること、(2)はメモ化(通常は配列への記録)を用いることを表していると言えます。なお、英語名称の**dynamic programming**というのは考案者がそうつけただけで、特別に**dynamic**でも特別に**programimng**でもないので注意してください。

メモ化から動的計画法へ (2)

ではさっそく、先の fib の計算を動的計画法に書き替えた版を示します。メモ配列と関数 fib2 の型が unsigned(32 ビット符号なし 2 進表現) になっていますが、これは最大の 99 まで扱うのには通常の int だとあふれてしまい負の数になるからです。また、符号なし整数の出力には printf の書式文字列で「%u」を指定します。

コードについてはほとんど説明はいらないですが、メモ配列は関数の外に出し (2 つの関数で利用するため)、まず initfib で配列に値を計算してしまい、fib2 では指定した要素を取り出して返すだけとなっています。

メモ化から動的計画法へ (3)

```
// fibdp.c --- calcurate fib with dynamic progrmming.
#include <stdio.h>
#include <stdlib.h>
#define ARRSIZE 100
static unsigned memo[ARRSIZE];
void initfib() {
  memo[0] = memo[1] = 1;
  for(int i = 2; i < ARRSIZE; ++i) { memo[i] = memo[i-1]+memo[i-2]; }
}
unsigned fib2(int n) { return memo[n]; }
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]); initfib();
  printf("fib(%d) = %u\n", n, fib2(n));
  return 0;
}
```

メモ化から動的計画法へ (4)

演習 2 上のフィボナッチ数の動的計画法版を動かし、確認しなさい。納得したら、演習 1 に出て来た以下の再帰関数による計算に対し、動的計画法を用いて計算するプログラムを作ってみなさい。通常版と結果を比較して間違った計算になっていないことを確認すること。

a. 3 の n 乗

b. n の階乗

c. ${}_nC_r$ **(ヒント: メモ配列は 2 次元配列にする必要あり)**

2 次元の動的計画法

例題: 経路数問題

前節では分割統治とメモ化の組み合わせとして動的計画法を定めていましたが、もっとくだけた言い方として「全部答えが埋まるまで配列を埋めて行く」というふうにとらえた方が分かりやすいことがあります。とくに、2 次元配列を用いる場合は (2 次元は紙の上に描きやすいので) そうです。

典型例として、経路数問題を取り上げます。問題は簡単で、図 2 のようなます目で「S からスタートし、東 (右) か南 (下) のます目へのみ動けるものとしたとき、G に到達する経路は何通りあるか」というものです。右のバージョンでは、網掛けになっているます目は通れないものとします。

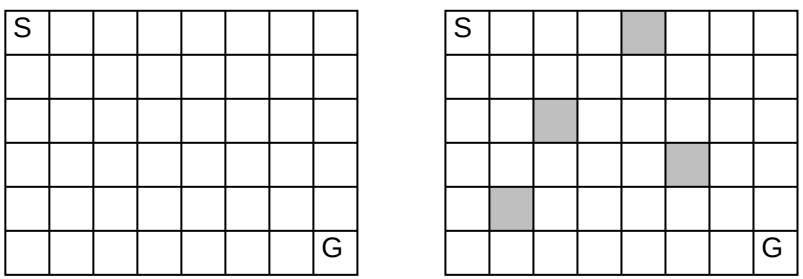


図 2: ゴールまでの経路数は?

例題: 経路数問題 (2)

この問題はどのように考えたらいいでしょうか。まず、全部通れる左の問題からです。上端の 1 列、左端の 1 列については「S からずっと横/縦に移動してくる」以外の方法がないことはすぐ分かります。なので、これらのます目では経路数は「1」です。

それ以外のます目についてはどうでしょう? あるます目に来るには「左隣のます目から 1 個右に来る」場合と「上隣のます目から 1 個下に来る」場合の 2 通りがあり、そしてこれらは (当然) 違う経路です。ということは、「左隣のます目の経路数」「上隣のます目の経路数」が分かっていれば、その 2 つを足したものが「このます目の経路数」なわけです。

前記の「1」を記入したあと、配列を上/左から順に埋めていけば、どの場所でも左/上は既に記入済みですから、上の計算は問題なくできます。プログラムを示しましょう。

例題: 経路数問題 (3)

```
// numpath1.c --- number of paths using DP.
#include <stdio.h>
#define W 8
#define H 6
static int map[H][W];
int main(void) {
  for(int i = 0; i < W; ++i) { map[0][i] = 1; }
  for(int j = 0; j < H; ++j) { map[j][0] = 1; }
  for(int i = 1; i < W; ++i) {
    for(int j = 1; j < H; ++j) {
      map[j][i] = map[j][i-1] + map[j-1][i];
    }
  }
  printf("%d\n", map[H-1][W-1]);
  return 0;
}
```

例題: 経路数問題 (4)

では動かしてみましょう。

```
% gcc8 numpath1.c
% ./a.out
792
```

792通りだそうです。実際、図**3**左のように手で埋めると (あまりやりたくないですが)、確かに**792**通りです。

1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792

1	1	1	1		0	0	0
1	2	3	4	4	4	4	4
1	3		4	8	12	16	20
1	4	4	8	16		16	36
1		4	12	28	28	44	80
1	1	5	17	45	73	117	197

図 3: 経路数のます目を埋めた結果

例題: 経路数問題 (5)

実は左側の障害物のない問題は「左上から右下まで移動する12ステップのうち、5ステップは下への移動、残りは右への移動」であることから、すべての場合は組合せの数${}_{12}C_5$となり、確かに792です。ですが、図の右のように「通れない所がある」場合はそう簡単には行きませんから、そのような場合はプログラムで動的計画法を使うのがよいでしょう。あと、「斜めに右下に行ける」ことにした場合もそうですね。

演習3 上の経路数のプログラムを動かし、結果を確認しなさい。ます目が8x6以外の場合も試してみること。そのあと、次のことをしなさい。

a. 図2右の網掛けの部分が通れないとした場合のSからGまでの経路数を求めるプログラムを書きなさい。(ヒント: 通れない部分は「-1」などの目印を入れておき、その目印だったら加算しない。)

b. 斜めにも移動でき、障害物がない場合について経路数を求めるプログラムを書きなさい。

c. 斜めにも移動でき、障害物がある場合について経路数を求めるプログラムを書きなさい。

方向の決まらない場合/トレースバック

さて、先の問題 (障害物あり) をさらにおしすすめて、図 4 のように迷路にしたとします。今度は、迷路を最短何ステップで抜けられるかという問題にします。今度は、進む方向が右と下だけとは限りませんが、どうしたらいいでしょうか。次の疑似コードを見てください。

- S のます目に 1 を記入し、残りの (障害でない) ます目に 999 を記入
- 値が変化しなくなるまで繰り返し、
- 　すべてのます目について、
- 　　上下左右のます目 (障害物除く) の数値+1 の最小値が現在のます目の値より小さいなら、
- 　　ます目にその最小値を書き込む
- 　ここまでが枝分かれの範囲
- ここまでが繰り返しの範囲
- ここまでが繰り返しの範囲

方向の決まらない場合/トレースバック (2)

「変化しなくなるまで繰り返し」というループがありますが、その実現にはバブルソートの時と同じように旗を使い、ループの先頭で旗を立て、最小値を書き込んだら降ろすようにすれば、次のループ先頭で旗が立ったままのときは変化しなくなったことが分かる、というふうにします。

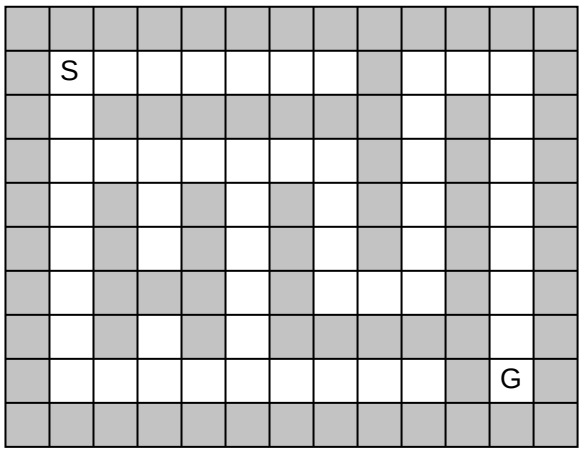


図 4: ます目を使った迷路

方向の決まらない場合/トレースバック (3)

ます目の計算を行う方向が一定でなかったとしても、「最小のステップ数」を求めるのですから、上の疑似コードのように「これ以上変化しなくなるまで繰り返し」最小値を記入していけばよいのです。そして、正の整数値は無限に小さくなり続けることはできませんから、このアルゴリズムは必ず停止し、そのとき G のます目に記入されている値が最小のステップ数です (図 5)。なお、最初に G のます目に記入されたときに停止してはいけないことに注意。ほかの経路でもっと短いものがまだ残っている可能性がありますから。

	1	2	3	4	5	6	7		19	20	21	
	2								18		22	
	3	4	5	6	7	8	9		17		23	
	4		6		8		10		16		24	
	5		7		9		11		15		25	
	6				10		12	13	14		26	
	7		11		11						27	
	8	9	10	11	12	13	14	15	16		28	

図 5: ます目にステップ数を記入した迷路

方向の決まらない場合/トレースバック (4)

ところで、迷路にはもう1つ別の問題が残っています。それは「その最短ステップ数の経路は具体的にどこを通る経路か」も知りたい、ということです (知りたいですよね?)。

それには、次のようにします。まず、ます目を表すのとまったく同じサイズの2次元配列を用意します。そして、ます目に最小値を記入するとき、「上下左右のどちらから来た値が最小か」をこちらの配列に記録しておきます。そうすれば、Gから逆に「記録した最小の方向を次々にたどる」ことでSまでの経路が分かります。これを、目的地から逆向きにたどることから、トレースバック (trace back) と呼びます。

このように、動的計画法で最小値や最大値を求めたあと、具体的にどの選択を取った結果その最小/最大を達成したかを知りたい場合、各地点での選択を記録するような (トレースバックのための) データ構造が別途必要になるわけです。

方向の決まらない場合/トレースバック (5)

演習 4 迷路の問題について次のことをおこないなさい。迷路そのものは好きな大きさ/形にしてよい。

a. 迷路の最小ステップ数を出力する動的計画法プログラムを作りなさい。

b. 上記に加えてトレースバックをおこない最短経路を表示しなさい。

c. さらに上記に加えてトレースバックを分かりやすく表示しなさい。画面に **ASCII** 文字を使って図を表示してもいいですし、**EPS** ライブラリを使ってもいいですし、いっそアニメーションを生成してもよいです。

なお、トレースバックでは「目的地から逆向きの」列が求まりますが、それを順方向に直したければ、配列なりスタックなりを使って適宜行ってください。

最長共通部分列

動的計画法は「2つの列がどれくらい似ているか」を調べるのにも使えます。その具体例として、2つの文字列の対応づけの問題を考えましょう。たとえば、「isasaka」「sassa」「wakasa」の3つの文字列のうち、互いに一番似ているのはどの2つだと思いますか。

「似ている」の定義によりますが、ここでは「2つの文字列の同じ文字を対応させた組ができるだけ多くできる」ものが似ているものと考えます (図6)。ただし、対応づけの線はクロスしてはいけないものとします。こうすると、「isasaka」「sassa」の組が一番多く (4箇所で) 対応づけができると分かります。

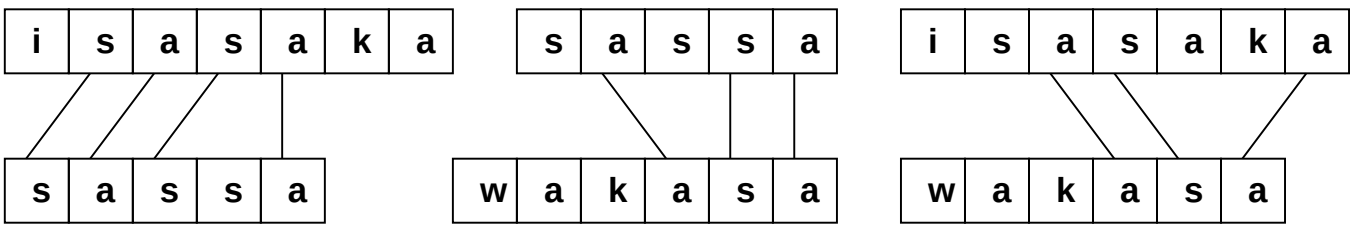


図 6: 最長共通部分列

最長共通部分列 (2)

なお、この問題「最長共通部分列 (**longest common subsequence, LCS**) と呼ばれています。なぜなら、両方の文字列から一部を (順序を変えずに) 抜き出して来たもの (部分列) で、互いに一致するもの (共通部分列) のうち、最長のものを求めているからです。

では、これをどのようにして求めるのがいいでしょう。実は、文字列の対応づけは先にやった「ます目の経路」で表すことができます。具体的には、次のように考えます。

最長共通部分列 (3)

図7左のように、比較する2つの文字列それぞれに「カーソルを」対応させます。カーソルは、2つの文字列で次の文字位置を対応づける場合は、同時に進めます。そうでない場合は、aだけ、またはbだけを進めます。aの文字列の各位置とbの文字列の各位置を縦横に並べた図7中のような表を考えます。ここで、aのカーソルだけを進めることは、右のます目への移動を表し、bのカーソルだけを進めることは、下のます目への移動を表し、同時に進めることは斜め右下への移動を表します。そうすると、あらゆるカーソルの移動はます目の中の移動で表されます。

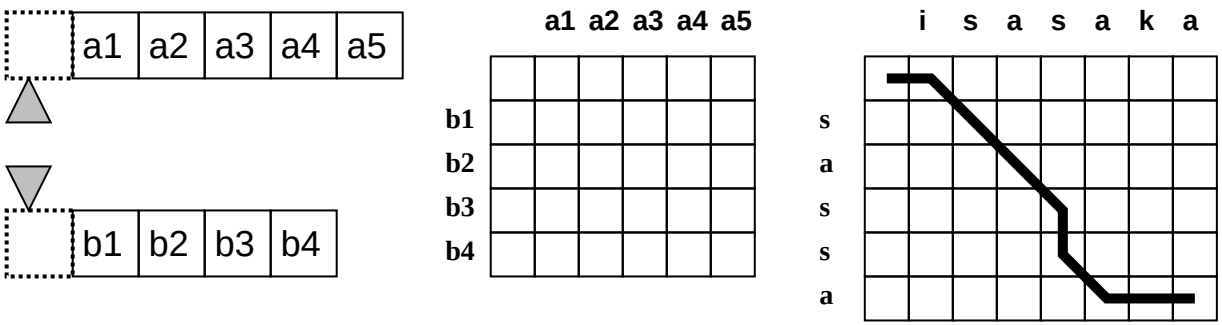
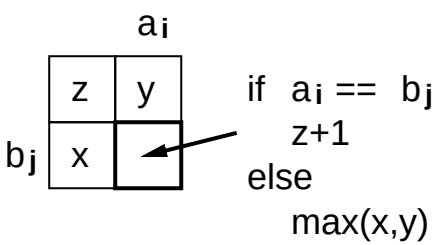


図 7: 文字列の対応づけの考え方

最長共通部分列 (4)

ここでLCSの問題をこの表現にあてはめると、対応づけ (斜めの移動) は対応する位置の文字どうしが等しい場合に限るという条件で、斜めの移動の最大数を求める、という問題と同等です。

そこでまず、上端/左端の列は (斜めの移動はないので) すべて0を入れます。その上で、それぞれのます目を図8左のように、「斜め移動できるなら斜め左上の数プラス1、そうでないなら上または左の数値のうち大きい方」を入れる形で埋めていきます。そうすると、右下隅に入った値が対応づけられる個数の最大数となるわけです。



		i	s	a	s	a	k	a
	0	0	0	0	0	0	0	0
s	0	0	1	1	1	1	1	1
a	0	0	1	2	2	2	2	2
s	0	0	1	2	3	3	3	3
s	0	0	1	2	3	3	3	3
a	0	0	1	2	3	4	4	4

図 8: 動的計画法による LCS の計算

最長共通部分列 (5)

これを C のプログラムにしたものを示します。上に示したように配列の上端と左端を 0 に初期化し、あとは左上から順にそれぞれのます目の値を計算すれば済みます。文字列の対応が無い場合は上と左のます目のうち大きい方、文字列の対応がある場合はさらに斜め左上のます目の値+1 も加えた最大値を入れるということです。

最長共通部分列 (6)

```
// lcs.c --- longest common sequence length for two strings.
#include <stdio.h>
#include <string.h>
#define MAXSTR 50
static int a[MAXSTR][MAXSTR];
int lcs(char *s1, char *s2) {
  int l1 = strlen(s1), l2 = strlen(s2);
  for(int i = 0; i <= l1; ++i) { a[0][i] = 0; }
  for(int j = 0; j <= l2; ++j) { a[j][0] = 0; }
  for(int j = 1; j <= l2; ++j) {
    for(int i = 1; i <= l1; ++i) {
      int m = a[j][i-1];
      if(m < a[j-1][i]) { m = a[j-1][i]; }
      if(s1[i-1] == s2[j-1] && m < a[j-1][i-1]+1) { m = a[j-1][i-1]+1; }
      a[j][i] = m;
    }
```

```
  }
//for(int j = 0; j <= l2; ++j) {
//  for(int i = 0; i <= l1; ++i) { printf("%3d", a[j][i]); }
//  printf("\n");
//}
  return a[l2][l1];
}
int main(int argc, char *argv[]) {
  char *s1 = argv[1], *s2 = argv[2];
  printf("lcs(%s,%s) = %d\n", s1, s2, lcs(s1, s2));
  return 0;
}
```

最長共通部分列 (7)

2次元配列の表示はコメントアウトしてありますが、必要ならはずして見てください。実行例を示します。

```
% gcc8 lcs.c
% ./a.out isasaka sassa
lcs(isasaka,sassa) = 4
% ./a.out sassa wakasa
lcs(sassa,wakasa) = 3
% ./a.out isasaka wakasa
lcs(isasaka,wakasa) = 3
```

最長共通部分列 (8)

なぜこれでLCSが計算できるかを考えるには、LCSを次のように再帰関数として定義してみるのが分かりやすいかも知れません。

$$lcs(a_1 \cdots a_m,\ b_1 \cdots b_n) = \begin{cases} 0 & (m = 0 \quad or \quad n = 0) \\ lcs(a_1 \cdots a_{m-1}, b_1 \cdots b_{n-1}) + 1 & (a_m = b_n) \\ max(lcs(a_1 \cdots a_m, b_1 \cdots b_{n-1}), \\ \qquad lcs(a_1 \cdots a_{m-1}, b_1 \cdots b_n)) & (otherwise) \end{cases}$$

つまり、2つの文字列の最後の文字が等しければ、その両方を削除した文字列どうしのLCSより1多い値が全体のLCSであり、等しくなければ、aから最後の1文字を除いた場合のLCSとbから最後の1文字を除いた場合のLCSのうち大きい方が全体のLCSということになります。

最長共通部分列 (9)

演習 5 上の LCS のコードでは具体的な最長部分文字列を求めていない。トレースバックをおこない、具体的な最長部分文字列も求めるようにしなさい。

演習 6 LCS の問題の類似品として編集距離 (edit distance) というものがある。これは、文字列 s_1 に対して何回 (1)1 文字挿入、(2)1 文字削除、(3) 隣接する 2 文字の交換を行ったら文字列 s_2 に変更できるかの回数の最小値である。LCS のプログラムを参考に編集距離のプログラムを作れ。

本日の課題 14A

「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 14a 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. この授業開始時の自分と現在の自分を比べてどのような変化があったと思いますか。

   Q2.「プログラミングができる」ようになるためにはどのように学ぶ (教わる) のがよいと考えますか。

   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。