パズルで鍛えるアルゴリズム力

第6回

虫食算を解いて、作る!

前回は、筆算を復元するパズルである覆面算を作る方法について扱いました。今回も、同じく筆算を復元するパズルとして虫食算を扱います。前回と同様、虫食算パズルを「解く」だけではなく、「作る」方法を考えます。

Author けんちょん(大槻 兼資、Kensuke Otsuki) 株式会社NTTデータ数理システム E-mail dr.ken.1215@gmail.com Twitter @drken1215

虫食算とは

虫食算[注1]は、足し算や掛け算や割り算の筆算において、いくつかの数値が□で置き換えられたものが与えられ、もとの筆算を復元するパズルです。復元に際してのルールは次のとおりです。

- 1個の□には0から9までのいずれかの数値が入る
- 最上位の桁の□には0は入らない

たとえば、図1の虫食算を実際に解いてみましょう[注2]。数値「3」が「3の形」に並んでいて、楽しい見た目となっています。

まず、図2の矢印で記した段に着目します。「□33□□×3=□□□□3」となる必要があることから、最上段の一の位が「1」と確定します。次に、図3の矢印で記した段に着目することで、2段めの百の位が「3」と確定します。それによって、最上段の十の位も「1」と確定します。さらに図4のように、最下段で繰り上がりが発生することに着目すると、最上段の万の位が「3」でなければならないことも確定します。そして残りのマスを自然な流れで数値を埋めていくと、

▼図1 「3」の形をした虫食算

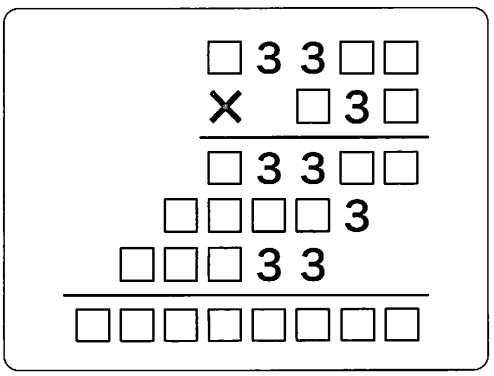


図5のように、解がただ1つに決まることがわかりました。このように、虫食算の作品は通常、一意解となるように作られています[注3]。

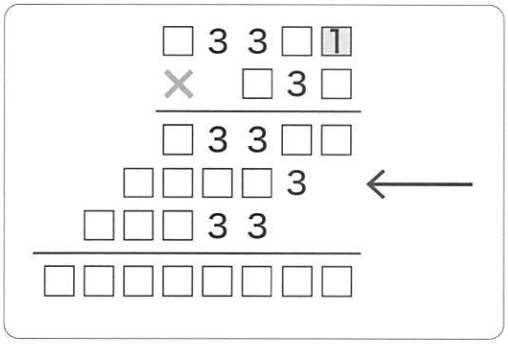
本記事では、虫食算をお手軽に作ることのできるアルゴリズムを紹介します。この方法を用いることで、高品質で美しい虫食算を高確率で作成できます。なお、本記事のソースコードはC++で記述していますが、基本的な機能のみを用いているため、ほかの多くの言語でも同様の処理を実現できます。

虫食算の例

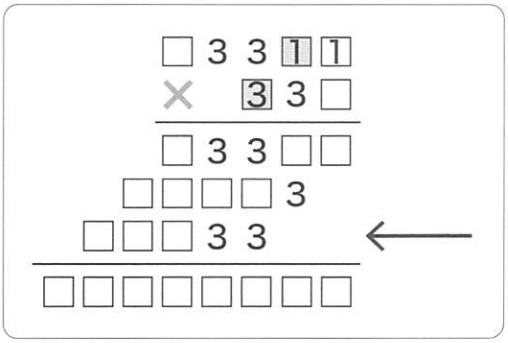
虫食算を解くアルゴリズムや作るアルゴリズ

注1) 「虫食い算」と表記される場面をよく見ますが、虫食算愛好家の間では「虫食算」と表記するのが通例となっています。
注2) 図1の虫食算は、実際に本記事の方法によって作成した自作問題です。
注3) 一般にパズルにおいて、条件を満たす解がただ1つに決まるとき、一意解であると言います。虫食算に限らず、数独など多くのペンシルパズルでは、その作品に一意解であることを課しています。

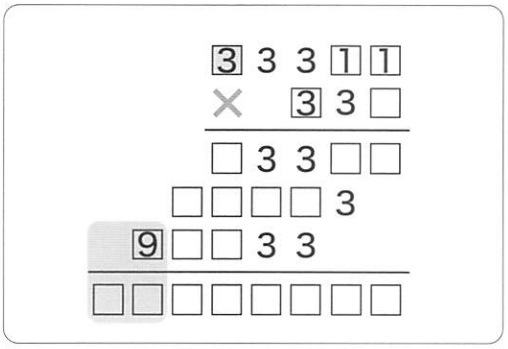
▼図2　「3」の虫食算を解く過程1



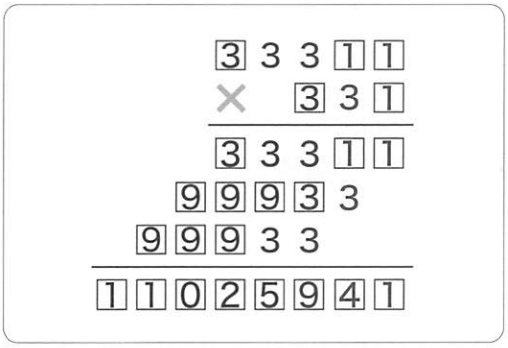
▼図3　「3」の虫食算を解く過程2



▼図4　「3」の虫食算を解く過程3



▼図5　「3」の虫食算の答え



ムを考える前に、おもしろい虫食算作品をいくつか紹介します。

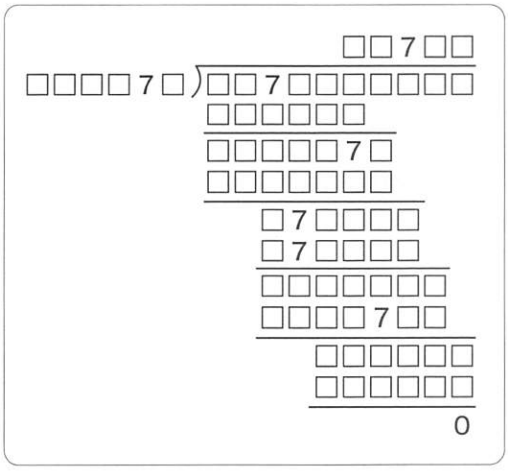
nつのn

図6は、1906年にE. H. Berwickによって発表された作品「7つの7」です。ヒントとなる数値が7個[注4]であるのに対し、□が72個もある作品です。見た目も美しく、虫食算が世界的に流行するきっかけを作った記念碑的名作です。このように、数値nをn個表出した虫食算を「nつのn」と呼ぶことがあります。本記事でも、「7つの7」に分類される虫食算を作ります。

孤独のn

図7は、1925年にE. F. Odlingによって発表された作品「孤独の7」です。ヒントとなる数値

▼図6　E. H. Berwick作「7つの7」



が1個しかないという衝撃的な作品です。その見た目のインパクトから、先述の「7つの7」とともに虫食算が大きく流行するきっかけとなり

注4) 割り算の虫食算において、割り切れたことを示す最下段の0は、ヒントとなる数値の個数としてカウントしないのが通例となっています。

ました。このように、数値nを1個表出した虫食算を「孤独のn」と呼びます。図8も「孤独の8」に分類される有名問題です。図7、8の虫食算は、ともに見た目とは裏腹に、パズル問題としての難易度はやさしめです。ぜひ挑戦してみてください。

小町表出虫食算

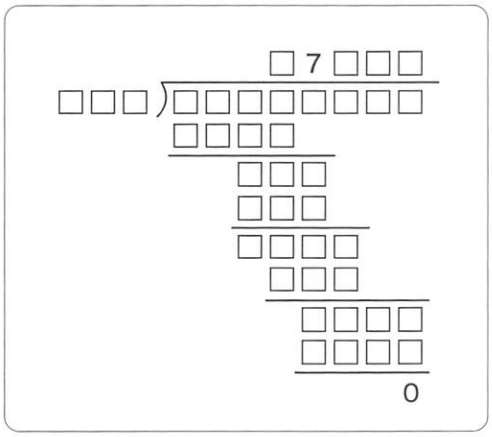
1から9までの数値を1個ずつ表出した虫食算を、小町表出虫食算と呼ぶことがあります[注5]。見た目が美しいだけでなく、解きごたえのある作品となることが多いです。図9、10はいずれも、本記事で紹介するアルゴリズムによって、実際に作成した小町表出虫食算です。

虫食算を解く

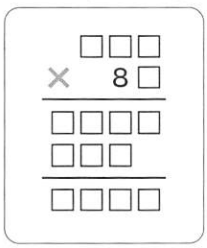
それではまず、虫食算ソルバーを実装しましょう。一意解ではない入力が与えられたとしても、それに対して解をすべて求めるアルゴリズムを実装します。ここでは掛け算の筆算のみを扱いますが、割り算の筆算に対しても同様に考えることができます。また簡単のため、2段め(掛ける数)に0が入るものは扱わないこととします。

なお、虫食算の入力はリスト1のような形式で、標準入力で受け取るものとします。リスト1は、図1の虫食算を表しています。1行めには、最上段の桁数と、2段めの桁数(dとします)を記入します。続く2行は、最上段の情報と、2段めの情報を表します。さらに続くd行は、筆

▼図7　E. F. Odling作「孤独の7」



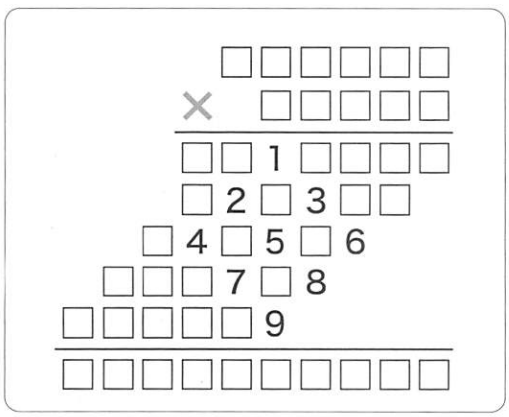
▼図8　「孤独の8」



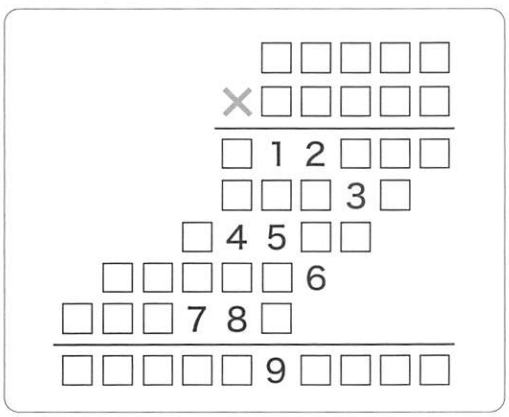
▼リスト1　虫食算(図1)の入力例

```
5 3
*33**
*3*
*33**
****3
***33
********
```

▼図9　小町表出虫食算その1



▼図10　小町表出虫食算その2



注5) 単に小町虫食算と言った場合は、「□が9個あって、1から9を1個ずつ入れることで計算式を復元する虫食算」という別のものを指すことが多いです。

算の過程を表します。そして最後の行は、筆算の結果を表します。また、虫食算の□に対応するところは文字*で表しています。

虫食算を解く方針：枝刈り探索

まず掛け算の筆算においては、最上段と2段めの値を決めれば、残りの部分の値がすべて決まることに注意しましょう。そこで虫食算を解く方法として、次のような探索方法が考えられます[注6]。

- 最上段については、すべての可能性を順に試す
- そのそれぞれについて、2段めの一の位、十の位、百の位、……に順に数値を入れながら条件を満たすものを探索する

具体的には、**リスト2**のような再帰関数を用いて実装できます。整数の一の位、十の位、百の位、……を順に0桁め、1桁め、2桁め、……と呼ぶことにします。またmu.reset_second(d)は、虫食算muの2段めにおいて、d桁めに置いた値を0にする処理を表しています。これは、探索を一歩前の状態に戻しながら次の選択肢を試していく「バックトラッキング法」を適用するためのものです。

探索における注意点として、2段めの0桁めから順に数値を入れていくとき、矛盾が生じた場合にはそれ以降の桁についての探索を打ち切ります。このように、解を導く見込みのない選択肢を切り捨てる考え方を「枝刈り」と呼びます。枝刈りは、実用的なアルゴリズムを設計するうえでたいへん有効な考え方です。

▼リスト2　虫食算を解く探索アルゴリズム

```
// 虫食算の2段めをd桁めから再帰的に決めていく
void rec(Mushikuizan &mu, int d) {
    if (すべての桁に矛盾なく数値が入ったら) {
        解を格納する;
        return;
    }

    // d桁めの値を決めて、次の桁へとバックトラッキング
    for (int v = 1; v <= 9; ++v) {
        if (2段めのd桁めにvを入れて矛盾を生じない) {
            rec(mu, d + 1, res);
            mu.reset_second(d);
        }
    }
}

// 虫食算を解く(最上段の桁数をA桁とする)
void solve(Mushikuizan &mu) {
    for (long long val : (A桁の整数)) {
        if (最上段をvalとして矛盾が生じないならば) {
            rec(mu, 0, res);
        }
    }
}
```

虫食算ソルバーの実装

本記事では、虫食算を解く過程を管理するクラスMushikuizanを、**リスト3**のように実装しておきます。また、各種メンバ関数については**リスト4**のように実装します。ここで、メンバ関数is_digit_valid(r)を切り離して実装しているのは、のちほど解説する「虫食算を作る方法」を意識しています。この虫食算クラスを用いて、虫食算ソルバー全体は**リスト5**のように実装できます。

この虫食算ソルバーに**リスト1**の入力を与えてみましょう。その結果は**図11**のようになります。たしかに一意解であることが見て取れます。筆者の手元の環境[注7]では、0.035秒の計算時間で解が得られました。また、**図8～10**の虫食算もそれぞれ入力として与えてみると、解くのに要した計算時間は0.012秒、6.2秒、0.34秒でした。

注6）ここで紹介する方法は、わかりやすさのため単純なものとしています。図9、10のような中規模サイズの虫食算は解くことができますが、大規模サイズの虫食算を解くためにはより洗練された工夫が必要です。そのような工夫については、筆者がSlideShareにアップロードした資料で紹介しています。23桁×20桁という大規模サイズの虫食算も1秒以内に解くことができます。
https://www.slideshare.net/drken1215/ver-86356575

注7）MacBook Air(13-inch, Early 2015)、プロセッサ：1.6 GHz Intel Core i5、メモリ：8GB

▼リスト3　虫食算を解くためのクラス

```
class Mushikuizan {
 private:
    // 虫食算の問題を表す情報
    std::vector<std::string> problem_;

    // 虫食算を解く過程を表す情報
    std::vector<long long> field_;

    // 現時点でr段めの桁数に矛盾が生じていないかを確認する
    bool is_digit_valid(int r);

    // 現時点でr段めに矛盾が生じていないかを確認する
    bool is_valid(int r);

 public:
    // コンストラクタ
    Mushikuizan(const std::vector<std::string> &input) :
    field_(input.size(), 0) {
        for (int i = 0; i < input.size(); ++i) {
            std::string str = input[i];
            std::reverse(str.begin(), str.end());
            problem_.emplace_back(str);
        }
    }

    // 虫食算の最上段の桁数を返す関数
    size_t get_digit_first() { return problem_[0]. ☑
size(); }

    // 虫食算の2段めの桁数を返す関数
    size_t get_digit_second() { return problem_[1]. ☑
size(); }

    // 虫食算のr段めのd桁めの情報を返す関数
    // "*"の場合や、最上位の桁より上にアクセスするときは-1
    int get(int r, int d) {
        if (d >= problem_[r].size()) return -1;
        else if (problem_[r][d] == '*') return -1;
        else return problem_[r][d] - '0';
    }

    // 虫食算の最上段に数値を置く関数
    // 矛盾が発生する場合はfalseを返す
    bool set_first(long long val);

    // 虫食算の2段めのd桁めに数値vを置く関数
    // 矛盾が発生する場合はfalseを返す
    bool set_second(int d, int v);

    // 虫食算の2段めのd桁め以降の値をリセットする
    void reset_second(int d);

    // fieldの様子を出力する
    void print() {
        for (int i = 0; i < field_.size(); ++i) {
            std::cout << field_[i] << std::endl;
        }
    }
};
```

▼リスト4　虫食算クラスの詳細の実装

```
bool Mushikuizan::is_digit_valid(int r) {
    int digit = 0;
    long long val = field_[r];
    while (val) { ++digit, val /= 10; }
    return (digit == problem_[r].size());
}

bool Mushikuizan::is_valid(int r) {
    if (!is_digit_valid(r)) return false;

    // 一の位から順に数値が一致するかを確認する
    long long val = field_[r];
    for (int d = 0; d < problem_[r].size(); ++d) {
        if (get(r, d) != -1 && get(r, d) != ☑
val % 10)
            return false;
        val /= 10;
    }
    return true;
}

bool Mushikuizan::set_first(long long val) {
    field_[0] = val;
    return is_valid(0);
}

bool Mushikuizan::set_second(int d, int v) {
    // 2段めのd桁めにすでに値がある場合を確認する
    if (get(1, d) != -1 && get(1, d) != v)
        return false;

    // 2段めのd桁めについて計算して確認する
    field_[d + 2] = field_[0] * v;
    if (!is_valid(d + 2)) return false;

    // 2段めに値を追加する
    long long add = v;
    for (int i = 0; i < d; ++i) add *= 10;
    field_[1] += add;

    // dが最上位の桁の場合は掛け算の結果も確認する
    if (d == get_digit_second() - 1) {
        field_.back() = field_[0] * field_[1];
        if (!is_valid(field_.size() - 1)) {
            field_[1] -= add; // 2段めをもとに戻す
            return false;
        }
    }
    return true;
}

void Mushikuizan::reset_second(int d) {
    long long mod = 1;
    for (int i = 0; i < d; ++i) mod *= 10;
    field_[1] %= mod;
}
```

▼リスト5　虫食算ソルバー全体の実装

```
#include <iostream>
#include <vector>
#include <string>

// 虫食算の2段めをd桁めから再帰的に決めていく
void rec(Mushikuizan &mu, int d,
         std::vector<Mushikuizan> &res) {
    // すべての桁に矛盾なく数値が入ったら
    if (d == mu.get_digit_second()) {
        res.emplace_back(mu);
        return;
    }

    // d桁めの値を決めてバックトラッキング
    for (int v = 1; v <= 9; ++v) {
        if (mu.set_second(d, v)) {
            rec(mu, d + 1, res);
            mu.reset_second(d);
        }
    }
}

// 虫食算を解く
void solve(Mushikuizan &mu,
           std::vector<Mushikuizan> &res) {
    // 最上位の値の範囲を出す
    long long mv = 1;
    for (int i = 0; i < mu.get_digit_first() - 1; ++i)
        mv *= 10;

    // 全探索
    for (long long val = mv; val < mv * 10; ++val) {
        // 最上段が整合するならば、2段めを決めていく
        if (mu.set_first(val)) rec(mu, 0, res);
    }
}

int main() {
    int A, B;
    std::cin >> A >> B;
    std::vector<std::string> input(B + 3);
    for (int i = 0; i < B + 3; ++i)
        std::cin >> input[i];

    // 再帰的に解く
    Mushikuizan mu(input);
    std::vector<Mushikuizan> res;
    solve(mu, res);

    // 解を出力
    for (int i = 0; i < res.size(); ++i) {
        std::cout << i << " th result:" << std::endl;
        res[i].print();
    }
}
```

虫食算を作る

いよいよ、虫食算を解くアルゴリズムを有効活用して、虫食算を作る方法を考えてみましょう。まずすぐに思いつく方法として、「計算式が成立する筆算式を作り、そのうちのいくつかの数値を□で置き換えていく」というものが考えられます。しかしこの方法では、表出数値が意味を持つような作品を作ることは難しいです。虫食算を作るとき、「表出数値をおもしろい配置にすること」にはこだわりたいところです。一方、虫食算の各段の桁数についてはそれほど神経を使う部分ではないでしょう。つまり図12のように、

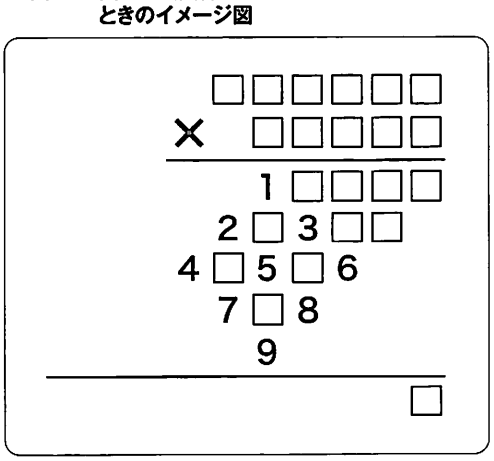
- 表出数値の配置を固定する
- 掛け算の最上段、および2段めの桁数は固定する
- それ以外の段の桁数は固定しない

という条件下で虫食算を作ることができれば喜

▼図11　リスト1(図1)の虫食算の解

```
0 th result:
33311
331
33311
99933
99933
11025941
```

▼図12　図9の虫食算の配置を実現したいと考えたときのイメージ図

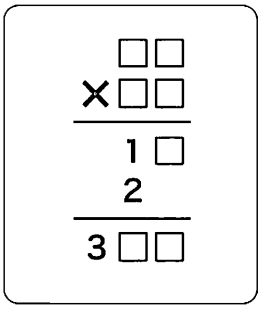


ばしいと言えます。本記事ではその方法を紹介します。

桁数制約を外して解く

たとえば簡単な例として、図13のような配置の虫食算を作ることを考えてみましょう。まず、最上段と2段め以外の段については、桁数の制約を外して解きます。具体的には、リスト4中の関数Mushikuizan::is_digit_valid(int r)が常にtrueを返すように変更します。

▼図13　虫食算の作成に用いる簡単な例



この変更により、図13の配置に対しては9個の解が出力されます(図14)。これらの解を、各段の桁数を並べてできるベクトルによって分類します。このとき、ただ1つの解が属するパターンについては、それ自体が一意解の虫食算となっていることがわかります！　こうして、桁数制約を外して解く方法によって、さまざまな虫食算作品を作れることがわかりました。以上の作業は、たとえばリスト6のように自動化できます。

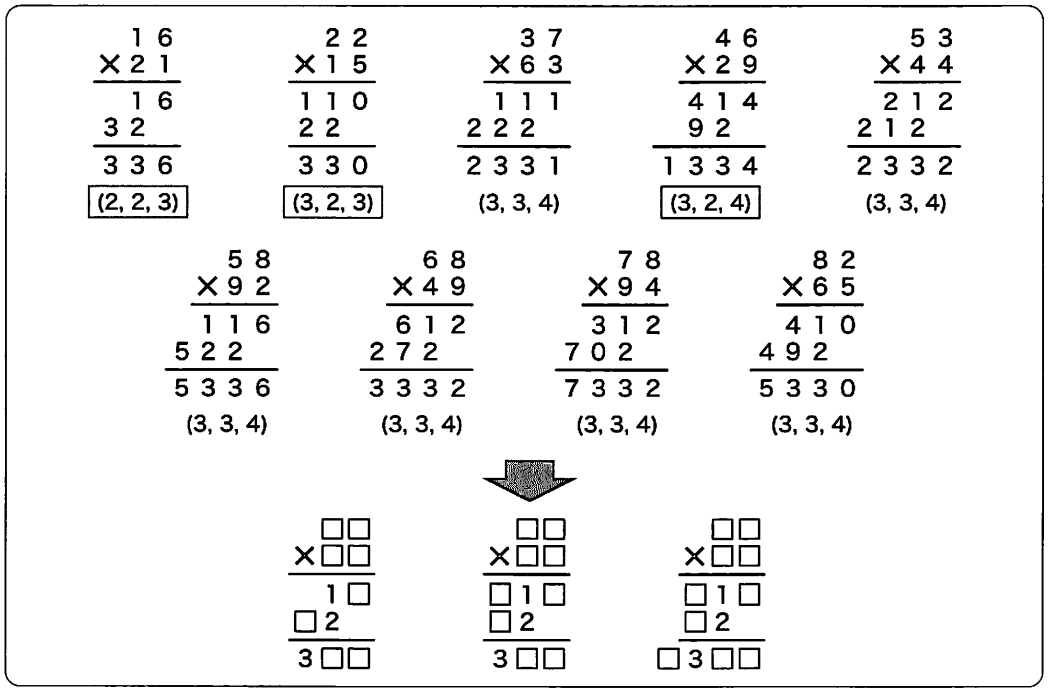
小町表出虫食算の場合

上記の方法を図12の数値配置で試してみましょう。桁数制約を外して解くと108個の解が得られます。そのうち桁数ベクトルが一意となるものを出力すると、6個の一意解作品が見つかります(図15)。図9に示した小町表出虫食算はこのうちの(7, 6, 6, 6, 6, 10)に該当します。また、同様に図10の小町表出虫食算の数値配置で試すと、2個の一意解作品が見つかります。

「7つの7」の場合

最後に、E. H. Berwickによる「7つの7」(図6)の配置も試してみましょう。本記事のソー

▼図14　虫食算の作成手順



▼リスト6　虫食算を作る手順の自動化

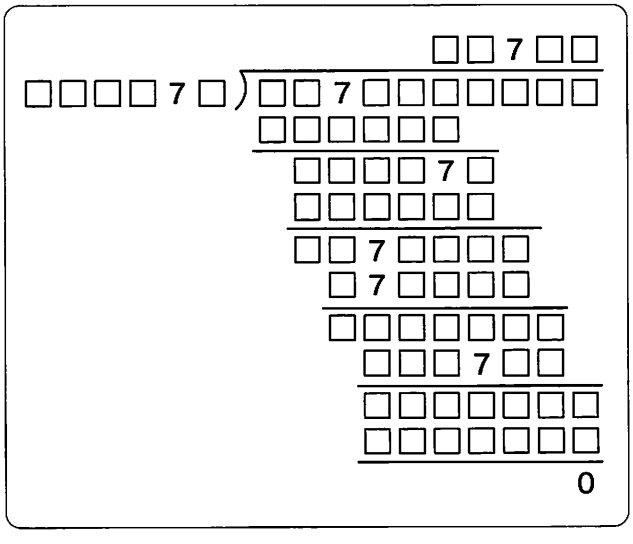
```
// 虫食算の形状を表すベクトルを出力する
std::vector<int> Mushikuizan::pattern() {
    std::vector<int> res(field_.size() - 2, 0);
    for (int d = 2; d < field_.size(); ++d) {
        long long val = field_[d];
        while (val) { res[d - 2]++, val /= 10; }
    }
    return res;
}

int main() {
    // 解を求める部分は略

    // パターンごとにカウントする
    std::map<std::vector<int>, int> nums;
    for (int i = 0; i < res.size(); ++i) {
        nums[res[i].pattern()]++;
    }
    for (auto it : nums) {
        if (it.second != 1) continue;
        for (auto v : it.first) std::cout << v << " ";
        std::cout << std::endl;
    }
}
```

スコードは掛け算についてのみ示していますが、割り算についても同様に実現できます。まず桁数制約を外して解くと5,001個の解が得られます。そのうち、一意解となるパターンを抽出すると7個の一意解作品が得られます。奇しくも「7つの7」は7人兄弟の1人だと言うことができます。

▼図16　E. H. Berwick作「7つの7」の亜種



▼図15　図9の小町表出虫食算の親戚となる6個の虫食算の桁数ベクトル

```
6 6 6 6 6 11
6 6 6 7 7 11
6 6 7 7 7 11
6 7 7 6 7 11
7 6 6 6 6 10
7 6 6 7 7 11
```

さらにそのうち、割られる数の桁数も「7つの7」と等しいもの(10桁)を選ぶと、図16に示す作品が生まれます。オリジナルの「7つの7」と比べるとややややさしめの難易度です。ぜひ挑戦してみてください。

おわりに

今回は、虫食算を「作る」方法について考えました。虫食算も覆面算と同様、パズルとしておもしろいだけでなく、メッセージ性の高い見た目を追求する芸術としての側面もあります。作品を作るアルゴリズムを考える題材として格好のものと言えます。

私事になりますが、筆者は虫食算作りを趣味としており、虫食算を手で作るときにも桁数制約を外して解く方法をよく採っていました。本記事で紹介した方法は、その作問手続きを自動化する試みから生まれました。

世の中のさまざまな現場には、シフトスケジューリングなどをはじめとして、手作業で長時間かけて行われている熟練のノウハウが多数存在します。それらを自動化することで作業時間を短縮したいという需要は大きいです。その際には、人手作業における着眼点を明確にして、それを活かしたアルゴリズムを考えることが肝要と言えます。SD