

3章

私が決して書けなかった、 一番美しいコード

ジョン・ベントリー (Jon Bentley)

ある主任プログラマーが次のような言葉で賞賛するのを聞いたことがあります。「彼は、コードを削ることで機能を追加する」。フランス人の作家で飛行家のサン＝テグジュペリ (Saint-Exupéry) は、同じ心情をもっと一般的に表現してこう言っています。「デザイナーが自分は完璧に達成したんだと分かるのは、付け加えるべきものが何もない時ではなく、取り去るべきものが何もない時である」。ソフトウェアでは、一番美しいコード、一番美しい関数、一番美しいプログラムはそもそもそこにはない、ということが時々あります。

もちろん、そこにはないものについて語るのは難しいことです。この氣勢も削られるような困難に、本章では古典的なクイックソートのプログラムの実行時間について、今までにない新しい分析をしてお見せすることで、挑んでみます。第1節では、個人的な見方でクイックソートを再検討して、準備を整えます。その次の節が、この章の肝心なところになります。まず最初に、プログラムにカウンタを1つ付け加え、次にコードをどんどん小さく、しかしどんどん強力で作り変えていき、最後にはほんの2、3行が平均実行時間を完全に占めるまで進めます。第3節はこのテクニックについてまとめ、2分探索木のコストを簡潔に分析してお見せします。最後に2節にわたって、この章を読んであなたが今よりエレガントなプログラムを書く参考になる知見を引き出します。

私が書いたことのある、一番美しいコード

グレッグ・ウィルソン (Greg Wilson) 氏が最初この本のアイデアを説明してくれたとき、私は、「自分が書いたことのある一番美しいコードは何だったか?」を、まず自分自身に問いました。この心から愉しくなる問いかけが一日中私の脳の周りでぐるぐる廻った後、私ははたと答えが分かりました。クイックソートだったのです。ただあいにく、この問いかけには、フレーズの取り方によって、3つの違う答えがありました。

私は分割統治型の各種アルゴリズムについて論文を書き、その後ホア (C. A. R. Hoare) のクイックソート (Computer Journal 第5巻「Quicksort」掲載) がそれらアルゴリズム全部のおじいちゃん位の祖先にあたると思いました。その論文のアルゴリズムは基本的な問題を解く美しいアルゴリズムであり、エレガントなコードで実装できます。私はそのアルゴリズムをととも気に入っていましたが、いつも一番内側のループのあたりではムズムズしていました。そのループに基礎を置く複雑なプログラムを2日間に渡ってデバッグしたことがあり、その後何年間も同様の仕事をする必要があるたびに、そのコードを注意深くコピーして使ったのです。そうやって私は自分の問題を解決はしましたが、私は本当にはそれを理解していなかったのです。

とうとう私は、ニコ・ロムト(Nico Lomuto)からエレガントなパーティション(最内側ループでやる作業)のやり方を教わり、最終的にはちゃんと理解できて、正しいことが証明さえできるクイックソートのプログラムを書くことができました。ウィリアム・ストラंक Jr.[†] (William Strunk Jr.) の「勢いのある筆は簡潔である」という観察は、英文だけでなくコーディングにも適用できるので、私は彼の「不要な言葉は省きなさい」という忠告にも従いました(以上の格言は『The Elements of Style』から)。それで結局、だいたい40行ほどのコードをたったの12行までにも減らしたのです。ですから、「あなたが書いたことのある一番美しいコードの小片は何ですか?」という問いかけに対してなら、私は自分の本『「珠玉のプログラミング」(原題『Programming Pearls Second Edition』)に載せたクイックソートです」と答えることになるでしょう^{††}。Cで実装したクイックソートの手続きを例3-1に示します。以下ではこのコードをよく検討し、次の節で改良して行きます。

例 3-1 C で実装したクイックソート

```
void quicksort(int l, int u)
{
    int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

このコードは、引数を与えて quicksort(0, n-1) のように呼び出すと、グローバルな配列 x[n] を並べ替えます。2つの引数は、配列の並べ替える部分の範囲を添字で指定するもので、l が下限、u が上限の添字です。swap(i, j) は、x[i] と x[j] の中身を入れ替えます。最初の swap は、l と u の間の一様乱数を1つ選んで、その位置の要素が持つ数をピボット(分割に使う値)にします。

『珠玉のプログラミング』では、関数 quicksort をどうやって導き出していったかを説明し、またこの関数が正しいことが証明してあります。この章の以下では、その本のレベルの説明を理解できているか、他のアルゴリズム入門書などを見てクイックソートを良く分かっている読者を対象に議論を進めます。

もし問いかけが、「あなたが書いたことのある一番美しいコードで、広く使われているものは何ですか?」と変わっても、再び私の答えはクイックソートです。マクロイ (M. D. McIlroy) と共著の論文(「整列関数をエン

[†] 訳注: William Strunk Jr. の小冊子『The Elements of Style』は、英文作法についての永遠の名著とされています。現在でも入手できます。ISBN-10: 0486447987, ISBN-13: 978-0486447988。日本語版(改訂増補版)は次の通りです。ストラंक著、松本訳、『英語文章読本』、荒竹出版、1979。

^{††} 訳注: Bentley 著『Programming Pearls Second Edition』(Addison-Wesley) 日本語版は次の通りです。ベントリー著、小林訳、『珠玉のプログラミング』、ピアソン・エデュケーション、2000。ISBN-13: 978-4894712362。

「ジェニアリングする」、Software – Practice and Experience, Vol. 23, No. 11) で、Unix の由緒ある関数 `qsort` の重大な性能上のバグについて述べました。そして新しいCライブラリ用の整列用関数の構築に着手し、その仕事のためにマージソート、ヒープソートも含む多くのアルゴリズムを検討しました。何通りもの実装を比較して、アルゴリズムとしてはクイックソートを採用することに落ち着きました。その論文には、新しい関数を、他のアルゴリズムに比べてより明快に、より高速に、より頑丈に設計していくには、一体どのようにやったのかを書きました。これらの特性は、新しい関数がより小さかったために得られたという面がかなりありました。ゴードン・ベル (Gordon Bell) の賢明なアドバイス「一番コストがかからず、一番速くて、一番信頼できるコンピュータシステムの要素は、そこに存在していない要素である」は正しいことがまたしても示されました。この関数はもう 10 年以上も、バグの報告もなく、広く使われています。

コードサイズを小さくすることで達成されるさまざまな見返りを考えると、この章の最初の問いかけの3つ目のバリエーションを問いかけなければいけないでしょう。「あなたが決して書かなかった、一番美しいコードは何ですか?」と。ほとんど何もせずに、多くのことを成し遂げるには、一体どうやれば良いでしょう? 3つ目の答えもクイックソートに、特にその実行性能の解析に、関係します。次の節でその話をしましょう。

より少ない行でより多くのことを

クイックソートはエレガントなアルゴリズムであり、込み入った分析の対象にうってつけです。1980年ころ、私はトニー・ホーア[†]と、彼のアルゴリズムの歴史について素晴らしい議論をしました。その時聞いたところでは、彼が最初にクイックソートを開発したときは、あまりに単純なので公表するほどのこともないと思ったそうで、このアルゴリズムの実行時間の期待値を分析できてからはじめて、古典的な論文「クイックソート」を書いた、ということでした。

クイックソートが最悪の場合、 n 個の配列要素を整列するのに n^2 の時間を要することは簡単に確かめられます。最良の場合は、毎回のピボットが中央値(メジアン)になる場合で、 $n \lg n$ ^{††}回の比較で済みます。それでは、 n 個の値がランダムに並んだ配列では、平均何回比較を行えば整列できるのでしょうか?

この問いについてのホーアの分析は美しいですが、残念なことに難しい数学なので多くのプログラマには理解できません。私が学部生にクイックソートを教える時は、真面目に努力しても多くの学生は証明を読んで理解さえできないのがっかりします。そういうわけなので、今回はこの問題を実験から攻めることにしましょう。ホーアのプログラムからはじめて、最終的に彼の分析に近いところまでたどり着こうと思います。

私たちのやるべきことは、例3-1のランダムクイックソートを土台に修正を施し、異なる n 個の値から成る配列を並べ替えるのに必要な比較の平均回数を分析できるようにすることです。その際、最小のコードと実行時間と使用領域とで、最大の洞察を得られるように努力しましょう。

比較の平均回数を求めるために、まず比較回数を数えられるように直します。内側のループにおいて比較を行う前に変数 `comps` を 1 増やすことで、比較回数を数えます(例 3-2)。

[†] 訳注: Charles Antony Richard Hoare、つまり C.A.R.Hoare のこと。ウィキペディアによると「チャールズ・アントニー・リチャード・ホーア卿(Sir Charles Antony Richard Hoare)、もしくはトニー・ホーア(Tony Hoare)、1934年1月11日生まれ、はイギリスの計算機科学者。1960年にクイックソートを開発したことで最もよく知られる。」となっています。2000年に教育とコンピュータサイエンスに関する功績により、ナイトの称号を授与されました。

^{††} 訳注: $\lg N$ は「2を底とするNの対数」つまり「2を何乗するとNになるかの答え」を表します。

例 3-2 内側のループに比較回数を数えるための仕掛けを付ける

```
for (i = l+1; i <= u; i++) {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

n の値を決めてプログラムを実行すると、その回の並べ替えに何回比較を行ったかわかります。 n をいろいろに変えて何回も繰り返し実行して、その結果を統計的に分析すると、クイックソートは n 個の要素を並べ替えるのに平均で $1.4n \lg n$ 回の比較を行うということが観測できるでしょう。

このやり方は、プログラムの振る舞いを考察する方法として悪くありません。13行のコードと何回かの実験で、多くのことが明らかになります。ブレイズ・パスカル (Blaise Pascal) やエリオット (T.S.Eliot) などの作家の有名な言葉として、「もっと時間があったら、もっと短い手紙を書くことができるのに!」という一文があります。私たちには時間があるので、もっと短い(かつ、もっと良い)プログラムを作ろうという試みのために、そのコードを使って実験をしましょう。

統計上の正確さとプログラミングへの考察を高めつつ、この実験をスピードアップするというゲームを楽しんでいきましょう。内側のループはいつも正確に $u-1$ 回の比較をするので、比較回数を数えるのをループの外で1回だけの演算で行うことで、プログラムをほんの少しだけ速くしましょう(例 3-3)。

例 3-3 比較回数の数え上げをループの外に移したところ

```
comps += u-1;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

このプログラムは配列を並べ替え、それをする傍ら並べ替えに使われる比較回数を数えます。ところが、私たちの目標が比較回数を数えるだけなら、実際に配列を並べ替える必要はありません。例 3-4は要素を並べ替えるという「本来の仕事」を取り除いて、このプログラムのさまざまな呼び出しの「骨組み」だけを残したものです。

例 3-4 回数を数えるために骨組みだけになったクイックソート

```
void quickcount(int l, int u)
{
    int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-1;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

このプログラムがうまく使えるのは、クイックソートがピボットを選ぶ方法がランダムであり、またすべての要素に重複がないという前提があるためです。この新しいプログラムは今や、 n に比例する時間で実行できます。また例3-3は n に比例する領域が必要でしたが、使用領域は今や再帰のための呼び出しスタックだけです。平均で $\lg n$ に比例するところまで減らすことができます。

配列の添字(1と u)は実際のプログラムにとっては大切かもしれませんが、骨組みバージョンにはこれは関係ありません。この2つの添字は、並べ替える配列部分の大きさを表す整数(n)で置き換えられます(例3-5)。

例 3-5 配列の大きさだけを引数にする

```
void qc(int n)
{   int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

この手続きはもはや、比較回数数え上げ関数と捉え直した方が自然です。1回のクイックソートのランダム実行で使用された比較の回数を返す関数の形に直したものは、例 3-6 のように書けます。

例 3-6 比較回数数え上げ関数の形にしたクイックソートの骨組み

```
int cc(int n)
{   int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-1) + cc(n-m);
}
```

例3-4、3-5、3-6はどれも、全く同じ基本問題を、しかも同じ実行時間と領域使用で解きます。後のものほど前の関数の形を改良しており、そのおかげで前のものよりも分かりやすく、かつ少少だけ簡潔になります。

ジョージ・ポリヤ(Geroge Polya) は、彼の著書『How To Solve It』(Princeton University Press) [†]で**発明家のパラドックス**について言及し、「野心的なプランほど、成功するチャンスが膨らむ」と言っています。私たちもこのパラドックスをクイックソートの解析に利用してみましょう。ここまでは、「大きさ n のクイックソートを1回実行したら何回比較を行うか?」を扱ってきました。今回は、もっと野心的な問題「大きさ n のランダムな配列を並べ替えるのに、クイックソートでは平均で何回の比較を行うか?」を扱ってみましょう。例 3-6 を拡張して、例 3-7 の擬似コードを導くことができます。

[†] 訳注：日本語版は次の通りです。ポリヤ著、柿内訳、『いかにして問題をとくか』、丸善、1975、ISBN-13: 978-4621045930。

例 3-7 クイックソートの平均比較回数を求める擬似コード

```
float c(int n)
    if (n <= 1) return 0
    sum = 0
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m)
    return sum/n
```

入力の要素数が0または1なら、例 3-6 の通り 1 回も比較をしません。 n が 1 より大きい場合については、例 3-7 では m をピボット値とみなして(つまり最初の要素から最後の要素まで、どれも同確率でピボットになったとして)、そのピボットによる分割のコストを求めます(このために、大きさ $m-1$ の問題と大きさ $n-m$ の問題とを再帰で解いています)。そうしておいてから、これらの値の合計を n で割って平均を求めます。

この値を使えば、実験はずっと強力になります。平均を求めるために n を変えて何度も実験するよりも、1 回の実験で本当の平均がわかります。ただし、あいにくこの強力さのために犠牲になることがあります。このプログラムの実行時間は 3^n に比例してしまうのです(この章全体で説明している解析手法自身を使って、この実行時間を分析してみることは、良い演習になります)。

例 3-7 は、何度も何度も部分問題の解を計算するので、上記のように時間がかかってしまうのです。こういう場合は、再計算をしなくて済むように部分問題の解を覚えておく手法である、**動的計画法**を使うのが常道です。ここでは、 $c(n)$ の計算結果を覚えておく配列 $t[N+1]$ を導入して、その値を使いながら次数を上げていきます。例 3-8 にそのプログラムを示します。

例 3-8 動的計画法で計算するクイックソートの問題

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

このプログラムはほぼ機械的に例 3-7 の $c(n)$ を $t[n]$ に置き換えただけです。実行時間は N^2 に比例し、使用領域は N に比例します。このプログラムの利点のひとつは、実行が終わったとき、配列 t には配列要素の 0 番から N 番までのところに(実験による見積もりではなく)真の平均値が入っていることです。これらの値について分析することで、クイックソートで使われる比較回数の予測を数式の形で表すための洞察が得られます。

このプログラムをさらに簡単にしましょう。まず、項 $n-1$ をループから外に出すと例 3-9 のようになります。

例 3-9 コードをループの外に移して計算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
t[n] = n-1 + sum/n
```

ここで対称性を使って、さらにループを調整しましょう。例えば n が 4 のとき、内側のループは次の和を計算します。

$$t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]$$

並んでいる対を見ると、最初の要素は増えていき、同時に2番目の要素は減っていきます。したがって、上記の和は次のように書き直せます。

$$2 * (t[0] + t[1] + t[2] + t[3])$$

この対称性を使うと、クイックソートの計算は例 3-10 のようになります。

例 3-10 対称性を使って計算する

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
t[n] = n-1 + sum/n
```

しかしこのコードもまだ、同じ和を何度も再計算するので、無駄があります。前の項を全部加えていくのではなく、sum をループの外で初期化し、順次項を加えていくようにすると、例 3-11 のようになります。

例 3-11 内側のループを取り除いて計算する

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
t[n] = n-1 + sum/n
```

この小さなプログラムは大変有用です。 N に比例する時間で、1 から N のすべての整数に対して、クイックソートの真の実行時間予測表を作り出します。

例 3-11 の計算は簡単に表計算ソフトに移すことができます。表計算ソフトを使えば、計算結果をさらに表

計算ソフトの機能で分析できますね。表 3-1 に表の最初の方を示します。

表 3-1 例 3-11 の表計算ソフトによる計算出力

N	Sum	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

表の1行目はコード中にあるのと同じ3つの定数で初期化されています。表計算ソフトの記法を使えば、次の行の数(表の3段目になります)は、以下の関係式を用いて計算されます。

$$A3 = A2+1 \quad B3 = B2 + 2*C2 \quad C3 = A3-1 + B3/A3$$

この(相対セル指定を用いた)関係式を下方へドラッグコピーすることで、表が完成します。スプレッドシート版のクイックソートの計算は、少ない行数で多くのことを成し遂げるという基準点からすれば、「私が書いた一番美しいコード」の実質上のライバルです。

しかしもし、これらの値全部は必要ないとしたらどうでしょうか？ この方法による値を、ほんの一部についてだけ分析したい、というような場合はどうでしょう(例えば、 2^0 から 2^{32} までの2のべき乗のすべて、とか)。例 3-11 は配列 t をすべて作り上げますが、使うのは表の中の最近計算した値だけです。したがって例 3-12 に示すように、配列 t を使って N に比例する領域を消費する代わりに、単独変数 t を使って N に関わらず一定の領域だけで計算することができます。

例 3-12 クイックソートの計算(最終版)

```
sum = 0; t = 0
for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n
```

こうすれば、とびとびの n について分析をしたい時も、 n が分析したい値の時にそれを出力するというコードを挿入することで必要なデータが得られます。

長い道のりでしたが、この小さなプログラムが最終段階です。アラン・パリス(Alan Perlis)の観察「簡単なものから複雑なものになるのではなく、複雑なものに簡単になる」(「Epigrams on Programming」、Sigplan Notices, Vol. 17, Issue 9) はまさに、この章でたどった道のりを振り返るのにぴったりです。

展望

表 3-2 に、この章でクイックソートを解析するのに使ったプログラムをまとめます。

表 3-2 クイックソートの比較回数を数えるプログラムの進化の様子

例番号	行数	答えの種類	答えの数	実行時間	メモリ
2	13	サンプル	1	$n \lg n$	N
3	13	"	"	"	"
4	8	"	"	n	$\lg n$
5	8	"	"	"	"
6	6	"	"	"	"
7	6	厳密	"	3^N	N
8	6	"	N	N^2	N
9	6	"	"	"	"
10	6	"	"	"	"
11	4	"	"	N	"
12	4	厳密	N	N	1

コードの発展の段取りは極めて素直でした。例3-6から3-7へ移るときに厳密な答えを導くように変えたところが、一番巧みな細工でした。このような経過で、コードはより高速でより役立つものになっていきながら、小さくなりました。19世紀半ば、ロバート・ブラウニング(Robert Browning)は「less is more (より少なくは、より多く)」と述べましたが、表3-2はこの最小主義を数量化できた実際の一例として有効なものではないかと思えます。

紹介してきたプログラムは、基本的に3種類のものでした。例3-2と3-3は、実際に配列の中身を整列しながら比較回数を数える仕掛けを装着した、クイックソートのプログラムでした。例3-4から3-6はクイックソートを単純にモデル化したものの実装にあたります。整列時のアルゴリズムを真似ただけで、実際に配列のデータを整列はしません。例3-7から3-12ではそのモデルがどんどん洗練されていき、特定の場合の並べ替えを調べなくても、並べ替えに必要な比較回数の真の平均値を計算しました。

それぞれのプログラムに用いた手法は、次のようにまとめられます。

- 例 3-2、3-4、3-7：問題の定義を根本から変えてしまったところ。
- 例 3-5、3-6、3-12：関数定義のちょっとした変更。
- 例 3-8：動的計画法を実装するために、新しいデータ構造を導入。

上記はいずれも典型的な手法です。このように、「自分(達)が本当に解く必要がある問題は何ですか?」「その問題を解くためなら、もっと良い関数があるのではないですか?」と問い直すことで、プログラムを簡素化できることがしばしばあります。

今回の分析過程を大学の学部生にやってみせたとき、プログラムはついにコード行数がゼロ行にまで縮ん

でしまい、そしてぼっと煙のように消えてしまい、気づいたら数式が残っていました。例3-7は、次のような漸化式に解釈し直すことができます。

$$C_0 = 0 \quad C_n = (n-1) + (1/n) \sum_{1 \leq i \leq n} C_{i-1} + C_{n-i}$$

この式はまさに、ホーアの分析方法、そしてまたそれ以後もクヌース (D. E. Knuth) が古典的名著『The Art of Computer Programming, Volume 3: Sorting and Searching』(Addison-Wesley)[†]の中で示した分析方法なのです。例3-10を導き出した時の、表現し直したり対称性を使ったりするプログラミングのワザを使うと、上の数式の再帰の部分は次のようになります。

$$C_n = n-1 + (2/n) \sum_{0 \leq i \leq n-1} C_i$$

シグマ記号を取り除くクヌースのテクニックを使うとほぼ例3-11に等しいものが得られ、これを、2つの未知数がある2つの漸化式の系として書き直すと次のようになります。

$$C_0 = 0 \quad S_0 = 0 \quad S_n = S_{n-1} + 2C_{n-1} \quad C_n = n-1 + S_n/n$$

クヌースはsumming factorという呼ばれる数学のテクニックを使って漸化式を解き、以下に示す解に達しました。

$$C_n = (n+1)(2H_{n+1}-2) - 2n \sim 1.386n \lg n$$

ここで H_n は n 番目の調和数 $(1+1/2+1/3+\dots+1/n)$ です。こうして私たちは、模索しながら拡張するというやり方でのプログラム上での実験から、アルゴリズムの計算量についての数学的に完全な解析へと、違和感なく考察を進めていくことができました。

得られた数式で、私たちの冒険はおしまいです。締めくくりはアインシュタインの有名な言葉。「すべてのことは、できる限りシンプルにおやりなさい。ただし、シンプル過ぎるはいけません。」

おまけの解析

ゲーテの有名な言葉に、「建築は凍り付いた音楽だ」というのがあります。それと全く同じ意味で、私は「データ構造は凍り付いたアルゴリズムだ」と断言しましょう。そしてもし、クイックソートのアルゴリズムが凍りついたら、私たちが得るデータ構造は2分木です。クヌースの本ではこの構造を提示した上で、その実行時間はクイックソートと類似した漸化式になる、と分析しています。

2分木に要素を1つ挿入する平均のコストを解析したい場合、そのコードを最初にして、次にそのコードを比較回数を数えるように拡張し、それから欲しいデータを集める実験を行います。そうしているうちに、前の節と同様の感じで、コードを簡素化(かつ、機能的には強力に)することができます。この簡素化した解と

[†] 訳注：日本語版は次の通りです。クヌース著、石井他訳『The Art of Computer Programming Volume 3 Sorting and Searching Second Edition』、アスキー、2006、ISBN-13: 978-4756146144

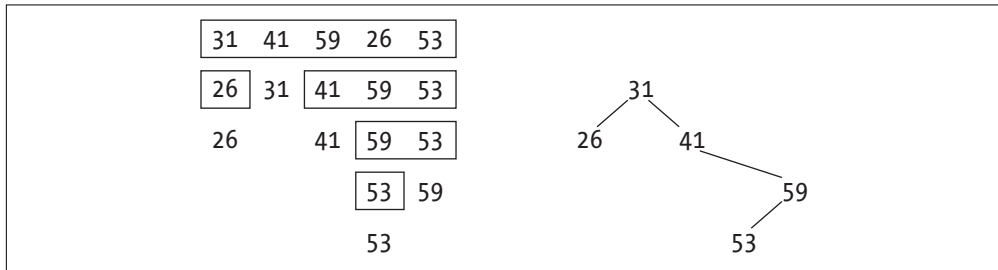


図 3-1 理想的な分割によるクイックソートと 2 分探索木の対応

は、**理想的な分割**(要素の相対的な順番はそのままにしておくような分割)を用いるクイックソートに対応します。図 3-1 に示すように、このクイックソートは 2 分探索木と同形です。

左の箱が理想的な分割でクイックソートが進む様子で、右側のグラフはそれに対応する、同じ入力データから作り出される 2 分探索木です。実際、2 つの処理に必要な比較回数が同じであるだけでなく、比較するデータの組まで全く同じです。重複のないランダムデータをクイックソートして平均実行時間を分析するというのを最初の方でやりましたが、あれはつまり、重複のないランダムデータを 2 分探索木に挿入するのに必要な比較回数の平均を求めていることでもあるわけです。

プログラムを「書くこと」とは?

弱い意味で、私は例 3-2 から 3-12 のプログラムを「書き」ました。最初は走り書きしたメモで、次に学部生の目の前で黒板に書き、最終的にこの本のこの章に書きました。プログラムは系統立って導き出していきましましたし、十分時間をかけて分析もしましたので、掲載したプログラムは正しいものであると私は信じています。しかし、(例 3-11 のスプレッドシート版は別として)どの 1 つのコードもコンピュータのプログラムとしては動かしていません。

ベル研究所にいた 20 年弱の間に、多くの先生から(とりわけカーニハンから…彼のプログラミング教育についての文が本書の第 1 章にあります)、公衆の面前で提示するプログラムを「書くこと」には、単に記号を打ち込むことを大きく超越したものが含まれる、ということを学びました。プログラムをコードに実装し、いくつかのテストケースで実行し、それから足場を築き、テストドライバを作り、系統的にチェックするためのテストケースのライブラリを作ります。さらに理想を言えば、人手の介入なしに、コンパイル検査済みのソースコードを文章中に自動的に埋め込めるようにします。例 3-1 (および『珠玉のプログラミング』に載っているすべてのコード)は、そのようにして書いたものです。

名誉のために書いておきますが、例 3-2 から 3-12 までを実装して動かさないことによって、私はこの章のタイトルが嘘でないようにしたかったのです。約 40 年間に渡ってプログラミングが私に残してくれたのは、ものを作ることの難しさに対する敬意です(バグを出して惨めな思いをするのは怖いものです)。そこでちょっと妥協して、例 3-11 を表計算ソフトで実装し、もう 1 列を追加してそこに解析解を表示させました。その 2 つの結果がぴったり合ったときの私の喜び(と安堵)を想像してみてください! そういうわけで私は、これらの書かなかったけど美しいプログラムを、その正しさにある程度の確信を持って、未発見のバグがある可能性は意識しつつも、世の中に出すことにしたのでした。これらの例題の中に私が見出した深い美しさが、つ

まらない表面上の欠点によって損なわれたりすることのないよう願っています。

これらの書かなかったプログラムをお見せしている苦痛と不安の中にいる私を、パリスの洞察の言葉で慰めてもらおうと思います。「ソフトウェアは他のもの、いつか捨てられてしまうものと何か違うのだろうか？ 石鹸の泡のようなものと思うしかないだろう。」

結論

美しさには、多くの源泉があります。この章では、シンプルさ、エレガントさ、簡潔さ、によって与えられる美しさに注目しました。以下の名言はそのどれもが、この支配的なテーマを表しています。

- コードを削ることで機能を追加しなさい。
- デザイナーが自分は完璧に達成したんだと分かるのは、付け加えるべきものが何もない時ではなく、取り去るべきものが何もない時である。(サン＝テグジュベリ)
- ソフトウェアでは、一番美しいコード、一番美しい関数、一番美しいプログラムはそもそもそこにはない、ということが時々ある。
- 勢いのある筆は簡潔である。不要な言葉は省きなさい。(ウィリアム・ストラック Jr.)
- 一番コストが掛からず、一番早くて、一番信頼できるコンピュータシステムの要素は、そこに存在していない要素である。(ゴードン・ベル)
- だんだん少なくなしながら、だんだん多くのことをするように努力しなさい。
- もっと時間があつたら、もっと短い手紙を書くことができるのに! (ブレーズ・パスカル)
- 発明家のパラドックス: 野心的なプランほど、成功するチャンスが膨らむ。(ジョージ・ポリヤ)
- 簡単なものから複雑なものになるのではなく、複雑なもの後に簡単になる。(アラン・パリス)
- より少なくは、より多く。(ロバート・ブラウニング)
- すべてのことは、できる限りシンプルにおやりなさい。ただし、シンプル過ぎてはいけません。(アルベルト・アインシュタイン)
- ソフトウェアは他のもの、いつか捨てられてしまうものと何か違うのだろうか？ 石鹸の泡のようなものと思うしかないだろう。(アラン・パリス)
- 簡潔さを通して美しさを求めなさい。

レッスンはこれにておしまいとします。後はあなたが心しておやりなさい。

もっと具体的なヒントが欲しい方のために、大きく3つに分けてアドバイスをします。

プログラムの解析

プログラムの動きの深いところまで見抜こうと思ったら、ひとつの方法としては例3-2でやったように、計測の仕組みを取り付けて代表的なデータで実行してみます。一方で、全体としてのプログラムよりも、特定の観点に関心があるという時もしばしばあります。例えば今回も、クイックソートの平均比較回数だけを考慮して、他の点はすべて無視しました。

セジウィック (Sedgewick, 「クイックソートプログラムの解析」、Acta Informatica, Vol. 7) は、必要とする記憶領域など、クイックソートの実行時に変化するさまざまな要素について調べています。中心となる問題に集中することで、(当面は)プログラムの他の観点を無視できます。

私は、論文「応用アルゴリズム設計のケーススタディ」(IEEE Computer, Vol. 17, No. 2) の中で、単位正方形内の N 点を通る巡回セールスマン問題の近似解を見つけるためのストリップヒューリスティックの実行性能を評価する問題に、どんなふうになり向かったか述べました。私は、これを解く完全なプログラムは100行程度であると見積もりました。この章で見てきたのと同様のやり方で進めて、最後は12行ほどのシミュレーションプログラムが得られ、それで十分な精度が得られました。ときに、そのシミュレーションを完了した後で、私はベアウッド (Beardwood, 「多点を通る最短経路」、Proceedings of Cambridge Philosophical Society, Vol. 55) でらが、私の行ったシミュレーションの内容を2重積分で表現しているのを見つけました。ですから数学的には、この問題は20年ぐらい前に解かれていたわけです。

小片のコード

私はプログラミングは実用的なスキルだと信じており、「実用的なスキルは、真似をすることと練習することで獲得する」というポリヤの意見に賛成です。美しいプログラムを書くことに憧れるプログラマは、美しいプログラムを読まなければならないし、自分でプログラムを書くときには学んだテクニックを真似ようとしてみるべきです。練習の場として最も有効なのが、だいたい12行から24行程度の小さなコード片です。

『珠玉のプログラミング』の第2版の準備を進めるのは、困難な仕事でしたが、とても楽しいことでした。ことごとくすべてのコードを実装した後、本質へと切り詰めるために労力を惜しみませんでした。私の望みは、私があコードを書いて楽しんだのと同じくらいに、他の人々があのコードを読んで楽しんでくれることです。

ソフトウェアシステム

ここでは私は小さいタスクの詳細について熟考する話をしてきましたが、そこに現れている原理は小さいコード片だけでなく、大きなプログラム、巨大なシステムにも通用すると信じています。パルナス (Parnas, 「ソフトウェアを拡張と短縮が容易なように設計する」IEEE Transactions on Software Engineering, Vol. 5, No. 2) は、本質に向かってシステムを削っていく手法について述べています。すぐに使えることとしては、トム・ダフ (Tom Duff) の深い洞察を覚えておいてください。「可能な時はいつでも、コードは盗みなさい。」

謝辞

ダン・ベントリー (Dan Bentley)、ブライアン・カーニハン (Brian Kernigham)、アンディ・オラム (Andy Oram)、デビッド・ワイス (David Weiss) の洞察力のあるコメントに感謝します。