キュート・アルゴリズム

稲葉 一浩[a)]

概要：アルゴリズムやデータ構造は，速度やメモリ消費など効率面での評価が第一になされることが多いが，「美しさ」もまた，その価値を計る重要な指標である．なかでも，神秘的・魔術的な美しさではなく，わかりやすい美しさ，言い方をかえれば，キーアイデアさえつかめば誰でもいつでもどこでも応用できる類いの，「近寄りやすい・親しみやすい」美しさには価値があろう．本稿では，筆者が今までに出会ったアルゴリズムの中から，そのような親しみを感じたアルゴリズムを紹介し，また，いかにそれらのアイデアに親しんで異なる問題に転用したかの実例を論ずる．

Cute Algorithms

Kazuhiro Inaba[a)]

Abstract: "Beauty" is yet another good measure for evaluating the importance of an algorithm, in addition to the widely used measures like time or memory consumption. In particular, I would like to emphasize the value of *cuteness*, a kind of beauty that feels familiar, easy to approach, and easy to implement and apply its core idea once I hear about the algorithm. In this article, I introduce several algorithms that I felt *cute*, and then show some instances of how I applied the idea to other problems.

1. はじめに

アルゴリズムやデータ構造は，多くの場合，ある与えられた特定の問題を解く計算手順として導入される．たとえば，グラフ構造上の最短経路を求める問題の解き方としてDijkstra の方法 [3] が提示され，あるいは値の配列を高速に並び替える方法としてクイックソート [5] が提示される．そして，重要なアルゴリズムは，しばしば非常に広い応用を持つ．「このアルゴリズムは，こんな問題にも使うことができたのか！」という驚きに出会うのは，プログラマにとって楽しい経験と言える．

しかし，もっと大きな楽しさを感じるのは，「このアルゴリズム**の考え方**は，他の問題にも応用することができる！」と気づいたときである．単にアルゴリズムをそのまま便利に使えるだけでなく，その神髄となるアイデアを我々に応用させてくれる，そのようなアルゴリズムは非常に魅力的だ．

以下では，筆者が経験したアルゴリズムの考え方の応用について，三つの事例を紹介する．

a) kiki@kmonos.net

2. ソート

2.1 比較ソートの計算量

アルゴリズムの代表例といえば，順序の定義された値の配列を順序通りに並べ替える「ソート」の問題をまず挙げることができる．ソートを行うアルゴリズムは数多く知られているが，この問題の特徴的な点の一つは，最も速いアルゴリズムでも最低これだけは時間がかかってしまう，という計算量の下限が証明されていることである．

定理 1. データの大小比較のみによってソートを行う場合，n 個のデータをソートするには，最悪時に $\Omega(n \log n)$ 回の比較が必要である．

証明． データの並べ方は $n!$ 通りの可能性がある．x 通りの可能性がある状況で 1 回の大小比較を行うと，可能性の空間は二分割されるが，悪い方の結果が返った場合残る可能性は $\frac{x}{2}$ 通り以上である．これを繰り返して可能性を 1 通りに確定するには，$\log_2(n!) = \Omega(\log(n^n)) = \Omega(n \log n)$

回の比較が必要. □

この定理によって，マージソートやヒープソートのような，実際に比較回数を $O(n \log n)$ 回に抑えることができるアルゴリズムは，ある意味で「最速」のソートアルゴリズムと呼ぶことができる．

自明でない計算量の下限が，このように簡潔な考え方で示されている問題は数少ない．さて，この美しい手法を，他に適用できないものだろうか．

2.2 最遅ソートの計算量

漸近的な計算量の意味で最も速いソートの存在が証明されているとなると，逆に，最も「遅い」，最も多い回数比較を行うソートアルゴリズムは何であるか，という疑問が自然と持ち上がる．この問題について考えてみよう．

ただし，ただ単純に遅くするだけなら，意味のない比較を繰り返すことでいくらでも回数を増やすことができてしまう．例えば以下の疑似コードは，任意の計算可能関数 f に対して $f(n)$ 回以上の比較演算を実行するソートアルゴリズムである．

```
function cheat_slow_sort(array) {
  for(i=0; i<f(array.length); ++i) {
    if(array[0] < array[1]) { /* nothing */ }
  }
  quicksort(array);
}
```

このような「狡い」低速化を排除するため，ここでは比較の回数として，比較によって情報が得られる，意味のある比較のみを数えることとする．すなわち，一度 a と c というデータを比較した以降は，a と c の比較は 1 回の比較と数えない．また，既に $a < b, b < c$ という結果が判明しているときは，a と c の比較結果も定まっているので，このような比較も無意味な比較とみなし，カウントしない．

例: バブルソート

```
function bubble_sort(array) {
  for(i=0; i<array.length-1; ++i) {
    for(k=1; k<array.length; ++k) {
      if(array[k-1] > array[k])
        swap(array[k-1], array[k]);
    }
  }
}
```

教科書的な遅いソートと言えばバブルソートが代表的だが，多少驚くべきことに，意味のある比較の回数を考えると，バブルソートは，実は速いアルゴリズムである．**最悪**のケース，つまり一番速く比較が終わってしまうケースでは入力がソート済みのときである．この場合は要素の移動は一切発生せず，最初の一回目のループで隣り合う要素を全て比較して，全てのデータの順序を確定してしまう．つまり，バブルソートは最悪 $O(n)$ の「速い」ソートであり，単に比較の回数だけを数えると $\Omega(n^2)$ に見えるのは，意味のない比較を繰り返し時間稼ぎをしているに過ぎない．

例: 挿入ソート

```
function insertion_sort(array) {
  for(i=1; i<array.length; ++i) {
    for(k=i-1; k>=0; --k) {
      if(array[k] < array[k+1])
        break;
      else
        swap(array[k], array[k+1]);
    }
  }
}
```

遅いソートのもう一つの代表例である挿入ソートも，ソート済みの入力に対しては最悪時には $O(n)$ 回の比較で処理が完了してしまう．

とはいえ，嬉しいことに，挿入ソートは平均的には遅いソートである．上記の擬似コードのように挿入位置を線形探索で探す場合，平均で，挿入済みの部分の長さの半分の位置まで比較による探索を行うことになるから，平均すると，ソート処理全体で $\frac{1+2+\cdots+n-1}{2} = \Omega(n^2)$ 回の比較が行われる．また，ここには意味のない比較は含まれていない．

例: クイックソート [5]

```
function quick_sort(array) {
  if(array.length > 1) {
    // 配列を array[0] 未満，array[0]，以上
    // に並べ直し，array[0] の位置を返す.
    p = partition(array, array[0]);
    quick_sort(array[0..p));
    quick_sort(array[p+1..$));
  }
}
```

少し考えるとわかるが，クイックソートは無駄な比較を行わないので，意味のある比較の回数の見積もりは，普通に比較の回数を数える時と同じ方法で評価することができる．そして，よく知られているように，クイックソートは常に $\Omega(n \log n)$ 回以上の比較を行う．従って，クイックソートは，最悪（繰り返すが，ここでは，一番速くソートが完了してしまうケースのこと）でも $\Omega(n \log n)$ の，安定して遅いソートアルゴリズムである．

速いソート法として知られるクイックソートが，意味のある比較の回数の意味で，同時に遅いアルゴリズムでもあるという事実は興味深い．平均の回数では，これもよく知

られているように，クイックソートの比較回数は $O(n \log n)$ で抑えられる．従って，平均的には，クイックソートは挿入ソートよりも速い．

	「最悪」	平均
バブルソート	n	?
挿入ソート	n	n^2
クイックソート	$n \log n$	$n \log n$

さて，ここまでいくつかの具体例を見てきたが，これらより遅いアルゴリズムを構成することは，可能だろうか．ここで，比較ソートの計算量の下限を与える証明の考え方を，ほぼそのまま転用することができる．

定理 2. データの大小比較のみによってソートを行う場合，n 個のデータをソートするには，最悪時には $O(n \log n)$ 回しか意味のある比較を行うことはできない．

証明. データの並べ方は $n!$ 通りの可能性がある．x 通りの可能性がある状況で 1 回の大小比較を行うと，可能性の空間は二分割されるが，小さい方の結果が返った場合残る可能性は $\frac{x}{2}$ 通り以下である．これを $\log_2(n!) = O(\log(n^n)) = O(n \log n)$ 回繰り返すと，1 通りに確定してソートが完了してしまう． □

すなわち，$\Omega(n \log n)$ のクイックソートは，「最悪」ケースに関しては，漸近オーダの意味で，最も遅いソートと呼ぶことができる．

また，最大で行える比較回数の上限は明らかに，n 個のデータから 2 つを取り出す $\frac{n(n-1)}{2}$ 通りであるから，平均の評価では，挿入ソートの $\Omega(n^2)$ は可能な限り最も遅いオーダを達成している．

2.3 最遅ソートの具体的な構成

残る疑問は，最悪 $\Omega(n \log n)$，平均 $\Omega(n^2)$ を同時に達成するソートアルゴリズムは存在するか？ という点に絞られる．これは，二つのソートアルゴリズムを組み合わせることで得られる．すなわち，以下のアルゴリズムである．

```
function slowest_sort(array) {
  quick_sort(array[0..array.length/2));
  insertion_sort(array);
}
```

まず配列の前半をクイックソートする．この部分だけで，まず確実に $\Omega(\frac{n}{2} \log \frac{n}{2}) = \Omega(n \log n)$ 回の意味のある比較を行うゆえ，最悪ケースには対応できている．配列の後半の要素は，クイックソート済みの領域へ，挿入ソートで挿入していく．この部分で，平均 $\Omega(n^2)$ 回の比較回数を確実に消費することができる．

「クイックソートと挿入ソートを組み合わせる」という手法は，実用的に速いソートを実現する技法としてよく使われているものであるが，この組み合わせ方を敢えて捻ることで，最も遅いソート法を作ることにも適用できるのは，またこれも興味深い点である．

2.4 最遅ソートの応用

計算機上でデータをできるだけ遅く並び替える，という行為には，おそらく実用的な意味は無い．つまり最遅ソートは，基本的には，言ってしまえばジョーク・アルゴリズムである．

しかしながら，見方を変えると，最遅ソートの目標である「意味のある比較演算をできるだけ増やす」という考え方は，意味のある問題への応用も考えることができる．次のような問題を考えてみよう．

問題: n 人の選手の総当たり戦をする．選手達の強さはほぼ全順序関係に近いとしよう．総当たりの計 $\frac{n(n-1)}{2}$ 戦の試合の順番を，消化試合，つまりやる前から順位の変動に影響しないことが確定している試合ができるだけ少なくなるように，スケジューリングせよ．

これは，最遅ソート問題そのものである．

2.5 質疑応答

発表後の質疑の時間および，終了後の個人的な会話で，以下のような質問があった．

Q: まず前半クイックソートを行い後半で挿入ソートに切り替えるという方法では，前半のクイックソートの影響で，後半の挿入ソートの段階では $O(n^2)$ 回の比較に無駄な比較が含まれてしまっているのではないか．
A: 時間的に前半クイックソート，後半挿入ソートと分けるのではなく，空間的に，つまり配列の前半をクイックソートし，後半を挿入ソートする，という方法であることに注意されたい．この場合，前半のクイックソートの段階では後半の要素は一切見ないため，後の比較と重複することはない．

Q: 選択ソートは同じ比較を二度行うことはない．これは平均・最悪ともに $\Omega(n^2)$ 回の無駄ではない比較をしていることにならないか．
A: 既に値 a と c を一度比較済みのときだけでなく，比較の結果 $a < b$ と $b < c$ が判明しているときに，a と c を比較するのも無駄である，と今回は見なしている．この定義の下で，選択ソートも，やはり無駄な比較を行う可能性がある．

Q: 例えば挿入ソートの際に，既ソート済みの部分の性質を利用すれば，線形探索ではなく二分探索で挿入点を決めることができ，この場合比較回数は $O(n \log n)$ になる．このような高速化が可能とわかっているのに敢えて線形探索

で挿入ソートを行うのは狡くないと言えるのか.

A: 非常に本質的な指摘．今回の定義では狡くないが，確かに定義の仕方によっては狡いと定義されることもあるだろう．実は，計算量の解析以上に，いかに巧妙に「狡くなさ」を定義するかが最も面白い問題である．本文で例にあげたような明らかに狡い，任意の計算可能関数の回数同じ比較を繰り返すようなソート法を排除しつつ，Stooge sort[*1] や Bogosort[*2] などの既存の自然な遅いソート法を許す定義を形式化することが，将来の研究課題である.

最後の質疑に関連する研究として，Broder と Stolfi [1] により **Simplexity** という概念が提唱されている．なんらかの意味で最終的な解の導出に近づいている処理は無駄な処理ではないと認める，という基準で「狡くない」遅さを規定するもので，この考えに基づいて Slowsort と呼ばれる $\Omega(n^{\log(n)/(2+\epsilon)})$ の時間がかかるソートアルゴリズムが提案されている．ただし，彼らの定義はフォーマルな定式化はなされておらず，曖昧さが残る.

3. ブートストラッピング

3.1 非破壊的キュー

次のトピックとして，Okasaki [9] による**非破壊的キュー**の実装を紹介する.

キューは，値を格納する手続き *push* と値を取り出す手続き *pop* を持ち，値は入れた時と同順で取り出せる (FIFO) という性質を持つデータ構造である．広く普及しているキューの実装の一つでは，値を二重リンクリストの形で繋いで表現する.

```
class Queue<T> {
  class Node { T value;   Node prev, next; }
  Node head, tail;
}
```

このデータ構造は，破壊的に更新される．push や pop の際には，`prev` や `next` ポインタの指す先を書き換えることで，リストに要素を繋いだり外したりするのである．一度操作を行うと，操作の前のキューの状態を表すデータ構造は，メモリ上のどこにも存在しなくなる.

これに対し，書籍 [9] は，様々な**非破壊的**なデータ構造を提案している．非破壊的なデータ構造では，操作を行うと必ず，元の構造はそのまま残したまま，変更後の状態を表す新しい構造が作られる．新しい構造からポインタで古い構造の一部を指してデータを共有することはあるが，古い構造に破壊的な変更が加えられることは決してない.

可能な操作に制約が加わる分，効率の良い非破壊的データ構造を作るのは難しい．一方で，その困難を補う数多くの技法が開発されている．キューを題材に，いかに効率的な非破壊的データ構造が構成されるか，簡単に概要を記す.

まず，破壊的操作を取り除くために，二重リンクリスト１つではなく，単方向リンクリスト２つでキューを表現する．二重リンクリストは破壊的操作抜きでは何もできないが，単方向リンクリストは，先頭への要素の追加（Lisp の `cons`）や，先頭要素を抜いた残りのリストの参照（Lisp の `cdr`）は非破壊的に実現できるというのがその理由である.

```
class Queue<T> {
  class Node { T value;  Node next; }
  Node for_pop, for_push;
};
```

push 操作は，単純に，push 用のリストの先頭に要素を追加（`cons`）すればよい.

```
Queue<T> push(Queue<T> q, T v) {
  return Queue(q.for_pop, Node(v, q.for_push));
}
```

pop は，pop 用リストが空でなければ先頭要素を取り出す（`car`）が，空だった場合は，その瞬間に push 用リストを逆転して pop 用リストとする.

```
Pair<T, Queue<T>> pop(Queue<T> q) {
   Node po = q.for_pop, pu = q.for_push;
   if( po == null )
     po = reverse(pu), pu = null;
   return Pair(po.value, Queue(po.next, pu));
}
```

これによって，データ構造を破壊せずに，ほとんどの場合に $O(1)$ 時間で push と pop ができるキューが完成した．欠点は，pop 用リストが空になってしまった時に限り，それまでに push してきた要素のリスト全体を reverse する $O(n)$ 時間が必要となってしまうことだ.

reverse のコストを和らげる基本的なアイデアは，必要になったギリギリの瞬間に全体を reverse するのではなく，適度に分割し，こまめに push 用リストの中身を reverse して pop 用リストに移しておく，という方法である.

しかし，このアイデアには大きな難点がある．「pop 用リストに移しておく」時には，pop 用のリストの「末尾」に新しいリストを付け足す必要があるのだ．ところが，単方向リンクリストである pop 用リストは，リストの先頭に対する操作以外を非破壊的に行うことはできない.

ここで登場するのが，**ブートストラッピング** である．今の難点をもう一度振り返ってみると，「先頭から取り出す用途の pop リストに，末尾からの付け足しもしたくなった」という状況である．先頭から出し，末尾から入れる，これはまさにキューがあれば実現できる要求だ．というわけで，pop 用リストに reverse したリストを実際に付け足すのではなく，付け足し待ちのキューを用意して，そこに push すればよい．完成したデータ構造は以下のようになる.

[*1] http://en.wikipedia.org/wiki/Stooge_sort
[*2] http://en.wikipedia.org/wiki/Bogosort

```
class Queue<T> {
  class Node { T value;  Node next; }
  Node         for_pop;
  Queue<Node> for_pop_queue;
  Node         for_push;
}
```

キューを作ってるまさにその最中に，内部でキューを使う，というのは不思議である．しかし，内部では要素のキューではなく，要素をいくつかまとめたリストのキューを構成するため，元のキューより必ずサイズの小さなキューになっている．従って，この再帰構造はどこかで止まり，実際にプログラムとして動作するのである．

大きなコンパイラ自身を最初にコンパイルするための小さなコンパイラや，大きな OS 自身を起動するプログラムを動かすための小さな OS やブートローダ，といった**ブートストラッピング**の考え方は，計算機アーキテクチャではしばしば現れる．これがデータ構造やアルゴリズムの場面でも有効に活用できる，という事実が，私には面白く感じられた．

3.2 迷路の最短経路

ブートストラッピングの手法を他のアルゴリズム問題にも応用してみよう．

問題: 与えられた迷路を最短で抜ける経路を求めよ．

これは典型的な問題で，Dijkstra のアルゴリズム [3] や，迷路の形が単純ならば簡単に幅優先探索などで解を求めることができる．これに少しだけ追加の条件を加える．

問題 (改): 与えられた迷路を最短で抜ける経路を求めよ．ただし，返す解が最短であることの検査ルーチンも書きなさい．

問題によっては，条件を満たす解を求めることよりも，具体的な解が条件を満たすかのチェックを行う方が簡単で，このような検査をすぐに書くことができる．たとえば，「ソート関数の返す結果の配列が順序通りに並んでいることを検査せよ」という要求であれば，実際にループで隣り合う要素をすべて比較して検査すればよい．

さて，迷路の最短経路問題はどうだろうか．言葉の定義通りに，返す経路が最短である，と記述すると以下のようになる．

```
Route shortest(Maze m);
bool check(Maze m, Route r) {
  return r.len == shortest(m).len;
}
```

しかし，これではトートロジーである．shortest 関数が仮に間違っていても，常に検査は true を返してしまう．これは意味がない．

とはいえども，最短経路である，という性質を最短経路を実際に求めることをせずに検査するのはなかなか難しい．全ての経路を列挙してその中で一番短いことを確認するなどの方法は，計算量が爆発してしまい現実的ではない．

そこで**ブートストラッピング**の出番である．検査付き最短路検索ルーチンを実装する際に，一回り小さい検査付き最短路からブートストラップするのだ．検査付き最短路が適用できる「小さい」問題を作り出すために，最短路のある性質を利用する．最短路ということは，脇道にそれると距離が同じか長くなるということである．検査中のルートの一部を通行止めにして，強制的に脇道を通る最短路を計算させ，それがどうやっても元より短くなっていなければ，元のルートが最短であったことがわかる．

図 1 最短路には短い近道が無い

擬似コードで書くと，以下のようになる．

```
bool check(Maze m, Route r) {
  for( p,q ← r 上の場所のペア ) {
    m' = m の p から q までを全て埋め立てた迷路
    if( safe_shortest(m').len < r.len )
      return false;
  }
  return true;
}
```

ここで，safe_shortest は次で定義した，確実に最短路であることをチェック済みの最短路を返す関数である．

```
Route safe_shortest(Maze m) {
  Route r = shortest(m);
  if(!check(m,r)) throw Exception;
  return r;
}
```

途中を通行止めにした迷路は，動ける領域が小さくなっているので，「小さい」迷路であり，従ってこの再帰は止まることが証明できる．また，「元々の shortest 関数が，距離はさておき，迷路に解があるか無いかは正しく判定できている」という前提の元で，shortest が本当に最短路を求めていたか否かに関わらず，safe_shortest が例外を投げずに答えを返すときは必ず最短路を返している，ということが帰納法で証明できる．詳細と，Ruby による完全な実装を www.kmonos.net/wlog/105.html#_2232100114

に掲載している．

このブートストラップで特筆すべきは，元々の「間違っているかもしれない `shortest` 関数」を足がかりに，他の助けなしに，「間違うかもしれないが，少なくとも間違った時はそのことを自己検出して例外を投げる安全な `safe_shortest` 関数」へと一段上の安全性へグレードアップを遂げている点である．

3.3 質疑応答

非破壊的キューに関して，以下の質問があった．

Q: 非破壊キューを pop したときに何が起こるのか詳しく．
A: pop された値と，pop 後のキューを指すポインタの両方が返される関数になる．pop 前の古いキューも，ポインタを残しておけば変更前がそのまま参照できる状態で残る．

Q: そもそも非破壊的データ構造にはどのような用途があるのか．
A: Haskell のような純粋関数型言語で使える，という用途の他に，手続き型言語でも，データ構造の巻き戻し，バックトラックを頻繁に行いたい時に便利．例えばゲーム探索の最中の中間データ構造．あるいは，コンパイラの変数と型の対応表を非破壊的 map で持っておくと，スコープの出入りで変数セットを戻すのが非常に簡単．

Q: このように頑張って計算量のオーダを破壊的なものに合わせても，結局定数係数が遅いのであまり嬉しくないのではないか．
A: その通り遅いので，オーダを完全に合わせるデータ構造は，実用よりは理論的興味が先行していることが多い．しかし，今回紹介したブートストラッピングキューは，そのカテゴリには当てはまらない．この方法では計算量は $O(1)$ にならず，$O(\log^* n)$ という計算量に留まる．しかし，実装はどの $O(1)$ キューよりもシンプルなため，実用上は最も速い非破壊的キューになっているという実験結果がある．

Q: for_pop の先頭の要素に対して操作を行うためには，キューのメンバは `(Queue<Node>, Node)` ではなく `(Node, Queue<Node>, Node)` と先頭のリストだけ特別扱いが必要ではないか．
A: その通り．口頭発表の際の説明には誤りがあった．

Q: `Queue<T>` の中で `Queue<Node<T>>` を使うような多相な再帰は言語によっては実装が難しいのでは？
A: その通り．C++や D 言語のテンプレートでは，少なくともそのままの形で記述できない．Java の Generics などでは問題ない．

最短性を保証する迷路ソルバーについては，以下のような質問があった．

Q: 途中で例にあがったソートの結果の検査について，ソートの検査は順序通り並んでいることよりも，元の配列の並び替えになっていることの検査の方が難しい．
A: その通り．

Q: このアルゴリズムは正解の経路が 1 つという仮定を暗に置いているか？
A: いない．複数経路があっても問題ない．ただし，迷路が解けるときは，最短とは限らずとも何かを経路は返す，という仮定を置いている．

4. 計算過程のデータ化

4.1 正確実数計算

3 つめのトピックとして，**計算過程のデータ化** という技法を紹介する．この技法自体は古くから様々な形で使われている．例えば C++ の行列計算ライブラリの高効率化で有名な Expression Template [11] はその一つだ．最近では，Henglein ら [4] がソートやクエリの実装に対して，このアイデアを明示的に取り上げて研究を進めている．

この技法の具体例として，ここでは RealLib [8] というライブラリでの使われ方を紹介する．RealLib は，**正確実数計算** *(Exact Real Arithmetic)* と呼ばれる機能を実装した C++ のライブラリの一つである．このライブラリは，誤差の無い，完全に正確な実数を表す `Real` 型を提供する．たとえば

```
Real sq3 = cos(Pi/3-Pi/2)*2;
Real x   = 1;
for(int i=0; i<4; ++i)
  x *= sq3;
cout << setprecision(12345) << x << endl;
```

このコードは，三角関数，円周率，四則演算やループを使って $(\sqrt{3})^4 = 9$ を計算しているが，この結果，小数点以下 12345 桁まで正確に $9.000\cdots000$ が出力される．多くの言語で標準的に提供されている多倍長十進数（BigDecimal）と異なり，出力の際にどれだけ大きい桁を指定しても誤差が乗ることはない．

無限にデータを覚えておくことはできないにも関わらず，いくらでも正確な値を出力できるのは，一見すると不思議に思えるかもしれない．これを実現するのが，計算過程のデータ化である．RealLib が，$\pi/3$ を計算した結果として保持するのは $1.5707\ldots$ という数ではなく，単に $\pi/3$ という式を表現する木構造である．先ほどのプログラム全体の結果は，9 という数値ではなく，図 2 のような木構造（正確には，DAG 構造）になる．

この値を実際に文字列として出力しろと言われて初めて，式を評価する．その際に，出力すべき精度は与えられているので，必要なだけ高い精度で式全体を計算することができる．

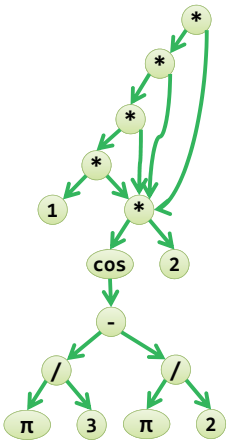

図 2　Real 型の内部

計算を指示されたその場で具体的な値を求めるのではなく，式の構造をそのまま覚えておき，計算は後で行うことの利点は，後で**値を使う時の文脈**の情報を利用できることにある．RealLib の場合で言えば，どの精度まで計算すれば十分かという情報である．

4.2　木構造クエリ

後で使われる際の文脈情報を利用できる，という特徴は，他の場面でも有効に活用できる．ひとつの例として，木構造データに対するクエリ言語の実装への適用について触れる．詳細は論文 [7] を参照して欲しい．

木構造に対するクエリ言語，例として「XML のノードの組 (x, y) で，x は <a> 要素であり，x の href 属性と y の id 属性が等しく，x と y の共通祖先に <p> 要素があるものを全て求めよ．」といった問いに答える効率的なアルゴリズムの実装が目標である．基本的には，次の擬似コードのように，木構造上を再帰的になぞって x の候補，y の候補，組 (x, y) の候補，祖先要素チェック後の確定解の集合 (ans) を計算することになる．

```
Result query(Tree t) {
  // 簡単のため t は二分木とする
  L = query(t.left)
  R = query(t.right)
  x   = L.x ∪ R.x ∪
        (t.tag = "a" ? {t} : {})
  y   = ...
  xy  = {(a,b) | a ∈ x, b ∈ y,...}
  ans = L.ans ∪ R.ans ∪
        (t.tag = "p" ? xy : {})
  return {x, y, xy, ans}
}
```

この方法の効率面での難点は，二種類の理由で，必要のない解候補集合を生成してしまう可能性があることである．一つは，あとで空集合との積を取られるため（図 3 左），中間生成物の集合が中身に関わらず捨てられてしまう場合で，もう一つは，木構造を登り切った結果，解の条件を満たさないことがわかったため捨てられてしまう場合である（図 3 右）．

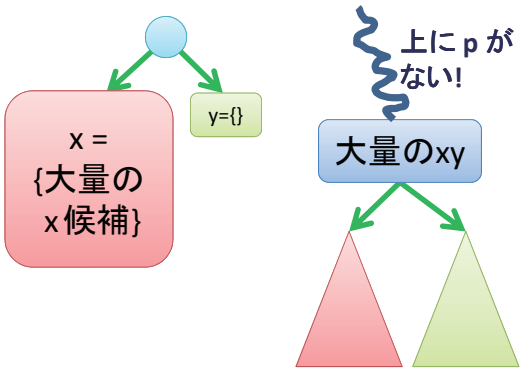

図 3　効率が不要に悪くなる例

どちらの場合も，単に木を再帰で一方向になぞるだけではなく，上下に何度かなぞって文脈情報を適切に前計算しておくことで，回避することは一応可能である．しかし，それでは，複数の再帰関数や，文脈情報を格納した再帰パラメタが必要となり，アルゴリズムが複雑化してしまう．

さて，お気づきと思うが，この問題はまさに，後で**値を使う時の文脈**の情報が必要となる種類の計算である．後で空集合と積演算を取ることにしか使われないなら，計算をする必要がない．後でそもそも使われないならば計算をする必要がない．このような状況にぴたりとはまるのが，**計算過程のデータ化** だ．今回の場合は，再帰によって集合の計算を行う際の集合演算の式を，そのまま式の構文木構造として保持すればよい．

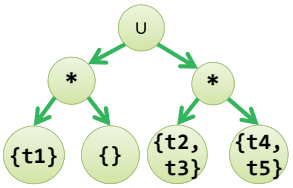

図 4　集合演算の式構造をデータ化

これによって，元のアルゴリズムの形には一切手を加えずに，上部の文脈情報を利用した最適化が可能になる．

なお，計算過程のデータ化によっ得られる利益の一部は，単純な遅延評価でも得られることがある．例えば RealLib の場合は，無限に精度を伸ばした無限リストとして実数を表現して実際の演算を行い，ただし評価戦略として遅延評価を採用する，という形でも同じことが実現できる．

単なる遅延評価と計算過程のデータ化の異なるところは，後者では，計算過程が明示的にデータ化されているため，追加的な操作を後から加えられるところにある．遅延

評価では，評価以外の操作を後で行うことはできない．木構造クエリの場合，空集合の判定は評価とは独立して別途行った方が効率がよく，これはデータ化によって初めて得られる利点である．また，クエリ結果の列挙や射影操作など，さらなる演算をデータ化をほどかずに実行し，時間・メモリともに効率を改善するなどの手法も研究されている [4, 7].

4.3 質疑応答

RealLib の実装方式について, 以下のような質問があった.

Q: たとえば $a-b$ を 80 桁の精度で求める必要がある場合，a, b をまず 80 桁 $+\alpha$ で求めておいて，桁落ちが発生して精度が足りないことが分かったら，α を大きくするようなことになっているのか？
A: その通り．予め必要な精度を見積もるのは難しいのでこのようになっている．足りなかった場合には，精度を倍々で増やしている．

Q: 0 に近い答えが出るような場合に，無限ループに陥って，結果がいつまでたっても出ないようなことがあるのではないか？
A: 四則演算など連続関数の計算ではそのような問題はない．不連続な関数，より具体的には，比較演算では指摘の通りの現象が発生する．等しい値同士を $<, >, =$ などで比較すると無限ループに陥る．

Q: 構成的数学の実数の構成を連想した.
A: 正しい．連続関数のみが正確に表現できるなどの特徴もそこから来ている．

計算過程を木としてデータ化するという技法全般について，以下のコメントがあった.

Q: Object Algebra [2], Polemy [6], Lightweight Modular Staging [10] などを連想した.
A: 参考になる.

木構造へのクエリへの応用について, 以下の質問があった.

Q: 実際の計算をデータ化して後回しにしても，空集合との掛け算で無駄になるなどの問題は，後で結局発生してしまい，同じ問題を先送りしているだけではないのか.
A: そうならないように，詳しくは，空集合かどうかの判定のみは最後まで遅延せずに別途先に計算しておく．任意の集合演算に対して空集合判定のみを先回りさせることは不可能なため，このデータ構造を使う際には，それが可能な演算のみを使う形で処理を行う必要があり，それが可能なようにクエリ言語とそのコンパイラを設計する必要がある [7].

謝辞 最遅ソートに関する議論は，Twitter 上での多くの方々との会話が元になっている．このテーマ自体を考えるきっかけは @y_benjo 氏のツイートから頂いた．平均，および平均最悪双方で最遅を達成するアルゴリズムの最初の具体例とその証明は，@rng_58 氏, @hos_lyric 氏, @oxy 氏, @xhl_kogitsune 氏によるものである．（本稿で紹介したバージョンはそこから更に簡略化されている.）また，最遅ソートの応用例に関しては，@args1234 氏との対話を通して発想が得られた.

参考文献

[1] A. Broder and J. Stolfi. Pessimal algorithms and simplexity analysis. *ACM SIGACT News*, 16:49–53, 1984. doi: 10.1145/990534.990536.

[2] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *European Conference of Object-Oriented Programming (ECOOP)*, pp. 2–27, 2012. doi: 10.1007/978-3-642-31057-7_2.

[3] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi: 10.1007/BF01386390.

[4] F. Henglein and K. Larsen. Generic multiset programming with discrimination-based joins and symbolic cartesian products. *Higher-Order and Symbolic Computation*, 23:1–34, 2011. doi: 10.1007/s10990-011-9078-8.

[5] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–16, 1962. doi: 10.1093/comjnl/5.1.10.

[6] K. Inaba. Polemy reference manual. `http://www.kmonos.net/repos/polemy/doc/tip/doc/`, 2010.

[7] K. Inaba and H. Hosoya. Compact representation for answer sets of n-ary regular queries. In *Conference on Implementation and Application of Automata (CIAA)*, pp. 94–104, 2009. doi: 10.1007/978-3-642-02979-0_13.

[8] B. Lambov. Reallib: An efficient implementation of exact real arithmetic. *Mathematical Structures in Computer Science*, 17:81–98, 2007. `reallib.sf.net`.

[9] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[10] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming and Component Engineering (GPCE)*, pp. 127–136, 2010. doi: 10.1145/1868294.1868314.

[11] T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995. `http://www10.informatik.uni-erlangen.de/~pflaum/pflaum/ProSeminar/exprtmpl.html`.