定番アルゴリズムを徹底理解!

ソートとサーチがすべての基本

斉藤 国博

このパートでは，プログラミングを勉強するうえで欠かせないアルゴリズムの中でも定番中の定番を紹介します。ソート（並べ替え）やサーチ（検索）などの機能は今では標準のライブラリとして提供されています。実用的なプログラムを作るときにそのものずばりをいちいち書く機会は少ないかもしれません。しかし定番のアルゴリズムは，様々に形を変えて普段のプログラミングに登場します。

解説を読んで仕組みがわかったら，ぜひそれをプログラムにしてみてください。読んだだけではプログラムを書けるようにはなりませんし，プログラムを書いてみて初めて，実は十分に理解できていなかったと気付くことがよくあります。しかもアルゴリズムは特定のプログラミング言語に依存しないので，一度身に付ければ，後でどんな言語を学ぶ場合でも役に立ちます。

1番目から6番目まではソートのアルゴリズム，7番目から9番目まではサーチのアルゴリズムです。一つひとつ作って動作を確かめたら，大量のデータを与えて性能を比較してみましょう。安直な方法では解決できない問題でも優秀なアルゴリズムがあればいとも簡単に解決できることを実感できます。

1 挿入ソート

トランプ手札の並べ替えを思い出せば簡単

「ソート（sort）」とは，たくさんのモノがあるときに，それらをある特定のルールに従って，例えば値の小さいものから大きいものに順に並べ替えることです。実生活では，トランプの手札を並べ替えるときや，お年玉付き年賀ハガキを番号の下2ケタで並べ替えるといった作業が該当します。

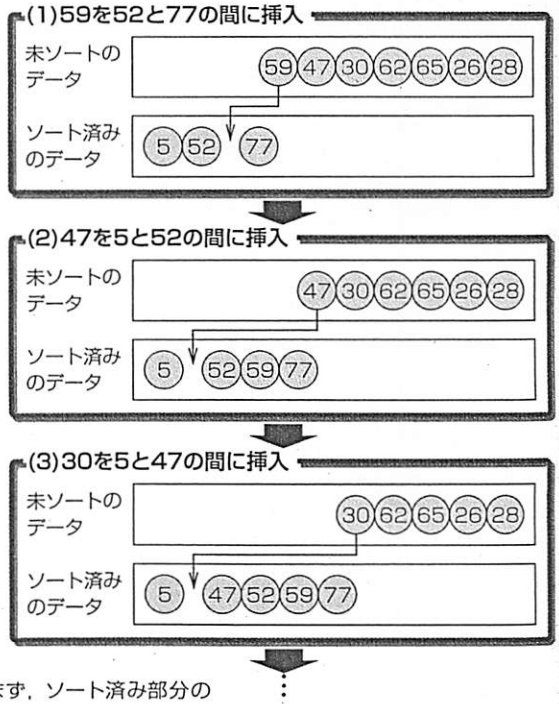
こうした並べ替え作業をする方法はいくつもあります。多くの人がまず思いつくのは，並べ替える対象を一つ選んで，それを適切な位置に挿入していくことでしょう。トランプの手札を並べ替える場合であれば，例えば，一番右のカードを抜いて，カードの数字に応じて適切な場所に差し込んでいく，という作業を何度か繰り返せば良いのです。

これを実現するアルゴリズムが「挿入ソート」です。つまり，未ソートのデータを一つひとつ，ソート済みの列の適切な位置に差し込んでいくわけです（**図1**）。アイデア自体はとても簡単ですね。

でも挿入ソートのアルゴリズムをプログラムで記述しようとすると少し複雑になります。ハガキやトランプなら無造作に差し込んでしまえばいいのですが，プログラムではデータをいちいち格納したり，取り出したりする必要があるからです。こうしたプログラムでは，ソートの対象にするデータを配列に格納するのが一般的です。配列のデータを並べ替える場合には，挿入先の場所（配列要素）を空けなければなりません*1。

図2は配列のデータに対して挿入ソートを実行している途中の様子です。未ソート部分の先頭のデータ（ここでは47）を取り出して，それをソート済み部分の適切な個所に挿入しようとしています。まず，ソート済み部分の末尾のデータ（77）と比較します。取り出したデータのほうが小さいので，末尾のデータを今までそのデータが保存されていた個所にコピーして場所を空けます。次に，対象を一つ先頭に向かってずらし，59と比較します。すると，また自分のほうが小さいので，仮の挿入場所を一つずらします。4回目

図1●挿入ソートのアルゴリズム。未ソートのデータを，ソート済みの列の適切な個所に挿入していく



2 シェルソート

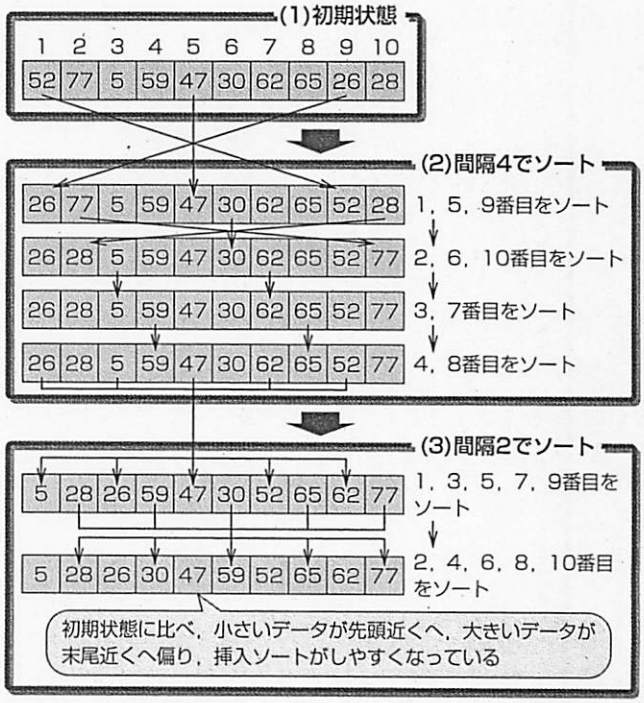
“下ごしらえ”で挿入ソートを高速化

シェルソートは挿入ソートの改良版と言えるアルゴリズムです。挿入ソートが，ソート済みのデータに対しては高速であるという性質を利用して，挿入ソートの前の下ごしらえで大まかに並べ替えの済んだデータを作ることで高速化を図ります。

具体的には，単純な挿入ソートをする前に，一定間隔で取り出したデータについて挿入ソートを行います（**図3**）。最初は間隔を広く取り（図3の(2)），徐々に狭めていきます。すると，小さいデータが先頭のほうに，大きいデータが後ろのほうにそれぞれ偏っていきます。最終的には間隔を1として単純な挿入ソートを行います。このときには，ソート済みとまではいかなくても，初期状態より整理された状態で挿入ソートができるわけです。

このアルゴリズムがうまく働くかどうかは，間隔の取り方にかかっています。うまく偏りを出すためには，なるべく共通の約数を持たない数（互いに素な数）を使い，なおかつ間隔の変化を小さくしすぎないのが良いとされています。よく知られているのが，$a_{k+1}=3a_k+1$で計算できる「1，4，13，40，121…」という数列の値を，データの個数に応じて大きなものから使っていく方法です。

図3●シェルソートのアルゴリズム



の比較対象となる5は自分より小さいので，そこに挿入すればよいことがわかります。コードは**リスト1**のようになります。

挿入ソートはわかりやすいのですが，ソートの対象にするデータの個数が増えると，計算量（繰り返し回数）が爆発的に増えてしまいます。詳しい計算式は省きますが，繰り返し回数はデータ個数の2乗に比例します*2。つまり，データの個数が10倍になるごとに繰り返し回数がおおむね100倍になります。このように，繰り返し回数がデータ個数の2乗に比例するアルゴリズムには，どんな場面にでも対応できるだけの汎用性はありません。

ただし，挿入ソートは非常に高速に動作する場合があります。最も高速なのは，ソート済みのデータをソートしようとしたときです。

「そんなの当たり前じゃないの」という声が聞こえてきそうですが，そうではありません。そのデータがソート済みかどうかわからない場合には，ソートのアルゴリズムを適用して処理を行う必要があるからです。

挿入ソートではデータがソート済みの場合，挿入を試みる最初の比較（図2では47と77との比較に相当）で必ず自分より小さいデータが見つかり，その場で挿入場所が確定します。このため，（データ個数－1）回の繰り返しでソートが終了します。処理がシンプルであることも相まって，挿入ソートはこの場合に限って，一般的に高速とされるどのアルゴリズムよりも高速にソートできます。ソート済みのデータ列の末尾にデータを何個か追加して全体をソートし直す，といった場合も有効です。

図2●配列内で挿入ソートを実行する際の手順

リスト1●挿入ソートのコード（Visual Basic 2005）

```
For i = 1 To n - 1
  ' 挿入したいデータを待避する
  temp = data(i)

  ' 挿入個所が見つかるまでデータをずらす
  j = i - 1
  While temp < data(j)
    data(j + 1) = data(j)
    j = j - 1

    ' 先頭に達した場合は無条件で終了
    If j < 0 Then Exit While
  End While

  ' 待避してあったデータを挿入する
  data(j + 1) = temp
Next
```

*1 挿入に手間がかかるのはデータを配列に保持していることが原因で，アルゴリズム自体の問題ではありません。
*2 n個のデータそれぞれについて，平均n/4回の比較が必要になるため，計算量はnの2乗のオーダーになります。

3 選択ソート

知恵を絞れば初心者でも作れる

挿入ソートが実生活からの類推で思いつくのに対し，選択ソートはプログラミング言語を使ってなんとかソートを実行しようとしたときに思いつきやすいアルゴリズムと言えます。

プログラミング言語の文法を覚えたばかりで，アルゴリズムについて何の予備知識もない人が「配列のデータをソートするプログラムを作ってごらん」といわれたらどうするでしょうか。

繰り返しと配列をきちんと勉強した人なら，データの中から一番小さいデータを見つけては取り出して最後の一個まで並べていくという方法を思いつくのではないでしょうか（**図4**）。一番小さいデータを探す手順は簡単な繰り返しで実現できます。まず適当な1個を手元に取り出して，それ以外のすべてのデータと一つずつ順に比較し，より小さいものを手元に残すようにすれば良いのです。

リスト2●選択ソートのコード（Visual Basic 2005）

```
For i = 0 To (n - 1) - 1
  ' 未ソート部分の最小値の添え字を探す
  minIndex = i
  For j = i + 1 To n - 1
    If data(minIndex) > data(j) Then
      minIndex = j
    End If
  Next

  ' 未ソート部分の先頭と最小値を入れ替える
  SwapValue(i, minIndex)
Next
```

図4ではソート済みのデータを別のところに並べていくような描き方をしています。しかし実際のプログラムではソート対象のデータが入っている配列に，そのままソート済みの列を作ることができます*3。一番小さいものを選び出した時点でソート対象のデータが1個減るため，そのぶんの配列の要素を利用できるからです（**図5**）。

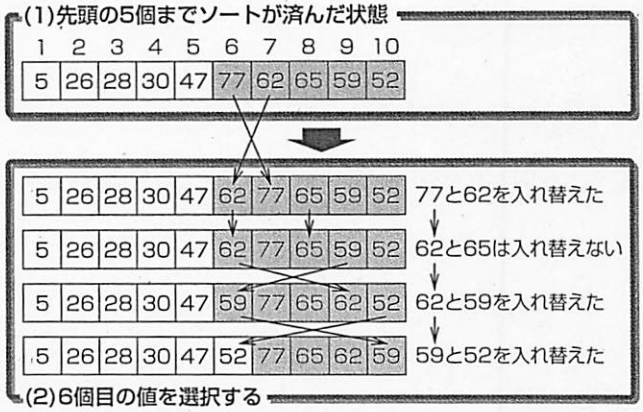


図5●配列内で図4のソートを実行する際の手順

これをちょっと改良したのがたいていの教科書に載っている「選択ソート」というアルゴリズムです。一番小さいデータを見つけるときに，いちいち手元のデータを取り替えていくのではなく，配列の添え字だけを覚えておいて，最後にデータを入れ替えます。例えば**リスト2**のようなコードで実行できます。

このアルゴリズムは単純であるぶん，膨大な計算量を必要とします。一番小さいものを選ぶ作業をソート対象のデータ個数だけ繰り返す必要があり，その繰り返し1回ごとに，すべての未ソートのデータを比較する必要があります*4。そのため，繰り返し回数はデータ個数の2乗に比例します。

また，挿入ソートと異なり，元のデータがどのように並んでいても，たとえソート済みであっても，データの個数に応じて決まった回数の繰り返しを行う必要があります。したがって，データの個数が十分に少ないとわかっている場合を除いて，ほとんど使われることはありません。

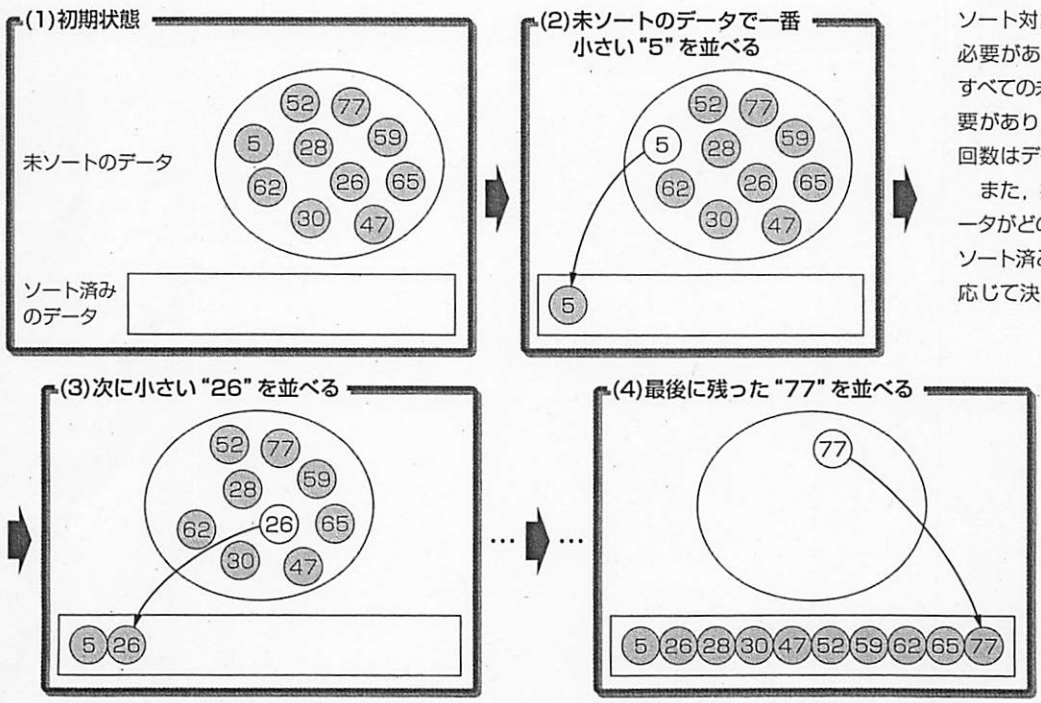


図4●単純なソート・アルゴリズム。データの中から最小のものを見つけ出して並べていく

*3 このように一定個数の作業領域を除き，データの格納領域内で並べ替えが完結するソートを「内部ソート」と呼びます。

*4 正確には（未ソートのデータ個数−1）回の繰り返しが発生します。

4 バブルソート

いかにも“アルゴリズムっぽい”方法

挿入ソートが実世界っぽく，選択ソートがプログラムっぽいというたとえで言うと，バブルソートは「いかにもアルゴリズムっぽい」とたとえられそうなアルゴリズムです。バブルソートの処理の単位は，隣り合った二つのデータを比較して，小さいものが先に来るように並べ替える，というものです。この処理を決まった順序で決まった回数だけ繰り返すと，いつの間にかソートが済んでいる，という魔法のようなアルゴリズムです。実際，バブルソートのプログラムを書くのは選択ソートや挿入ソートより簡単なくらいです。

「決まった順序と決まった回数」は，**図6**のようになります。つまり，配列の後ろから先頭に向かって，隣り合うデータ同士をそれぞれ比較，交換します。それが済んだら，先頭の1個を除いてまた同じように比較，交換を繰り返します。これをデータが2個残るまで実行すれば終わりです。

配列の後ろから先頭までの比較，交換を繰り返すと，その中で一番小さいデータが一番先頭に押し出されてきます。2番目以降の順序はバラバラですが，一番目がそのどれよりも小さいことは間違いありません。

そこで次は先頭の1個を除いた2番目以降のデータに対して同じ比較，交換を実行します*5。するとやはり2番目に小さいデータが2番目に押し出されてきます。つまり，2回の繰り返しで2番目のデータまでソートができました。これを最後の二つのデータを入れ替えるところまで繰り返せば全部のデータがソートされます。一番小さいデータが水中の泡のように上位に押し上げられてくることから「バブル（泡）ソート」というわけです。

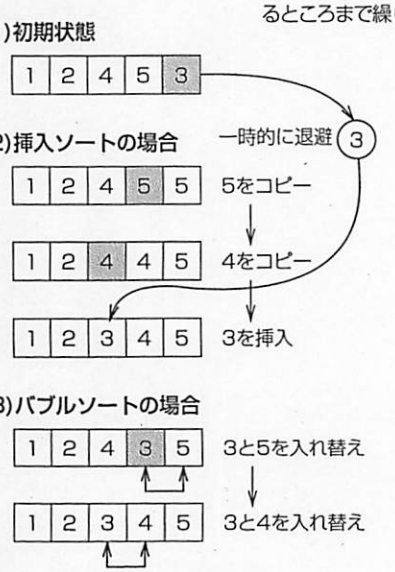
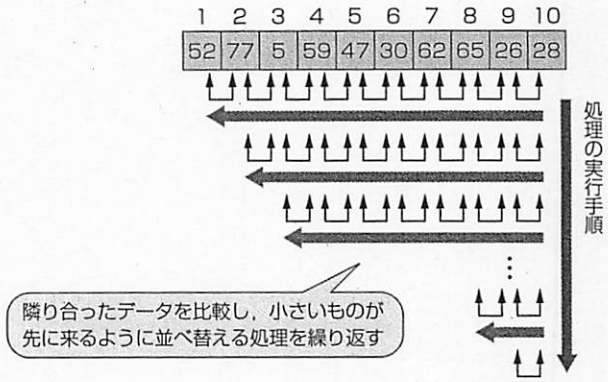


図7●データの挿入と入れ替えの違い

図6●バブルソートのアルゴリズム。隣り合ったデータを比較し，小さいものが先に来るように並べる。これを図のように繰り返すとすべてが並び替わる



バブルソートはわかりやすく，動作も名前も面白いのですが，残念ながらこれもデータの個数の2乗に比例して計算量が増えていく，非実用的なアルゴリズムです。もっとも，データの初期状態によっては，繰り返しの途中でソートが完了してしまう場合があります。これを検出して，繰り返しを途中で打ち切るようにすれば少し効率が良くなります。具体的には，配列の後ろから先頭に向かうループの中でデータの入れ替えがあった回数を数えて，1度もなければソート完了とします。この仕組みを組み込めば，大部分がソート済みのデータに対するソートは，挿入ソートと同じようにかなり高速な動作が期待できます。

ただ，バブルソートはデータの入れ替えを何度も実行するため，挿入ソートに比べると効率が悪くなります。例えば，**図7**(1)のように並んだデータをソートすることを考えましょう。挿入ソートでは，挿入したいデータ（ここでは“3”）を一時的に待避して，ソート済みの部分から一つずつデータをコピーして後ろにずらして最後に一時的なデータを適切な位置にコピーします(2)。

一方，バブルソートではデータのコピーではなく入れ替えで処理を進めます(3)。この図だけを見ると，データの入れ替えが2回で，挿入ソートよりも効率が良さそうです。しかし実際には，一般的なプログラミング言語で二つのデータを入れ替えるには，コピー操作が3回必要になります。つまり，図7(3)の処理ではコピー処理が最低でも6回行われることになります。

なお，バブルソートの変形版として「シェーカーソート」と「コームソート（櫛ソート）」というアルゴリズムがよく知られています。

シェーカーソートは，バブルソートが大きいデータの動きを成り行き任せにしている点に着目して変更を加えたアルゴリズムです。小さいデータを先頭に押し出す処理と，大きいデータを末尾に押し出す処理を交互に繰り返してソートします。カクテルを作るシェーカーになぞらえてこう名付けられています。一見すると良さそうなアルゴリズムですが，実際には比較・交換の回数は減らず，改良版とは呼べません。

一方，コームソートはバブルソートを劇的に速くするといわれる改良版です。バブルソートが1回の交換でデータを一つしか前に進められない点を改良しています。具体的には，隣り合ったデータを比較するのではなく，一定の間隔を空けたデータ同士での比較と交換を，徐々に間隔を縮めながら繰り返すことで高速化しています。

*5 1番目を取り除かずに，常に配列の先頭まで比較，交換をしてもソートできますが，繰り返しの回数が2倍に増えてしまいます。