基礎プログラミング+演習 ＃7 – 整列アルゴリズム+計算量

久野 靖 (電気通信大学)

2017.12.11

今回は「整列」のさまざまな手法を取り上げ、それに関連して、アルゴリズム解析において重要な「計算量」の概念について学びます。「整列」が題材なのは、1つの作業に対してさまざまなアルゴリズムがあるという点でぴったりだからです。というわけで今回の内容は次の通り。

- 基本的な整列アルゴリズムと
- より高速な整列アルゴリズム
- 時間計算量の考え方

1　さまざまな整列アルゴリズム

1.1　整列アルゴリズムを考える

配列を扱うアルゴリズムの代表例として、数値の並びを受け取り、**昇順** (ascending order — 小さいもの順) に並べ換えることを考えます。これを**整列** (sorting) と言います。[1]

アルゴリズムの前に、皆様が現実世界で整列を行うとき (たとえば数字を書いたカードを順に並べる際)、どうにするか考えてみてください。ただし、コンピュータに移すことを前提に、次の制限を設けます。

- カードは列にきっちり (間をあけずに左から詰めて) 並べること (配列に対応)。
- 2本の人差指だけを使ってカードを指して動かす (コンピュータでは本当は操作できるデータは一時には1個だけれど、さすがに不自由すぎるので2本にします)。
- カードの数値を読んだり比較するときは、2本の指のどちらかでそのカードを指す (これもコンピュータが操作できるデータは一時には1個だけだから)。

この条件で、実際にカードの並べ替えをやってみて頂きます。

演習 1 数字のカード (10枚くらい) をよく切って机上に1列に並べ、上の条件を守って昇順に並べ替えなさい。ペアのところはペアで互いに観察し、相手のアルゴリズムを推察しなさい。

1.2　基本的な整列アルゴリズム exam

整列のアルゴリズムは見つかりましたか。では、一番基本的な整列アルゴリズムを1つお見せしましょう。次の擬似コードを見てください。

- bubsort(a): 配列 a を昇順に整列
- 　done ← 偽。
- 　done でない間繰り返し、
- 　　done ← 真。

[1] 逆に大きいものが先に来るような順番の場合は**降順** (descending order) で、正確には列を昇順や降順に並べ換える処理が整列です。

- 　　i を 0 から a.length-2 まで変えながら繰り返し、
- 　　　もし a[i] と a[i+1] の順番が逆なら、
- 　　　　{a[i] と a[i+1] の値を交換。}
- 　　　　done ← 偽。
- 　　枝分かれ終わり。
- 　繰り返し終わり。
- 繰り返し終わり。

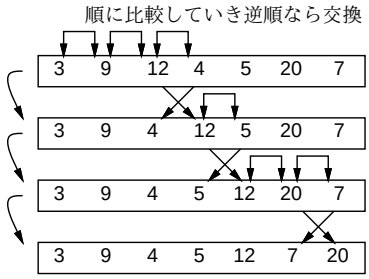

図 1: バブルソートによる整列

擬似コードでは手続きを呼ぶところは「{…}」で囲みました。このコードの肝となるところは、内側のループで隣り合う要素を順に見ていき、逆順になっているところがあれば交換する、というところです。これを次々におこなっていくと、大きい要素が右のほうに移動していきます (図 1)。この処理を繰り返していくと、最後は全要素が昇順で並び、交換が起きなくなります。

繰り返しを終わってよいかどうか判断するために、`done`(終了) という旗を用意し、まず立ててから上記の比較交換を行います。交換を行ったら、そのことを示すために旗を降ろします。最後まで旗が立ったままだったら、1 回も交換しなかった、つまり 1 箇所も逆順になっているところがなかったということなので、整列が完了しています。

この整列方法を**バブルソート** (bubblesort) と呼びます。各要素が移動する様子が水中から泡が浮かんでくるのに似ているためにこう呼ばれるとされています。

ところで、「`a[i]` と `a[i+1]` を**交換** (swap)」という命令は言語には直接ないので、これを手続きとして記述します。

```
def swap(a, i, j)
  x = a[i]; a[i] = a[j]; a[j] = x
end
```

これで配列 `a` の `i` 番目と `j` 番目の要素を入れ換えることができます (実際にそうなっていることをコードを追って確認しておいてください)。この程度のコードであれば、いちいち手続きとして抽象化しないで直接書いてしまいたい、と思うかもしれません。筆者の考えとしては、それはコードに対する慣れにもよってどちらもありだと思いますが、「交換」するところに「`swap`」と明示的書いてある分かりやすさも捨てがたいと思っています。[2]

バブルソート本体は次のようになります。

```
def bubblesort(a)
  done = false
  while !done do
```

[2]Ruby では**多重代入** (multiple assignment — 複数の代入を一度に行うこと) が使えるため、`swap` の本体を `a[i],a[j] = a[j],a[i]` と書くこともできます。多重代入機能を持たない言語も多いので、ここでは「普通の」やり方を示しておきました。

```
      done = true
      0.step(a.length-2) do |i|
        if a[i] > a[i+1] then swap(a, i, i+1); done = false end
      end
    end
  end
```

では実際に動かしてみましょう。

```
irb> a = [1, 9, 5, 4, 2]
=> [1, 9, 5, 4, 2]
irb> bubblesort(a)
=> nil
irb> a
=> [1, 2, 4, 5, 9]
irb>
```

bubblesort 自体は値を返さず、配列 a の中身を書き換えて昇順に整列していることに注意。したがって、まず配列 a を用意し、それを bubblesort に渡して整列させ、最後に a を打ち出して並んでいることを確認しています。

バブルソートのように、「求める状態が成り立っていない間、少しでもその状態に近付けることをずっと繰り返す」というのはコンピュータではよくありますが、人間はあんまりそのなやり方はしない気もします。もうちょっと自然な考え方のものとして、次のものはどうでしょうか。

> 数の並びから最小値を取り出してはその並びからは取り除くことを繰り返していく。その取り出したものを順に並べると昇順の整列結果になっている。

この方法を、単純に小さいものをそのつど選ぶことから**選択ソート** (selection sort) と呼びます。

これを作るに当たっては、「配列 a の i 番目から j 番目までの間で最も小さい要素が何番目にあるかを返す」操作を下請けメソッドとして用意するのがいいでしょう。それがあったとして、アルゴリズムは次のようになります。

- selectionsort(a): 配列 a を単純選択法で整列
- 　i を 0 から a.length-2 まで変化させながら繰り返し、
- 　　k ← {a の i 番以後の最小要素の番号。}
- 　　{a[i] と a[k] の内容を交換。}
- 　繰り返し終わり。

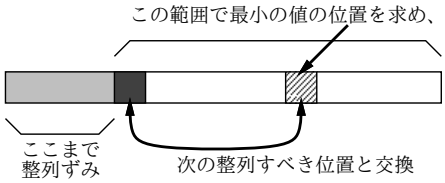

図 2: 単純選択法による整列

なぜ「交換」を使っているのかというと、まず選んだ最小の要素を先頭に置くには、先頭にある要素と最小の要素とを交換するのが合理的だからです。その後も、残っているものの中から最も小さい要素を選んではその先頭位置と交換することで、1 つの配列だけですべての作業が行えます (図 2)。

ではコードを示します。arrayminrange は以前やった最大/最小と同様に考えればよいでしょう (最小値そのものでなくその位置を返すことに注意)。

```
def selectionsort(a)
  0.step(a.length-2) do |i|
    k = arrayminrange(a, i, a.length-1); swap(a, i, k)
  end
end
def arrayminrange(a, i, j)
  p = i; min = a[p]
  i.step(j) do |k|
    if min > a[k] then p = k; min = a[k] end
  end
  return p
end
```

単純選択法は、数を 1 つずつ処理しますが、それらを「取り出す」時に正しい順になるようにするというものでした。人間にとって自然なもう 1 つのやり方は、取り出すのは「最初に並んでいる順」で、入れる時に正しい位置に入れる、というものです。

> 数の並びから順に数を取り出し、それを新しい列に加えるが、ただし新しい列に入れる時に「順番として正しい」位置に挿入するようにする (その後ろにある要素はずらす必要があることに注意)。

この方法は、単純に各要素を次々とあるべき位置に挿入していくことから、**挿入ソート** (insertion sort) と呼ばれます。

これを作るに当たっては、「配列の a の i 番目が空いているものとして、x より大きい要素を 1 つずつ詰めていって、最終的な空きの位置を返す操作」を下請けメソッドとして作るのがいいでしょう。それがあったとして、単純挿入法のアルゴリズムは次のようになります。

- insertionsort(a): 配列 a を挿入ソートで整列
- 　i を 1 から a.length-1 まで変化させながら繰り返し、
- 　　x ← a[i]。
- 　　k ← {a の i 番目以前で x より大な値をずらしていき、最終的に x が入る位置を返す。}
- 　　a[k] ← x。
- 　繰り返し終わり。

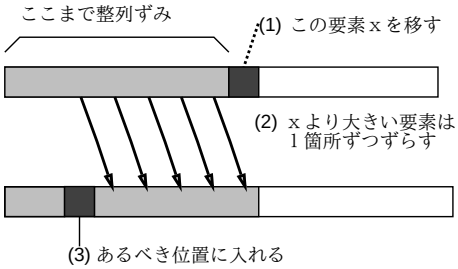

図 3: 単純挿入法による整列

これも、元の配列からデータを取り除きながらそれをもとに先頭部分に整列されている部分を作っていくので、配列は 1 つだけで済みます (図 3)。なお、配列を「後ろにずらす」時に後ろから順にやらないとまずいことに注意してください (図 4)。

ずらすコードと本体のコードは次のとおりです。

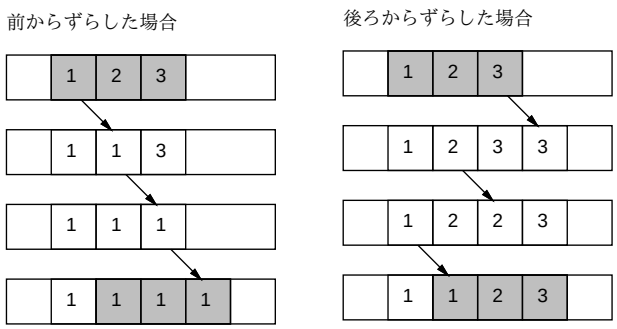

図 4: 配列をずらす

```
def insertionsort(a)
  1.step(a.length-1) do |i|
    x = a[i]; k = shiftlarger(a, i, x); a[k] = x
  end
end
def shiftlarger(a, i, x)
  while i > 0 && a[i-1] > x do a[i] = a[i-1]; i = i - 1 end
  return i
end
```

1.3 整列アルゴリズムの計測 exam

次の段階として、「整列プログラムの時間を計測する」問題に取り組んでみましょう。「バブルソート」「単純選択法」「単純挿入法」の 3 つの整列アルゴリズムについて、データ量を変化させて**時間計測**を行ってみます。データは次のコードにより個数を指定して**乱数**によりランダムに生成します。

```
def randarray(n)
  return Array.new(n) do rand(10000) end
end
```

rand は乱数を生成するメソッドで、パラメタを指定しないと区間 $[0, 1)$ の**一様乱数** (uniform random number) を (実数値で) 返し、パラメタとして整数 N を指定すると、0 以上 N 未満の整数値の一様乱数を返します。ちょっと試してみましょうか。

```
irb> randarray 10
=> [9257, 4988, 6894, 8064, 329, 4362, 1868, 472, 1527, 6317]
```

次に、時間計測に役立つメソッドを用意します。これは、実行回数とブロックをパラメタとして受け取り、「まず現在の時計を調べ」「指定回数ぶんだけブロックの中身を実行し」「再び時計を調べ」「2 つの時刻の差を表示」します。ただしここでの時計は「CPU をどれだけ使ったか」を示す時計になっています。

```
def bench(count, &block)
  t1 = Process.times.utime
  count.times do yield end
  t2 = Process.times.utime
  puts t2-t1
end
```

これもちょっと使ってみましょう。

```
irb> bench(100000) do 2 + 1 end
0.078125
=> nil
irb> bench(100000) do 20000000000000000 + 1 end
0.171875
=> nil
```

整数の場合、ある程度より大きくなると計算時間が余分に掛かるようになることが分かります。

さて、これらの材料を使って、整列の速度を測ります。randarray で多めの配列を生成し、bench では回数として 1 回を指定して整列を行い、時間を計測します。これを 1 行にまとめて書くとして、次のようになります。

```
irb> a = randarray(1000); bench(1) do bubblesort(a) end
=> 1.21875    ←計測結果が表示される
```

演習 2 バブルソート、単純選択法、単純挿入法のうちから 1 つ好きな整列アルゴリズムを選んで打ち込み、複数のデータ量で時間計測を行い、データ量と所要時間の関係がどうなっているか分析せよ (グラフに描いて観察するなど — グラフはレポートにはつけなくて良いです)。

なお、下請けのメソッドや計測メソッドも忘れずに打ち込む必要があります。具体的には次のようになります。

- バブルソート — bubblesort、swap、randarray、bench
- 単純選択法 — selectionsort、arrayminrange、randarray、bench
- 単純挿入法 — insertionsort、arrayshiftrange、randarray、bench

1.4 基本的な整列アルゴリズムの計測

とりあえず、筆者の手元のマシンでのバブルソート、単純選択法、単純挿入法の計測結果を、表 1 に示します。これを見ると、バブルソートが圧倒的に遅く、残りの 2 つはそれほど大きな差はない、

表 1: バブルソート/単純選択法/単純挿入法の所要時間 (msec)

データ数	1,000	2,000	3,000
バブルソート	1,219	4,945	11,117
単純選択法	305	1,242	2,766
単純挿入法	375	1,531	3,445

ということが分かります。これは、バブルソートはすべての要素を移すのに隣と 1 個ぶんずつ交換してゆくのでどうしても手間が多くなるのに対し、他の 2 つでは「1 個データを選んで、それを適切な位置に置く」ことを繰り返す形なので、それだけ手間が少なくなるからだと言えるでしょう。

では次に、データの量が 2 倍、3 倍になった時の所要時間を見てみると、こんどはどのアルゴリズムでも所要時間がほぼ 4 倍、9 倍になっていることが分かります。$4 = 2^2$、$9 = 3^2$ ですから、どのアルゴリズムでも「所要時間はデータ量の 2 乗に比例している」と言ってよいでしょう。ということは、データ数が 100,000(100 倍) になった時の所要時間は単純選択法でも $0.3 \times 10,000 = 3000$ 秒 $= 50$ 分 (!) となってしまい、終わるまで待つのはあまり嬉しいものではないと分かります。

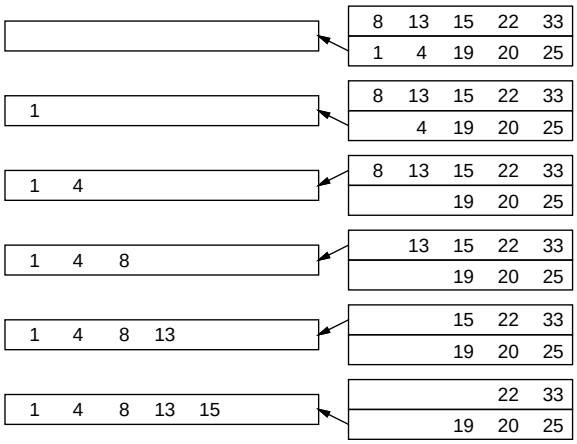

図 5: マージの処理

1.5 マージソート exam

では、整列アルゴリズムでもっと速いものはないのでしょうか。ここで**マージソート** (merge sort) と呼ばれるアルゴリズムを見てみましょう。**マージ** (merge) とは**併合**とも呼ばれ、図 5 のように 2 つの整列ずみの列を「あわせて」1 つの整列ずみの列にすることを言います。マージソートの手続きの呼び出し時には次のように、「配列のどこからどこまでを整列する」かを指定するものとします。

```
mergesort(a, 0, a.length-1);
```

擬似コードを示します。

- mergesort(a, i, j) — 配列 a の i 番から j 番の範囲を整列
- 　もし j ≤ i なら、
- 　　なにもしない。
- 　そうでなければ、
- 　　k ← (i + j) / 2。
- 　　mergesort(a, i, k)。mergesort(a, k+1, j)。
- 　　b ← merge(a, i, k, a, k+1, j)。
- 　　{b の内容を a の位置 i〜j にコピーし戻す }
- 　枝分かれ終わり。

考え方としては、まず再帰呼び出しによって列全体を半分ずつにしてゆき、長さ 1 以下の時は「もう整列ずみ」なので何もしないで帰ります。そして再帰から戻ってきたら、2 つの整列ずみの列をマージすることで長い整列ずみの列にします (図 6)。

下請けとなるマージの擬似コードは次の通り。

- merge(a1, i1, j1, a2, i2, j2) — a1[i1..j1] と a2[i2..j2] を併合
- 　b ← 空の配列
- 　i1..j1 と i2..j2 の少なくとも一方が空でない間、
- 　　もし i1..j1 が空 または
- 　　　　i2..j2 が空でなく a1[i1] > a2[i2] なら、
- 　　　a2[i2] を b に追加し、i2 を 1 ふやす。
- 　　そうでなければ、
- 　　　a1[i1] を b に追加し、i1 を 1 ふやす。

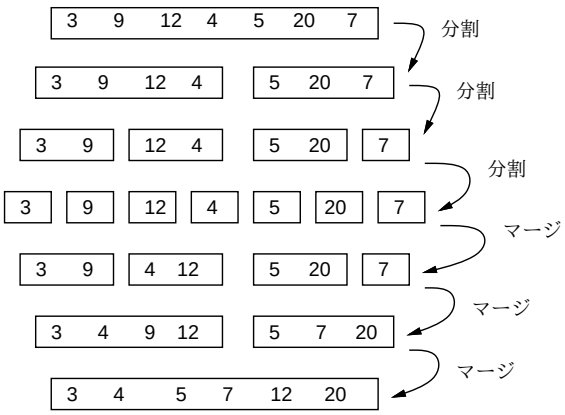

図 6: マージソートによる整列

- 枝分かれ終わり。
- 繰り返し終わり。
- b を返す。

では Ruby 版を見てみましょう。

```
def mergesort(a, i, j)
  if j <= i
    # do nothing
  else
    k = (i + j) / 2
    mergesort(a, i, k); mergesort(a, k+1, j)
    b = merge(a, i, k, a, k+1, j)
    b.length.times do |l| a[i+l] = b[l] end
  end
end
def merge(a1, i1, j1, a2, i2, j2)
  b = []
  while i1 <= j1 || i2 <= j2 do
    if i1 > j1 || i2 <= j2 && a1[i1] > a2[i2]
      b.push(a2[i2]); i2 = i2 + 1
    else
      b.push(a1[i1]); i1 = i1 + 1
    end
  end
  return b
end
```

マージソートは次に出て来るクイックソートに比べて速くはないのですが、データを端から順に処理していけるという特徴があります。このため、メモリに入り切らない (ファイルに保管されている) 大量データの処理に多く使われます。その原理は次のようなものです。

- データをファイルから読みながら、メモリに入る最大量ずつクイックソートなどで整列し、別々のファイルに書き出す。

- ファイル 1 とファイル 2 をマージしてファイル 12 を作り、ファイル 3 とファイル 4 をマージしてファイル 34 を作り、…のようにファイルを対にしてマージしていく。
- これを繰り返していって、最後に 1 本のファイルになったら完了。

このような、ディスクなど外部記憶の使用を前提とした整列のことを外部整列 (external sorting) と呼びます。これと対比して、これまで扱ってきたような、メモリ上での整列のことを内部整列 (internal sorting) と呼びます。

1.6 クイックソート exam

もう 1 つ別のアルゴリズムを直接 Ruby プログラムで示しましょう。これは**クイックソート** (quicksort) という、いかにも速そうな名前がついています。

```
def quicksort(a, i, j)
  if j <= i
    # do nothing
  else
    pivot = a[j]; s = i
    i.step(j-1) do |k|
      if a[k] <= pivot then swap(a, s, k); s = s + 1 end
    end
    swap(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j)
   end
end
```

非常に短いですが、説明されないと分かりませんね。まず長さ 1 以下なら何もしないのはマージソートと同様です。次に、マージソートと同じく列を 2 つに分けますが、こちらは**ピボット** (pivot) と呼ぶある値 p を選び、「左半分は p 以下、続いて p の値、右半分は p より大きい」という状態にしてから、左半分と右半分をそれぞれ自分自身を再帰呼び出しして整列します。そうすると、「p 以下の整列された列」「p」「p より大きい整列された列」になるのでこれで整列が完了するわけです (図 7)。

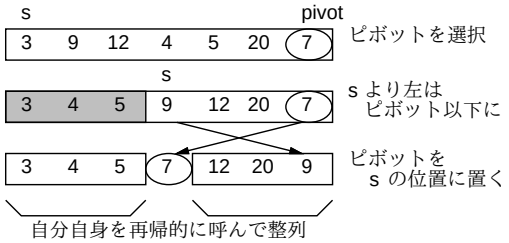

図 7: クイックソートによる整列

p としては「ちょうど列を半分ずつに分ける値」を使えるとベストですが、そんなものは分からないのでランダムに選ぶこととし、上のコードでは右端 (j 番目) の値を p にしています。変数 s は「この番号の 1 つ手前までは p 以下のものを詰めてあるので、次に p 以下のものが見つかったらこの位置に入れる」番号を表しています。そこで、k を i から j-1 まで左から順に調べて、[3]a[k] が p 以下ならそれを s 番目の要素と交換して s を増やすことで、左半分と右半分に分けられます。分け終わったら、最後に j 番目と s 番目を交換することで、保留してあったピボットの値をあるべき位置に置きます。その後、自分自身を再帰的に呼ぶわけですが、s 番目のピボットの位置はこれで合っているので、i〜s-1 と s+1〜j の範囲について自分自身を呼びます。

[3] j 番はピボットが入っているので保留します。

演習 3 マージソートとクイックソートの好きなほう (両方でもよい) を打ち込んで動かし、所要時間を計測してみよ。配列サイズを変化させた時の挙動は先にやったバブルソートや単純選択法や単純挿入法と比べてどうか考察せよ。

2 時間計算量

2.1 時間計算量の考え方 exam

ここまででさまざまなアルゴリズムを実現するプログラムの所要時間の計測について話題にしてきましたが、本節ではアルゴリズムの**性能** (performance) を評価する指針の 1 つである**計算の複雑さ** (computational complexity) ないし**計算量** (complexity) について取り上げます。complexity だと日本語は「複雑さ」になりそうですが、「複雑さ」という日本語では一般的すぎて何のことか分かりにくいので、日本語では「計算量」と呼ぶわけです。なお、計算量には「どれくらいメモリが必要になるか」を表す**領域計算量** (space complexity) もありますが、ここではとりあえず「所要時間」に着目する**時間計算量** (time complexity) を取り上げます。

`selectionsort` を例題にして、これがどれくらいの時間を要するかかを見積もってみましょう。その前に、まず次の前提を置きますが、これはいいですね?

> コンピュータは、ある 1 つの決まった動作はその動作に応じた決まった時間で実行している。

このことは、バブルソートなどの実行時間を測ってもそう大きくは変動しないことからも分かると思います。では次に、選択ソートのプログラムを「実行回数によって区分した」ものを図 8 に示します。

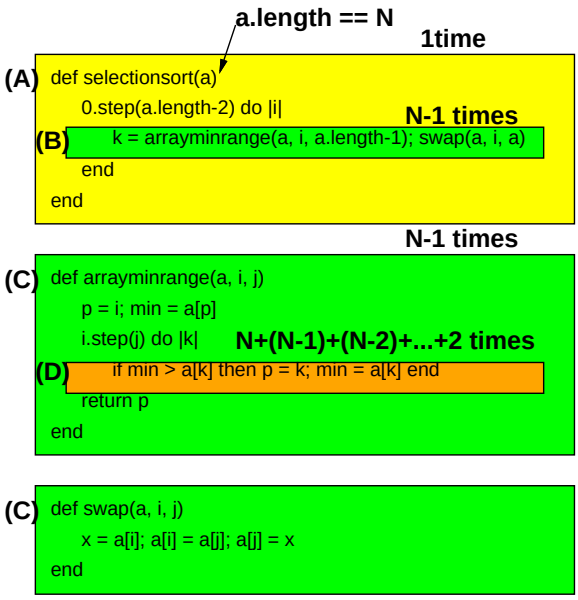

図 8: 選択ソートのコード実行回数の検討

ここで、渡された (整列する) 配列の長さを N とします。そうすると、実行回数について、次のことが分かります。

(A) メソッド `selectionsort` の本体部分は「1 回」実行される。

(B) その中の do の内側については、「$N-1$ 回」実行される。

(C) したがって、`arrayminrange` や `swap` も「$N-1$ 回」実行される。

(D) `arrayminrange` の中の do の内側は、最初は N 回、次は $N-1$ 回、次は $N-2$ 回…、ということで、合計 $\frac{N(N+1)}{2}-1$ 回実行される。

ここで、(A) の部分 (から (B)、(C) の部分は除いたもの、以下同様) の実行に掛かる時間を C_0、(B) と (C) の部分を 1 回実行するのに掛かる時間を C_1、(D) の部分を 1 回実行するのに掛かる時間を C_2 と置くと、合計実行時間は次のようになりますね。

$$T = C_0 + (N-1)C_1 + (\frac{N(N+1)}{2} - 1)C_2$$

これを展開して N について整理すると次のようになります。

$$T = \frac{C_2}{2}N^2 + (\frac{C_2}{2} + C_1)N + (C_0 - C_1 - \frac{C_2}{2})$$

ここで、たとえ C_0 や C_1 が C_2 の 100 倍あるとしても (確かに C_1 の部分の方が沢山あるけれど、どう見ても 100 倍は無いですよね?)、N が 1000 とか 10000 とかもっと大きな値で動かすわけなので、結局、次のように近似してもほぼ間違いではなくなってしまうわけです。

$$T \sim \frac{C_2}{2}N^2$$

そういうわけで、時間計算量とは「最も高次の次数だけを問題にする」考え方で、選択ソートについては $O(N^2)$ のように記すわけです。じゃあ係数はどうなんだ? ということですが、係数は「同じ計算量のプログラムどうし」では問題になりますが、計算量が違うプログラムであれば多少の係数の大小があっても結局は次数で決まってしまうので、無視してよい、というわけです。

N 個のデータを入力するようなプログラムでは、そのデータの読み込みに $O(N)$ は最低必要です (なお、N 個の値を扱うとしても、それを内部的に計算するだけなら、計算を工夫して $O(N)$ より小さいアルゴリズムを構成できる場合もあり得ます)。この、$O(N)$ のアルゴリズムのことを (N に比例するわけですから) **線形時間** (linear time complexity) と呼びます。たとえば最大や最小を求める問題はデータを読みながら一巡すれば結果が求まるので、線形時間のアルゴリズムで扱えます。このような問題はコンピュータで簡単に処理できると言えます。

少し込み入ったアルゴリズムは、$O(N^2)$ や $O(N^3)$ などの計算量になります。これを**多項式時間** (polynominal time complexity) と呼びます。さらに時間計算量の大きなものとしては、$O(C^N)$ すなわち**指数時間** (exponential time complexity) となる場合もあり、これだとコンピュータで実用的に扱えるのは小さい N に限られてしまいます。

2.2 整列アルゴリズムの時間計算量

では次に、単純挿入法の時間計算量はどうでしょうか。外側のループで i を 1〜N まで変えながらその番号の要素を適切な位置に挿入していきます。挿入位置を探索するのに平均して $\frac{i}{2}$ 個の要素を比較し、挿入位置が見つかったら平均して $\frac{i}{2}$ の要素を後ろにずらす必要があります。なのでこれも $1+2+\cdots+(N-1)$ の定数倍、つまり $O(N^2)$ になります。

バブルソートの時間計算量はどうでしょうか。内側のループでは $N-1$ 回の比較をおこないます。そして、**最良の場合** (best case) つまり最初から全部並んでいる場合は、1 回内側のループを実行したらそれで完成です。つまり $O(N)$ となります。しかし**最悪の場合** (worst case)、つまり完全に逆順に並んでいる場合は、内側の 1 回目のループは最も大きい要素を最後の位置に持ってくるだけで終わってしまい、2 回目は 2 番目に大きい要素を最後から 2 番目の位置に…というわけで、内側のループが N 回必要になります。つまり $O(N^2)$ となるでしょう。では**平均の場合** (average case) はどうでしょうか。平均的には、内側のループの繰り返しは N 回は必要ないとしても、N に比例する回数が必要になりそうです。ということは、平均でも定数倍は無視するのでやはり $O(N^2)$ になるわけです。

ここで表 1 の結果を振り返ると、単純選択法もバブルソートも N が 2 倍、3 倍になった時所用時間が 4 倍、9 倍になっているので、この計測結果はこれらが確かに $O(N^2)$ の時間計算量であることを裏付けています。

ではマージソートの計算量はどうでしょうか。1 つの `mergesort` の呼び出しを見ると、単純な場合 (長さが 1 以下) は一定時間で済みます。長さ N の場合は、それを前半と後半に分けて、それぞれ自分自身を再帰的に呼び出して整列し、最後にマージします。自分自身に掛かる時間は分けて考えるとして、マージは両方の列の先頭を見て小さいほうを取ることを繰り返せばいいので、$O(N)$ で済みます。さて、再帰呼び出しのほうはどうでしょうか。長さ N の列を半分にしてそれぞれ `mergesort` を呼ぶのですから、2 段目の呼び出しは $O(\frac{N}{2}) + O(\frac{N}{2}) = O(N)$ 。3 段目は 4 分の 1 の列について 4 つ呼ぶのでやはり $O(N)$ 、となります。これが合計何段あるかというと、「N を何回半分にしたら 1 になるか」だから $\log_2 N$ となります。なので、全体では $O(N \log N)$ の計算量となります (計算量の議論では log の底が何かも省略するのが通例です)。

では、クイックソートの計算量はどうでしょうか。1 回ぶんの処理はやはり $O(N)$ で、再帰の段数はピボットの選択が完璧なら $\log_2 N$ 回ですが、ランダムに選んでいるのでその定数倍と考えてよいでしょう。すると定数倍は無視するので、これも計算量は $O(N \log N)$ になります。

ただし、極めて運が悪い場合、つまりピボットの選択が悪くて毎回列の最大か最小の値をピボットにしてしまうと、段数が N になってしまうので、最悪の計算量は $O(N^2)$ ということになります。そんな運が悪いことはないだろうと思うかもしれませんが、既に整列ずみの値を渡されるとまさにそうなってしまうのです。

演習 4 クイックソートに既に並んでいる配列を与えると計算量が $O(N \log N)$ から $O(N^2)$ になってしまうことを計測により確認しなさい。また、この弱点を解消する工夫を考えて実現してみなさい。

3 整数値のための整列アルゴリズム

3.1 ビンソート

ここまでの方法とは考え方がまったく違う整列アルゴリズムである、**ビンソート** (bin sort) を紹介しましょう。このアルゴリズムは、整列する値が整数であり、かつ範囲があまり広くない場合に利用できます。たとえば、整列する値の範囲が 0〜3 の整数だけだったとします (もちろん、そのデータの個数は 1 万も 2 万もあるかもしれません)。

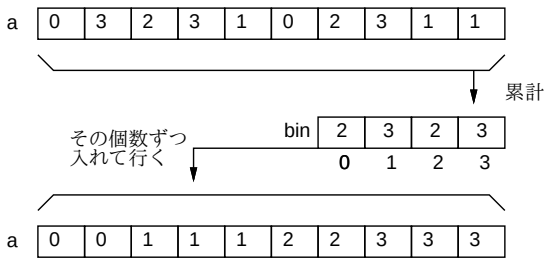

図 9: ビンソートによる整列

それなら図 9 のように、まず 0、1、2、3 それぞれの値について「何回現れるか」を数えてしまいます。そして数え終わったらこんどは「0 が 2 回、1 が 3 回、…」のように数えた個数ずつその値を繰り返せば、確かに元のデータを並べ換えたのと同じことになるわけです。0〜3 ではあんまり役に立たないと思うでしょうが、実際にはコンピュータのメモリは沢山あるので、0〜9999 とかでも全く問題ありませんし、それなら使い道は結構ありそうですね? [4]

演習 5 ビンソートのプログラムを作成し、所要時間を計測しなさい。もちろん、ビンソートの時間計算量についても検討すること。

[4]先に掲げた `randarray` が生成するデータもこの範囲の整数であることに注意してください。もちろんわざとそうしたのですが。

3.2 基数ソート

ビンソートの弱点は、現れる値の範囲があまりに広いと (100 万とか 1000 万とか) 巨大な配列を必要とし、効率も悪くなることです。そこで、やはり値が整数である必要があるものの、ビンソートよりも値の範囲に対する許容度が高い整列アルゴリズムである**基数ソート** (radix sort) を紹介しましょう。ここでは簡単のため、負の値はないものとして説明します。

基数ソートでは、整列する値を 2 進表現した時に「下から i ビット目が 1 であるか否か」を調べる必要があります。これを Ruby でどう書くかを説明しておきます。

Ruby では<<という演算子は**左シフト** (left shift) つまりビット列である整数値を 1 ビットぶん左にずらす働きがあります。だから `1 << 2` は $100_{(2)}$ だから 4 だし、一般に `1 << i` で `i` 番目のビットだけが 1 になった数値をえることができます。[5]

次に、&という演算子は**ビット毎 and**(bitwise and) 演算つまり 2 つの数の 2 進表現で「両方とも 1」の位置だけが 1、それ以外は 0 であるような 2 進表現に対応する数が得られます。[6] たとえば図 10 のように、`52 & 29` の結果は 20 ということになります。

```
  110100 — 52
&) 011101 — 29
  ------
  010100 — 20
```

図 10: ビット毎 and 演算

ここでようやく、基数ソートの説明に入ります。たとえば、変数 `mask` に 1 ビットだけが「1」になっている値を入れ、その 1 の位置を一番右 (下位) から順に左に移していきます。そして、その `mask` との&の結果が 1 か 0 かで、データを右半分と左半分に分割します (図 11)。そうするとあらふしぎ、一番上のビット (ここでは 4 ビットとしました) までやったときには、すべての数は小さい順に並んでいます。

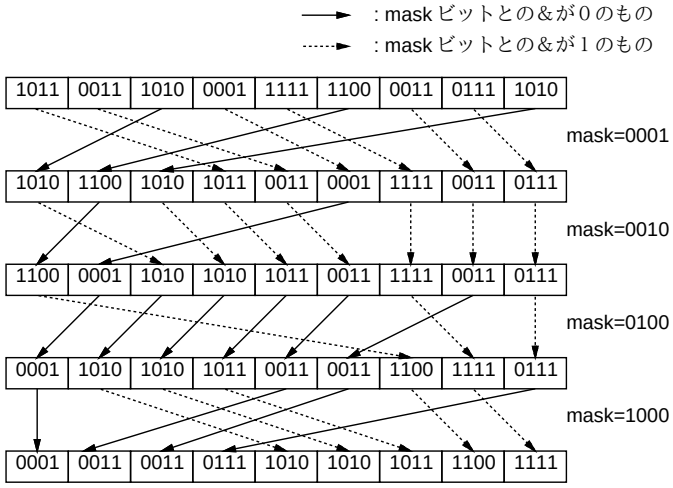

図 11: 基数ソートによる整列

これはなぜかというと、1 回目では一番下のビットが 1 のものが左、0 のものが右になるように振り分け、それについて 2 回目に下から 2 ビット目が 0 のものが左、1 のものが右になるように振り分けるわけですが、2 回目の振り分けをしても 1 回目の振り分けの順序は崩れないので、2 ビット目が 0 のものの中や、1 のものの中ではそれぞれ、まず 1 ビット目が 0 のもの、続いて 1 のものという順

[5] そして今回は使いませんが、もちろん>>は**右シフト** (right shift) 演算子です。

[6] 条件の「かつ」は&&でしたが、アンド記号が 1 個の場合はまったく別の意味になるわけです。ちなみに、|は**ビット毎 or**(bitwise or) 演算、~は**ビット毎反転** (bitwise inversion) 演算です。

序が維持されています。3 ビット目、4 ビット目でも同様にそれより下のビットについては順序が維持されているので、結局最後まで来たときには順番が完全に並んだ状態となるわけです。

演習 6 基数ソートのプログラムを作成し、所要時間を計測しなさい。もちろん、基数ソートの時間計算量についても検討すること。

演習 7 ここまでに出て来なかった整列アルゴリズム (あなたが考案したものでもよい) を 1 つ選び、所要時間を計測しなさい。もちろん、時間計算量についても検討すること。

本日の課題 7A

「演習 2」または「演習 3」の計測結果を、計測したプログラムと併せて、レポートとして提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 整列アルゴリズムを少なくとも 1 つは理解しましたか。

Q2. 時間計算量という考え方についてどう思いましたか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 7B

「演習 2」〜「演習 7」の (小) 課題から選択して 1 つ以上プログラムを (必要なら作って) 動かし、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 整列アルゴリズムをいくつくらい理解しましたか。また、それぞれの時間計算量についてはどうですか。

Q2. 自分で作れる程度のプログラムについてなら、その時間計算量が求められそうですか?

Q3. 課題に対する感想と今後の要望をお書きください。