第II部

プログラミングを通して学ぶ
情報科学の諸概念

第5章　アルゴリズムと計算量

コンピュータを使って問題解決をする場合には、解き方をいくつか考えて適切なものを選び、その解き方の通りに計算をするプログラムを作って実行する、という2つの段階を踏む。

アルゴリズムとはこの「解き方」のことである。より厳密には、問題を解くための手順で、有限の時間で答を出すことが保証されているものを指す。

問題が与えられたときに、いきなりプログラムを書かずに、アルゴリズムという中間段階を経由することには次のような意味がある。1つはプログラムを作る前から計算時間などがある程度—例えば「このやり方では一生かかっても終わらない」といった程度に—見積ることができる点である。もう1つはアルゴリズムに関する知識を問題から独立して蓄積できる点である。例えば、通信ネットワーク上に流すことができるデータ量を計算する問題と、工場においてどの機械にどの製品の製造させると効率的かという2つの問題は、問題としては全く違うもののように見えるが、実は共通のアルゴリズムを使って解くことができる。アルゴリズムを考えることで、こうした異なる問題についての知識やプログラムを使い回すことができるのである。

この章では、具体的な問題として Fibonacci 数列の計算を例にとる。まずこの問題を解くアルゴリズムが複数あることを見て、アルゴリズムによる計算時間の違いから、計算量の考え方を紹介する。さらに別の問題として整列のアルゴリズムを2つ紹介する。

5.1　Fibonacci 数を求めるアルゴリズム

ねずみ講、ねずみ算のようにネズミの増え方に関する話はよく知られている。ここではウサギの増え方を考てみよう。子ウサギは生後1ヶ月経つと親ウサギになり、以降、毎月1羽の子ウサギを生むようになるとする。このような場合に、

0月、1 羽の子ウサギがいたとする。	子
1月、親ウサギになる。	親
2月、1 羽の子ウサギを生む。	親子
3月、親ウサギはもう 1 羽の子ウサギを生み、子ウサギは親ウサギになる。	親親子
4月、2 羽の親ウサギがそれぞれ子ウサギを生み、子ウサギは親ウサギになる。	親親親子子
5月、3 羽の親ウサギがそれぞれ子ウサギを生み、2 羽の子ウサギは親ウサギになる。	親親親親親子子子
6月、5 羽の親ウサギがそれぞれ子ウサギを生み、3 羽の子ウサギは親ウサギになる。	親親親親親親親子子子子子

のように増えてゆくウサギの親子をあわせた総数を並べてみると

$$1,1,2,3,5,8,13,\ldots$$

という数列になる。この数列は **Fibonacci 数列**として知られており、続く 2 つの値の和が次の値になっている性質を持っている。この数列の (先頭を 0 番目としたときの)n 番目の値を $\mathrm{fib}(n)$ と書くことにし、$\mathrm{fib}(n)$ を求める問題についてのアルゴリズムを考えよう。

5.1.1　再帰的アルゴリズム

Fibonacci 数列の k 番目の値と、それより前の値との関係を書くと次のようになる。

$$\mathrm{fib}(k) = \begin{cases} 1 & (k = 0 \text{ または } k = 1) \\ \mathrm{fib}(k-1) + \mathrm{fib}(k-2) & (k \geq 2) \end{cases} \tag{5.1}$$

この定義に従ってそのまま計算を行うのが最初のアルゴリズムである。関係が再帰的なので**再帰的アルゴリズム**と呼ぶことにしよう。このアルゴリズムに従った計算は、例えば $\mathrm{fib}(4)$ であれば次のような順序で進む。

$$\mathrm{fib}(4) \Rightarrow \mathrm{fib}(3) + \mathrm{fib}(2) \tag{5.2}$$

のように展開できるので、$\mathrm{fib}(3), \mathrm{fib}(2)$ をそれぞれ求める (☆) ことになる。$\mathrm{fib}(3)$ を展開するとと

$$\mathrm{fib}(3) \Rightarrow \mathrm{fib}(2) + \mathrm{fib}(1) \tag{5.3}$$

となるので、今度は $\mathrm{fib}(2), \mathrm{fib}(1)$ をそれぞれ求める (★) ことになる。$\mathrm{fib}(2)$ を展開してみると

$$\mathrm{fib}(2) \Rightarrow \mathrm{fib}(1) + \mathrm{fib}(0) \Rightarrow 1 + 1 \Rightarrow 2 \tag{5.4}$$

となる。また★で残されていた fib(1) は

$$\mathrm{fib}(1) \Rightarrow 1$$

である。そこで式 5.3 に戻って

$$\mathrm{fib}(3) \Rightarrow \mathrm{fib}(2) + \mathrm{fib}(1) \Rightarrow 2 + 1 \Rightarrow 3 \tag{5.5}$$

を得る。

次に☆で残っていた fib(2) を求めることになるが、ここは式 5.4 と全く同じ展開を行って

$$\mathrm{fib}(2) \Rightarrow \mathrm{fib}(1) + \mathrm{fib}(0) \Rightarrow 1 + 1 \Rightarrow 2 \tag{5.6}$$

となる。

ここでようやく式 5.2 に戻って、式 5.5 と式 5.6 の結果を使って

$$\mathrm{fib}(4) \Rightarrow \mathrm{fib}(3) + \mathrm{fib}(2) \Rightarrow 3 + 2 \Rightarrow 5$$

となる。

この再帰的アルゴリズムは、ほとんどそのまま Ruby の関数として定義することができる。アルゴリズムごとに関数名を変えて区別するために、`fibr(k)` という名前で定義すると、次のようになる。

```
1 def fibr(k)
2     if k==0 || k==1
3         1
4     else
5         fibr(k-1) + fibr(k-2)
6     end
7 end
```

ファイル 5.1: `fib.rb` (その 1・再帰的アルゴリズム)

練習 5.1 (再帰的アルゴリズムの使用) `fibr` を使って $\mathrm{fib}(k) > 100000$ となる最小の k を見つけよ。(Control C を押すと計算を中止できる。)

5.1.2　数え上げアルゴリズム

最初に説明したように、Fibonacci 数列は続く 2 つの値の和が次の値になっている。従って $\mathrm{fib}(0), \mathrm{fib}(1)$ の値から $\mathrm{fib}(2)$ を、$\mathrm{fib}(1), \mathrm{fib}(2)$ の値から $\mathrm{fib}(3)$ を求め…といった順に、直前の 2 つの値を覚えておけば下から順に数列を作ってゆくこともできる。

いま k 番目の Fibonacci 数を f, その 1 つ前の値を p1, もう 1 つ前を p2 とすると、この 3 つの値の関係は次の表のようにまとめられる。

k	0	1	2	3	4	5	6	7	8	9	10	11	12
fib(k): f	1	1	2	3	5	8	13	21	34	55	89	144	233
1 つ前: p1	–	1	1	2	3	5	8	13	21	34	55	89	144
2 つ前: p2	–	–	1	1	2	3	5	8	13	21	34	55	89

従って k 番目の Fibonacci 数を求める場合には、最初 f, p1 を 1, 1 としておき、

1. 次の p2, p1 の値をそれぞれ今の p1, f の値として、
2. 次の f の値を 次のp1 + p2 にする

という操作を $k-1$ 回繰り返せばよい。($k-1$ 回になるのは最初の f, p1 の値を $k=1$ のときのものにしたためである。)

このアルゴリズムを Ruby の関数として定義するには、f, p1, p2 という変数の値を適切に与えておき、上のような操作を繰り返すだけでよいので、次のようになる。

```
8  def fibl(k)
9      f=1
10     p1=1
11     for i in 2..k
12         p2 = p1       #fib(i-2)
13         p1 = f        #fib(i-1)
14         f  = p1 + p2  #fib(i)
15     end
16     f                 #fib(k)
17 end
```

ファイル 5.2: fib.rb (その 2・数え上げアルゴリズム)

注意すべき点としては、p2 = p1(12 行目) と p1 = f(13 行目) の順序である。p1= f を実行してしまうと p1 は「次の p1」の値になってしまう。一方、p2 = p1 を実行するときの p1 は「今の p1」の値が入っていなければいけない。従って 12 行目と 13 行目はこの順序で実行しなければいけない。

繰り返しが終わった後の f には fib(k) の値が入っているので、それを関数の答としているのが 16 行目である。

練習 5.2 (正しさの確認) いくつかの k について、fibr(k) と fibl(k) が同じ値を返すことを確認せよ。また、繰り返しを使ってある範囲の k の値すべてについて同じ値になっていることを確認するような関数を考えてみよ。

練習 5.3 (近似値との比較) 黄金比 $\frac{1+\sqrt{5}}{2} = \phi$ としたとき、fib(k) は

$$\mathrm{fib}(k) \simeq \frac{\phi^{k+1}}{\sqrt{5}} \tag{5.7}$$

と近似できることが知られている。式 5.7 に従って fib(k) の近似値を計算する関数 fiba(k) を定義し、いくつかの k について fibr(k) との誤差を調べよ。

練習 5.4 (代入の順序) ファイル 5.2 の、12 行目と 13 行目を入れ替えた場合、どのような値が計算されるかその理由とともに説明せよ。

5.2　計算時間の違いと計算量

5.2.1　スピード競争

さて、Fibonacci 数を計算するアルゴリズムを 2 つ紹介したので、どちらが速いかを比べてみよう。腕時計を使って時間を計ってもよいのだが、測定結果をグラフにする配布プログラム bench.rb を使うことにする。次の画面のように bench.rb を読み込み、そこに定義されている run を使って Fibonacci 数の計算をさせてみよ。

> 注意:　以下の画面を実行する前に、配布プログラム bench.rb をダウンロードしておく必要がある。

```
1  irb(main):004:0> load("./fib.rb")  # fibrなどの定義
2  => true
3  irb(main):005:0> load("./bench.rb")# runの定義
4  => true
5  irb(main):006:0> run("fibr", 10)   # fibr(10)を測る
6  fibr(10)...finished in 0.000000 seconds.
7  => 89
8  irb(main):007:0> run("fibl", 10)   # fibl(10)を測る
9  fibl(10)...finished in 0.000000 seconds.
10 => 89
```

5 行目の命令 run("fibr",10) は、式 fibr(10) を計算し、それにかかった時間を表示する。関数名を文字列として与える必要があることに注意して

ほしい。画面を見て分かるように、fibr, fibl ともに計算時間が非常に短く、速度の違いは分からない。

そこで $k = 10, 11, \ldots, 24$ と変化させて fibr(k), fibl(k) の計算時間がどう変わるかを見てみよう。次のように繰り返し命令 for を使えば自動的に実行させることができる。

```
12 irb(main):009:0> for k in 10..24
13 irb(main):010:1>   run("fibr", k)
14 irb(main):011:1>   run("fibl", k)
15 irb(main):012:1> end
16 fibr(10)...finished in 0.000000 seconds.
17 fibl(10)...finished in 0.000000 seconds.
18 fibr(11)...finished in 0.000000 seconds.
19 中略
20 fibl(23)...finished in 0.000000 seconds.
21 fibr(24)...finished in 0.420000 seconds.
22 fibl(24)...finished in 0.000000 seconds.
23 => 10..24
```

もし $k = 24$ のときでも fibr の計算時間がほとんど 0 であった場合は、12 行目の 24 をより大きな値に変えて実行してみよ。

$k = 24$ くらいになると計算時間の差が見えてきた。関数 run は時間を測ると同時に、測った時間を図 5.1 のようなグラフとして表示してくれる。グラフは k の値を横軸に、時間を縦軸にとったもので、fibr の計算時間は k が大きくなるに従って除々に大きくなってきていることが分かる。一方、fibl はほとんどゼロのままである。つまり、

k が大きくなるにつれ、fibr(k) の計算時間は fibl(k) よりも長くなる

ことが分かる。

5.2.2　実行時間から計算時間を見積る

次に、例えば fibr(100) を計算するのにどのくらい時間がかかるかを予想してみよう。以下では fibr(k) の計算時間を $t_r(k)$ と書くことにする。

図 5.2 は、k の範囲を広げて fibr(k) の計算時間を測ってグラフにしたものである。k が大きくなると時間が急激に増えてゆくので時間軸を対数目盛りにしてみると、計算時間はほぼ直線的に増えていることが分かる。時間が対数目盛りなので、

方法は第 5.5 節 (p. 84) の「グラフの表示方法の変更」を参照せよ。

fibr(k) の実行時間は k の指数関数に近似できる

つまり A, B を定数として $t_r(k) = A \cdot B^k$ と近似できると推測できる。

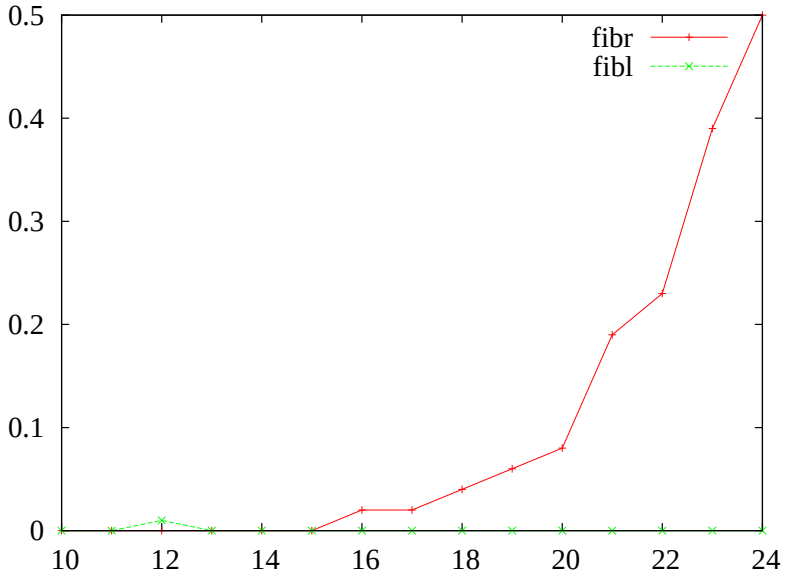


図 5.1: $10 \leq k \leq 24$ に対する `fibr(`k`)` と `fibl(`k`)` の実行時間

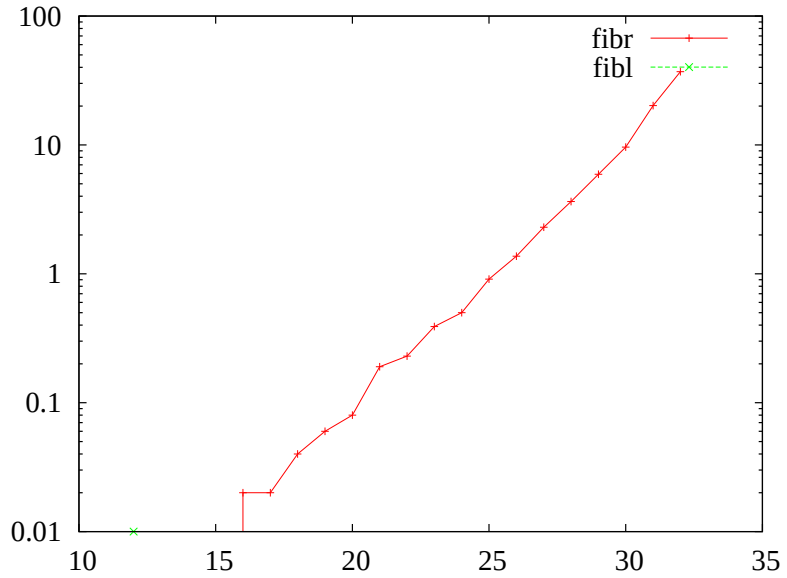


図 5.2: $10 \leq k \leq 32$ に対する `fibr(`k`)` の計算時間 (時間軸は対数目盛りになっている)

この式と実際の計算時間から A, B を求め、それを使って fibr(100) の計算時間を予想してみよう。いま、実行時間が $t_r(25) = 0.86$ 秒、$t_r(32) = 35.26$ 秒だったとする。これを上の式にあてはめると $A = 1.5 \times 10^{-6}$,　$B = 1.7$ となる。つまり、

$$t_r(k) \simeq 1.5 \times 10^{-6} \times 1.7^k$$

という近似式を得る。従って fibr(100) の予想計算時間は $1.5 \times 10^{-6} \times 1.7^{100} \simeq 1.7 \times 10^{17}$(秒) $\simeq$ 53 億 (年) となる。

次に fibl(k) の計算時間について考えてみよう。これまでの測定では fibl の計算時間はほとんど 0 だったので、より大きな k ついて時間を測定してみることにする。ここで fibl のかわりに **Fibonacci 数の下位 6 桁だけを求める**fibl6 を定義し、それの計算時間を測ることにする。

桁数の多い数の足し算は、その桁数に比例する時間がかかる。k が大きくなると Fibonacci 数は非常に大きな数になるため、それを求めるための足し算にかかる時間の変化も無視できなくなる。6 桁程度の数の足し算であれば、それにかかる時間は一定になる。

まず fibl6 の定義であるが、この関数は fibl の定義 (ファイル 5.2) の 14 行目の

```
f = p1 + p2
```

を

```
f = (p1 + p2) % 1000000
```

に変更するだけでよい。

fibl6 が定義できたら、計算時間を測ってみよう。以下は k を 10 万、20 万、...100 万と変化させたときの画面である。

```
25 irb(main):026:0> reset()
26 => []
27 irb(main):027:0> for m in 1..10
28 irb(main):028:1>   run("fibl6", 100000*m)
29 irb(main):029:1> end
30 fibl6(100000)...finished in 0.390000 seconds.
31 fibl6(200000)...finished in 0.770000 seconds.
32 中略
33 fibl6(900000)...finished in 3.440000 seconds.
34 fibl6(1000000)...finished in 3.820000 seconds.
35 => 1..10
```

25 行目の reset() は、いままで描いたグラフのデータを消去して、新しいグラフを作成させるための命令である。実行結果のグラフ (図 5.3) を見ると、計算時間 $t_l(k)$ はほぼ直線になっている。(縦軸は通常の目盛りに戻してあることに注意せよ。) これより、

fibl6(k) の実行時間は k の一次関数に近似できる

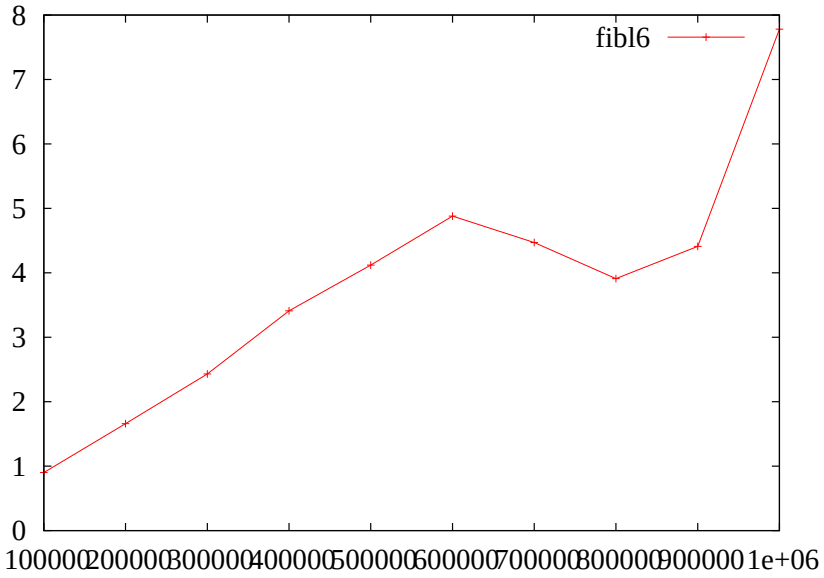


図 5.3: 10 万 $\leq k \leq$ 100 万 に対する `fibl6(`k`)` の計算時間

つまり C を定数として $t_l(k) = ck$ に近似できると推測できる。

今回の測定値では $C \simeq 7.8$ つまり $t_l(k) \simeq 7.8 \times 10^{-6} k$ となる。

練習 5.5 (計算時間の実測) 関数 `fibr(`k`)`, `fibl6(`k`)` の計算時間を実測し、$t_r(k) \simeq A \cdot B^k$, $t_l(k) \simeq Ck$ にあてはまる A, B, C を求めよ。

5.2.3　アルゴリズムから計算時間を見積る

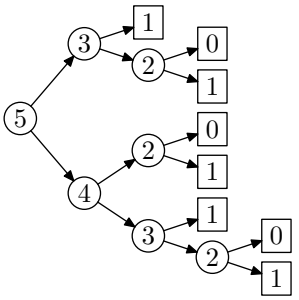
ここまでの話は、**プログラム**を実際に動してみた結果から計算時間を予想していたに過ぎない。つまり計算時間を予想したくても、実際にプログラムを作りそれを動かさなければいけない。これではプログラムを作る前に良いアルゴリズムを選ぶことはできない。

そこで今度は、**アルゴリズム**だけから計算時間を見積ってみよう。つまりアルゴリズムの性質から演算や式展開などの回数を理論的に求め、そこから計算時間を見積るという方法である。これであれば、プログラムを作る前からアルゴリズムによる計算時間の違いを予想できることになる。

再帰的アルゴリズムの計算時間

再帰的な Fibonacci 数の計算アルゴリズムは、(a) k が 1 より大きいかどうかを判定すること、(b) $\mathrm{fib}(k)$ という式を $\mathrm{fib}(k-1) + \mathrm{fib}(k-2)$ へ展開することと、(c) $\mathrm{fib}(k-1)$ と $\mathrm{fib}(k-2)$ の値を足すことから成り立っている。

いま fib(5) の計算を下図のように表わしてみる。⑤ は fib(5) の計算を表わし、○はさらに展開された計算、□はそれ以上展開されなかった計算を示す。



アルゴリズムとの対応を考えると、(a) は○と□の合計の個数回、(b) と (c) はそれぞれ○の個数回だけ行われている。また、どんな k であっても□の個数は $\mathrm{fib}(k)$ の値になり、○の個数は $\mathrm{fib}(k)-1$ の値になる。計算時間 $t_r(k)$ は、(a),(b),(c) の 1 回あたりの時間をそれぞれ T_a, T_b, T_c とすれば

$$t_r(k) = T_a(2\mathrm{fib}(k)-1) + (T_b+T_c)(\mathrm{fib}(k)-1) \tag{5.8}$$

となる。ところで、練習 5.3 で見た近似式を使うと式 5.8 はさらに

$$t_r(k) \simeq \frac{(2T_a+T_b+T_c)\phi}{\sqrt{5}}\phi^k - (T_a+T_b+T_c) \tag{5.9}$$

と近似できる。

T_a, T_b, T_c は、実行するコンピュータの性能や、プログラミング言語処理系によって異なるが、一定の値をとるとすればこの式は

再帰的アルゴリズムによる Fibonacci 数 $\mathrm{fib}(k)$ の計算時間は ϕ^k に比例した式で近似される

ことを示している。

第 5.2.2 節で実際の計算時間から求めた近似式は $t_r(k) \simeq 1.5\times10^{-6}\times1.7^k$ であったが、この式の k の底と式 5.9 の k の底 $\phi \simeq 1.618$ が近い値であることからもこの見積りの妥当性が分かる。

数え上げアルゴリズムの計算時間

数え上げアルゴリズムの場合の計算時間はもっと簡単である。このアルゴリズムでは、(d) 最初に「現在、1 つ前、2 つ前の値」を決めた後は、(e) 繰り返しが終わったかどうかの判定 (f) 「現在、1 つ前、2 つ前の値」の入れ換え (g) 現在の値の計算 (足し算) を $k-1$ 回繰り返しているだけである。従って計算時間 $t_l(k)$ は (d), (e), (f), (g) の 1 回あたりの時間をそれぞれ T_d, T_e, T_f, T_g としたときに

$$t_l(k) = T_d + (T_e+T_f+T_g)(k-1) \tag{5.10}$$

である。この式は

数え上げアルゴリズムによる Fibonacci 数 $\mathrm{fib}(k)$ の計算時間は k に比例した式で近似できる

ことを示している。実際、第 5.2.2 節で測定したプログラム `fibl6` の計算時間はほぼ k に比例していたので、この見積りに合致していると言える。

5.2.4　計算量

アルゴリズムの速さを比べる場合には、式 5.9 や式 5.10 の T_a, T_b などどをすべて 1 にした、**計算量**と呼ばれる値を用いる。Fibonacci 数を求める 2 つのアルゴリズムであれば、それぞれの計算量 $c_r(k), c_l(k)$ は次のようになる。

$$\begin{aligned} c_r(k) &\simeq \frac{3\phi}{\sqrt{5}}\phi^k - 3 \qquad \text{(再帰的)} \\ c_l(k) &= 5k + 2 \qquad \text{(数え上げ)} \end{aligned}$$

多くの場合は、さらにこれらの式の

- 定数係数や定数項を消し、
- 入力の大きさを表わす変数について、最も次数が高い項のみを残した

ような式を求めた上で比較をする。このような式は**計算量のオーダー**と呼ばれ、入力を大きくした際に、計算量がこの式に近似できるような変化をすることを意味する。Fibonacci 数の再帰的アルゴリズムであれば、ϕ^k の係数 $\frac{3\phi}{\sqrt{5}}$ や定数項 -3 を消し、数え上げアルゴリズムであれば係数 5 や定数項 2 を消して

$$\begin{aligned} &c_r(k) \text{ のオーダーは } \phi^k \\ &c_l(k) \text{ のオーダーは } k \end{aligned}$$

となる。

また、アルゴリズムによっては「ある場合は k^2 回、別の場合は k 回」のように場合によって計算量が違うこともある。そのような場合には最悪の場合として**最も大きくなる場合だけを考える**(この例では k^2) のが一般的である。

このように最悪の場合の計算回数を考えて、その最も変化の大きな項を取り出したことを示すためにO(オー)記法が用いられる。Fibonacci 数のアルゴリズムであれば次のように書くことができる。

$$\begin{aligned} c_r(k) &= O(\phi^k) \\ c_l(k) &= O(k) \end{aligned}$$

さて、計算量のオーダーは実際の計算時間の近似としてはかなり粗いものであるが、これを使ってアルゴリズムを比べても大丈夫なのだろうか。例え

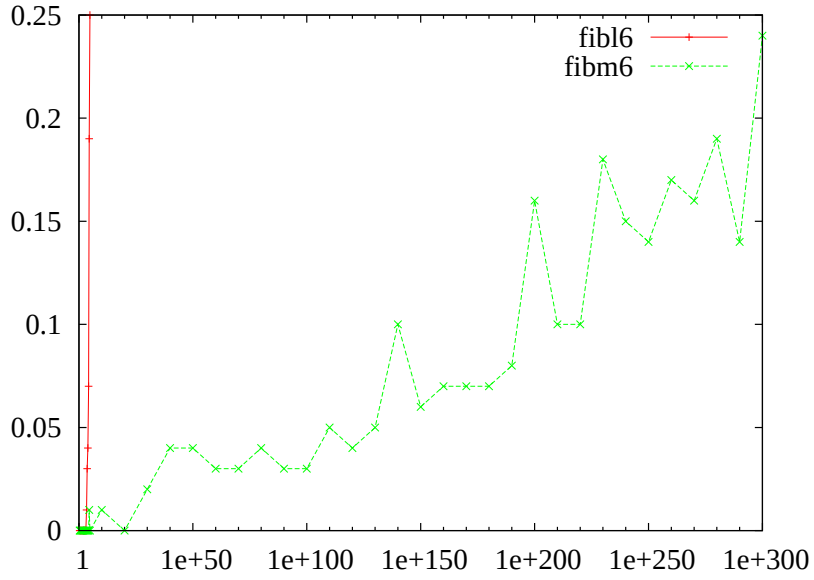


図 5.4: $k \leq 10^{300}$ に対する `fibm6`(k) の計算時間

ば $O(\phi^k)$ と $O(k)$ であれば、$O(\phi^k)$ の方が変化が大きいので悪いアルゴリズムだと結論するわけだが、実際の計算時間は $O(k)$ の方が 100 倍多く操作をしているかも知れず、実際には $O(\phi^k)$ のものの方が速いことがあるかも知れない。

しかし、計算量のオーダーの違いはそれ以上に大きい。下の表は k と ϕ^k の変化を示したものであるが、k が大きくなると ϕ^k は圧倒的に大きな数になる。これならば定数倍の違いを無視しても構わないことが分かるだろう。

k	1	10	100	1000	10000
ϕ^k	1.6	1.2×10^2	7.9×10^{20}	9.7×10^{208}	7.5×10^{2089}

5.3　行列を用いた Fibonacci 数のアルゴリズム

Fibonacci 数を求めるアルゴリズムとして、計算量が $O(\log k)$ のものを紹介しよう。つまり $\mathrm{fib}(k)$ を求めるのに k の桁数に比例した時間しかかからないものである。

これから紹介するアルゴリズムに従って作られた関数 `fibm6` の計算時間を図 5.4 に示す。このグラフは横軸が対数目盛りになっている。グラフがほぼ直線になっていることから、計算時間が $\log k$ に比例していることが分かる。一緒に `fibl6` の計算時間も描かれているが、`fibm6` の方が大幅に速いことも分かるだろう。

5.3.1　行列積による求解

このアルゴリズムでは連続する 2 つの Fibonacci 数をベクトル $v_k = \begin{pmatrix} \mathrm{fib}(k+1) \\ \mathrm{fib}(k) \end{pmatrix}$ として考える。すると $\mathrm{fib}(k)$ の定義から v_k と v_{k+1} の間の関係は行列 $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ を使って $v_{k+1} = Qv_k$ と表わせる。これより v_n は v_0 を Q に n 回掛けたもの、つまり $v_n = Q^n v_0$ と導かれる。

練習 5.6 (行列積による求解の正しさ) $v_{k+1} = Qv_k$ を証明せよ。

5.3.2　冪乗の計算アルゴリズム

ここで行列 Q の冪乗の計算方法について考える。単純には行列の掛け算を n 回行えば Q^n が求まる。しかしこれでは計算量は $O(n)$ になってしまい、数え上げアルゴリズムと変わらない。

実は、もっと計算量の小さな冪乗計算アルゴリズムがある。例として Q^{16} で説明する。$Q^{nm} = (Q^n)^m$ であるので $Q^{16} = (((Q^2)^2)^2)^2$ と変形できる。自乗の計算は 1 回の掛け算でできるので、Q^{16} が 4 回の行列の掛け算でできるということである。単純に計算すれば 16 回の行列の掛け算が必要だったのでかなり回数を減らせている。

この考え方を一般化すると次のような関係として定義できる。

$$Q^n = \begin{cases} E & (n = 0) \\ (Q^{n/2})^2 & (n \text{ は 2 以上の偶数}) \\ Q \times Q^{n-1} & (n \text{ は奇数}) \end{cases} \tag{5.11}$$

(ただし E は単位行列。) つまり、Q^n は n が偶数のときは $Q^{n/2}$ を求めてからそれを自乗し、奇数のときは Q^{n-1} に Q を掛けるという意味である。

この関係を使えばどのような n に対しても少ない回数の掛け算で Q^n を求める式が導ける。例えば Q^{20} であれば

$$Q^{20} \Rightarrow (Q^{10})^2 \Rightarrow ((Q^5)^2)^2 \Rightarrow ((Q \times Q^4)^2)^2 \Rightarrow ((Q \times (Q^2)^2)^2)^2 \Rightarrow$$
$$((Q \times ((Q^1)^2)^2)^2)^2 \Rightarrow ((Q \times ((Q \times Q^0)^2)^2)^2)^2 \Rightarrow ((Q \times ((Q \times E)^2)^2)^2)^2$$

のように展開できる。掛け算の回数は $\times$ と自乗の個数なので 6 回の行列の掛け算で求められている。

練習 5.7 (行列の冪乗) 行列 A の冪乗 A^n を計算する関数 `matpower(a,n)` を次のようにして定義せよ。行列は 2×2 のものだけを扱うことにして、2 次元配列で表わすことにする。

a) 行列 a, b の積を求める matmul(a,b) を定義せよ。ヒント: 行列を 2×2 に限っているので次のような計算をするだけである。

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix}$$

「行列 (matrix) を掛ける (multiply)」ことが分かるように matmul という関数名にしている。matsquare, matpower も同様の方法で名前がついている。

b) 行列 a の自乗を求める matsquare(a) を matmul を用いて定義せよ。

c) 式 5.11 に従って行列 a の n 乗を求める matpower(a,n) を定義せよ。

練習 5.8 (行列を用いた Fibonacci 数の計算) 練習 5.7 で定義した matpower(a,n) を用いて $\mathrm{fib}(k)$ を求める関数 fibm(k) を定義せよ。また、fibm(k) と fibl(k) が同じ答を出すことをいくつかの k について確認せよ。

5.3.3　行列を用いたアルゴリズムの計算量

最後に、この行列を用いたアルゴリズムの計算量を考えよう。今回の計算は Q^k の計算に帰着されるので、冪乗の計算アルゴリズムの計算量を考えればよい。

Q^k が式 5.11 に従って 1 回展開されると、行列の掛け算は 1 回増え、k は (奇数のとき)1 減るか (偶数のとき) 半分になるかのどちらかである。従って k が 0 になるまで何回展開されるか、つまり k がどのように減ってゆくかを考えればよい。k の減り方は一定ではないが、1 番遅い減り方をするのは 奇数 $\Rightarrow$ 偶数 $\Rightarrow$ 奇数 $\Rightarrow$ 偶数 $\Rightarrow \cdots$ のように減ってゆく場合なので、$k = 2^m - 1$ という値になっている場合である。このときに $2m$ 回つまり $2\log_2(k+1)$ 回の展開が起きるのが最悪の回数である。2×2 行列の掛け算 1 回に必要な基本的計算は (桁数が大きくないのであれば) 定数回なので、オーダーを考える場合には正確な回数は考えなくともよい。結局、このアルゴリズムの計算量 $c_m(k)$ は

$$c_m(k) = O(\log k)$$

であることが分かる。

練習 5.9 (行列を用いた Fibonacci 数の計算 (下 6 桁)) 練習 5.8 で定義した関数を変更し、Fibonacci 数の下位 6 桁のみを求める fibm6 を定義せよ。時間計測用の配布プログラムを用いて、大きな k に対する fibm6(k) の計算時間が $\log k$ に比例していることを確認せよ。

5.4　整列のアルゴリズム

沢山のデータを順番に並べ替えることを**整列**(sorting) という。コンピュータを使った情報処理では、大量のデータの整列が頻繁に行われるので、良いアルゴリズムは大量データ処理に不可欠である。

ここでは整列のアルゴリズムの中から簡単なものとして、**単純整列法**と**併合整列法**を紹介する。

以下では、数値がしまわれている大きさ n の配列があったときに、その数値を小さい順に並べ替える問題を考えることにする。

5.4.1　単純整列法

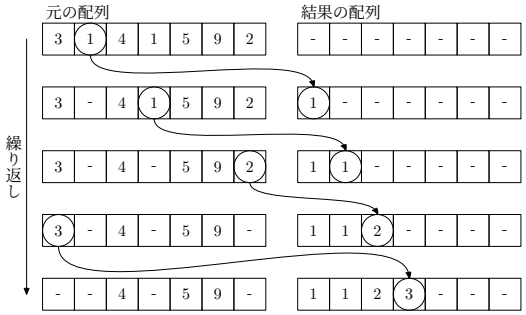


図 5.5: 単純整列法の考え方

単純整列法は図 5.5 のように「元の配列の中で 1 番小さな数を見つけてそれを取り除き、結果の配列の最後に置く。」という作業を元の配列が空になるまで繰り返す方法である。

ただし、図 5.5 のような方法をそのまま使うと、配列を 2 つ用意しなければいけない上に、元の配列では「取り除かれた値」を記録しておく必要がある。

この方法を少し工夫すると、元の配列だけを使って整列ができる。これは、図 5.6 のように「元の配列の中で 1 番小さな数を見つけそれを配列の 0 番目と入れ替える。次に配列の 1 番目以降で 1 番小さな数 (つまり全体では 2 番目に小さな数) を見つけ、配列の 1 番目に場所に置く」という作業を繰り返すものである。図中では、入れ替えが終わったものが灰色に塗られ、白色の中から 1 番小さな数が選ばれて入れ替えられている。

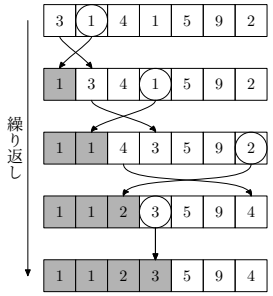


図 5.6: 工夫した単純整列法

この工夫した単純整列法をプログラムにするとファイル 5.3 のようになる。

```
1 def simplesort(a)
2   for i in 0..(a.length()-1)
3     k = min_index(a,i)
4     v = a[i]
5     a[i] = a[k]
6     a[k] = v
7   end
8   a
9 end
```

ファイル 5.3: simplesort.rb (一部)

i 回目の繰り返しでは、0 番目から i − 1 番目までは整列されているので、i 番目以降で最小の値が現われる場所を探し (3 行目の min_index(a,i)) その場所を k とする。元々i 番目にあった値を v とする (4 行目)。k 番目の値、つまり最小の値を i 番目にしまい (5 行目)、元々i 番目にあった値 v を k 番目へどける (6 行目) と入れ替えることができる。

練習 5.10 (単純整列法) 配列 a の (先頭を 0 番目としたときの)i 番目以降の値の中で、最小値が出現する番号を答える min_index(a,i) をファイル 5.3 に追加して、単純整列法を完成させよ。

```
1 irb(main):004:0> a=[3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3]
2 => [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3]
3 irb(main):005:0> min_index(a,0)
4 => 1
5 irb(main):006:0> min_index(a,4)
6 => 6
7 irb(main):007:0> simplesort(a)
8 => [1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9]
```

単純整列法の計算量

配列の大きさを n として単純整列法の計算量を考えてみよう。このアルゴリズムは全体で n 回の繰り返しをしている。i 番目の繰り返しでは $n-i$ 個の値から最小値を見つける必要があるので、$n-i$ 回の繰り返しが必要になる。つまり、全体では

$$\sum_{i=0}^{n-1}(n-i) \quad = \quad n+(n-1)+\cdots+1 \quad = \quad \frac{1}{2}n(n-1)$$

回の繰り返しということになる。従って**単純整列法の計算量は $O(n^2)$ である**。

5.4.2　併合整列法

単純整列法の計算量は $O(n^2)$ だった。これは、1000 万個のデータを並べ替えるには 1000 万の自乗、つまり 100 兆回程度の繰り返しをしなければいけないということである。コンピュータが 1 秒間に 10 億回の繰り返しを行えるとしても、10 万秒も時間がかかってしまう。

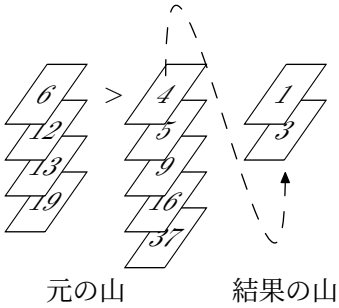


図 5.7: 2 つのカードの山の併合

もっと良い計算量の整列法の 1 つとして**併合整列法**を紹介しよう。この方法の基本は図 5.7 のように、小さい順に並んでいる 2 つのカードの山を混ぜ合わせて小さい順に並んだ 1 つの山にする、**併合**(merge) という作業である。元の 2 つの山は小さい順に並んでいるので、この作業は 1 番上にあるカードの値を比べて、小さい方のカードを結果の山の 1 番下に移すことを繰り返してゆき、一方の山のカードがなくなったらもう一方の山の残りのカードをそのまま結果の山の 1 番下につなげる。これで小さい順に並んだ結果の山を得ることができる。

予め小さい順に並んだ山が 2 つあれば併合によって整列された山を作れることは分かった。この「小さい順に並んだ山」はどうやって作るのかというと、やはり併合によって作ればよい。つまり、図 5.8 の左端の列のように、バラバラに並んでいるカードそれぞれをカード 1 枚だけの山だと思い、2 つずつを併合して 2 枚のカードの山を作る。できた山を再び 2 つずつ併合して 4 枚のカードの山を作る。この手順を繰り返してゆけば図の右端のように 1 つの小さい順に並んだ山になるというわけである。

併合整列法の計算量

プログラムを作るより前に、併合整列法の計算量を考えておこう。この方法は、併合を何度も行っているので、一見無駄なことをしているようにも思えるが、果たして単純整列法よりも速いのだろうか。

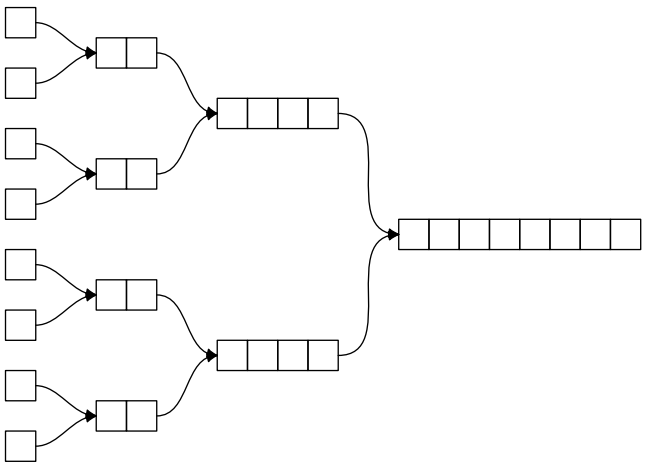


図 5.8: 併合の繰り返し

まず併合の計算量を考える。ℓ 枚のカードの山を 2 つ併合するには、2 つの山の 1 番上のカードを比較して、どちらかのカードを移動させることを 2ℓ 回繰り返すだけである。

次に、1 枚のカードの山が n 個ある状態から、併合を繰り返して n 枚のカードの山 1 個にするまでの手順を考えよう。以下、簡単のために $n = 2^e$ だとする。すると、(最初を 1 回目だとして)i 回目の繰り返しは2^{i-1}枚の山を2つ併合して2^i枚の山を得ることを 2^{e-i} 個の組について行うことになる。下線の作業は 2^i 回の繰り返しであるので、結局、i 回目の繰り返しは $2^i \times 2^{e-i} = 2^e = n$ に比例した作業を行うことになる。この繰り返しは i が 0 から $e-1$ までの e 回行われるので、全体では $n \times e = n \log_2 n$ 回の繰り返しとなる。つまり、**併合整列法の計算量は** $O(n \log n)$ であり、単純整列法の $O(n^2)$ よりも良いことが分かる。

例えば 1000 万個のデータを並べ替えを考えると、1000 万 $\times \log_2$(1000 万) $\simeq$ 2 億 3000 万 回程度の繰り返しになる。単純整列法は $O(n^2)$ だったので 100 兆回程度の繰り返しをしなければいけなかったので、劇的に速くなっていることが予想できる。

併合整列法のプログラム

さてそれでは併合整列法を Ruby の関数として定義してみよう。このアルゴリズムでは「カードの山」を使っていたが、これは Ruby の配列として表わすことにする。

2 つの配列 a, b を併合して 1 つの山にする関数 `merge(a,b)` はファイル 5.4 のように定義できる。

この関数では、まず a と b の合計の長さを持つ配列 c を作る (2 行目)。次に、

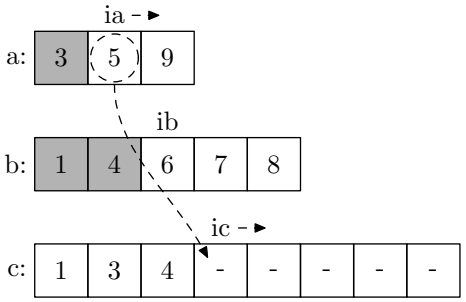
```
1  def merge(a,b)
2    c = Array.new(a.length()+b.length())
3    ia=0
4    ib=0
5    ic=0
6    while ia < a.length() && ib < b.length()
7      if a[ia] < b[ib]
8        c[ic] = a[ia]
9        ia = ia + 1
10       ic = ic + 1
11     else
12       c[ic] = b[ib]
13       ib = ib + 1
14       ic = ic + 1
15     end
16   end
17   a,b のうちまだ残っている方を全て c に移す (省略)
18   c
19 end
```

ファイル 5.4: mergesort.rb (前半)

- 配列 a から移動した値の個数を覚える変数 ia、
- 配列 b から移動した値の個数を覚える変数 ib、
- 配列 c に移動された値の個数を覚える変数 ic

を用意する。

あとは「配列 a, b のどちらか一方から配列 c に値を一つ移動させる」作業の繰り返しである (6 行目から 16 行目)。繰り返し 1 回の作業は、次の図のように配列 a と b の (先頭を 0 番目として)ia, ib 番目の値を比べ (7 行目)、小さい方を c の ic 番目に移し、ia, ib の移動した方と ic をそれぞれ 1 増やす (8 行目から 14 行目)。



この繰り返しは、a と b のどちらか一方の値をすべて移し終えるまで (つまり 6 行目の条件のように a と b の両方にまだ移していない値が残っている限り) 続ける。

16 行目までの繰り返しが終わったときは、配列 a または b のどちらか一方の値はすべて移し終え、もう一方にはまだ値が移し終わっていない状態になる。(上の図であれば配列 b の値をすべて移し終わり、a には最後の 9 が残る。) そこで、値が残っている方の配列の残りの値をすべて c に移す繰り返しを行う。ファイル 5.4 では 17 行目の部分だが、省略してある。

最後に配列 c を答として返す (18 行目)。

練習 5.11 (併合) ファイル 5.4 の省略された部分を補って関数 merge を完成させよ。

```
1 irb(main):004:0> merge([3,5,9],[1,4,6,7,8])
2 => [1, 3, 4, 5, 6, 7, 8, 9]
3 irb(main):005:0> merge([0,0.5,1.0],[0,0.9,1.0])
4 => [0, 0, 0.5, 0.9, 1.0, 1.0]
```

バラバラの順序の配列から、併合を繰り返して整列する関数 mergesort(a) はファイル 5.5 のようになる。

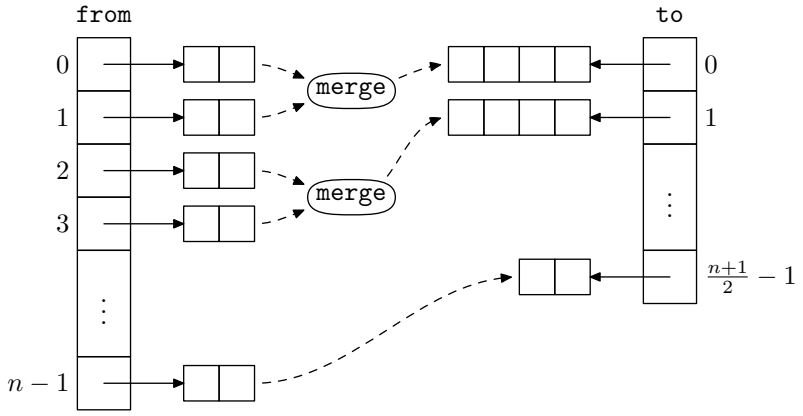
この関数の前半 (25 行目まで) は「カード 1 枚ずつの山」を作っている部分である。与えられた配列 a の大きさを n として、n 行 1 列の配列 from を作り、from[i][0] に a[i] の値をしまっている。24 行目 の式 [(a[i])] は、大きさ 1 の配列を作ることに注意してほしい。

後半は、from にしまわれている n 個の配列を 2 つずつ merge によって併合し、配列の個数を半分に減らしてゆく作業を繰り返している。

繰り返しの各回の作業を次の図に従って説明しよう。

```
20 def mergesort(a)
21   n = a.length()
22   from = Array.new(n)
23   for i in 0..(n-1)
24     from[i] = [ (a[i]) ]
25   end
26   while n > 1
27     to = Array.new((n+1)/2)
28     for i in 0..(n/2-1)
29       to[i] = merge(from[i*2], from[i*2+1])
30     end
31     if !is_even(n)
32       to[(n+1)/2-1]=from[n-1]
33     end
34     from=to
35     n=(n+1)/2
36   end
37   from[0]
38 end
```

ファイル 5.5: mergesort.rb (後半)



- まず、from の半分の大きさの配列 to を作る。式 (n+1)/2 は n が奇数のとき、n を 2 で割って切り上げた値になる。(図は n が奇数の場合である。)
- 次に from の 2i 番目と 2i + 1 番目を merge によって併合し、to の (先頭を 0 とした)i 番目にしまう (29 行目)。この作業は i が n/2-1 になるまで繰り返すことになっている (28 行目) が、n が奇数のときは余りが

切り捨てられるので、from[n-2] までが併合されて、from[n-1] は処理されないことになる。

- n が奇数のときは、from の最後の配列を to の最後にしまう (32 行目)。
- このようにして作られた配列の配列 to を、次回の from にし (34 行目)、長さ n を半分 (の切り上げ) にする (35 行目)。

最終的に n が 1, つまり 1 個の配列に併合されたら from[0] にしまわれている配列が整列された結果となる (37 行目)。

練習 5.12 (併合整列法の実行) 練習 5.11 で完成させた merge とファイル 5.5 をあわせて、併合整列法による整列を行ってみよ。

```
1 irb(main):007:0> a=[3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3]
2 => [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3]
3 irb(main):008:0> mergesort(a)
4 => [1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9]
```

練習 5.13 (整列法プログラムの速度の比較) 単純整列法と併合整列法のプログラムを見比べると、併合整列法では併合のための配列を作るといったより複雑な作業をしている。そのため、配列が短い場合は simplesort の方が、長くなると mergesort の方が速いことが予想される。以下のようにして実際の計算時間を比較せよ。

a) 整列させる配列を作るために、配布プログラム randoms.rb に含まれている randoms(id,size,max) を使う。

関数 randoms の詳しい説明は第 5.5 節にある。

b) 計算時間を測るために、第 5.2.1 節で紹介した配布プログラム bench.rb の関数 run(f,x,v) を使う。

この run の使い方は第 5.2.1 節のときと少し違う。詳しくはやはり第 5.5 節を参照せよ。

c) 2 つの整列法プログラム比較する関数 compare_sort(m,s) を次のように定義する。この関数は、大きさが $s, 2s, 3s, \ldots, m$ であるようなデタラメな配列を順に 2 つずつ作り、simplesort と mergesort を使って整列させて時間を測定している。

この compare_sort(max,step) を使って、計算時間の違いを実際に比較せよ。

run が測定する時間の精度は高くないので、正確な比較のためには、同じ計算を 1000 回繰り返すような関数を作り、その計算時間を計るといった工夫が必要になる。

5.5　定義のまとめ

計算時間の測定: (配布プログラム bench.rb) 関数 run は、関数の計算時間を測定し、それをグラフにする。

```
load("./randoms.rb")     # randoms(id,size,max)
load("./bench.rb")       # run(function_name, x, v)
load("./simplesort.rb") # simplesort(a)
load("./mergesort.rb")  # mergesort(a)

def compare_sort(max, step)
  for i in 1..(max/step)
    x=i*step
    a=randoms(i,x,1)
    run("simplesort", x, a)
    a=randoms(i,x,1)
    run("mergesort", x, a)
  end
end
```

ファイル 5.6: compare_sort

- run("fibr",n) のように実行した場合は、fibr(n) の計算を行い、その際の計算時間を X 座標が n の位置に表示する。
- run("fibr",x,n) のように実行した場合は、fibr(n) の計算を行い、その際の計算時間を X 座標が x の位置に表示する。

グラフの表示方法の変更: (配布プログラム bench.rb) command という命令を用いてグラフの表示方法を変更できる。主な命令には次のものがある。

- command("set logscale y") — グラフの Y 軸を対数スケールにする。y を x にすると X 軸が対数スケールになる。
- command("unset logscale") — 対数スケールをやめる
- command("set xrange [a:b]") — X 軸の範囲を a から b までにする。xrange を yrange にすると Y 軸の範囲を変更する。
- command("set autoscale") — 表示範囲を自動的に変更する。

実際には" "の内側は、グラフの表示に用いている GNUPLOT というソフトウェアに対する命令である。他の命令については GNUPLOT のマニュアルなどを参考にしてほしい。

また reset() という命令を実行すると、記録していたデータを全て消去して、以降の run から新しいグラフを作成する。

デタラメな中身の配列を作る: (配布プログラム randoms.rb) 関数 randoms(id,size,max) は、大きさ size の配列を作る。中身は 0 以上 max 未満のデタラメな値

であり、max が 1 のときは実数、1 より大きいときは整数になる。整数 id は、系列番号を意味し、id, size, max がすべて同じであれば同じ内容の配列が作られ、id が違えば異なる内容になる。

5.6　章末問題

練習 5.14 (組み合わせ数の計算量)　a) 組み合わせ数の計算を練習 4.2(p. 52) の定義に従って再帰的に行う場合の計算量を求めよ。

b) 同じ計算を第 4.3.1 節のように繰り返しによって行う場合の計算量を求めよ。

練習 5.15 (言葉探しの計算量) 文字列 s 中に文字列 p が現われる位置を探す方法として、第 4.3.2 節 (p. 55) で紹介したものがある。s と p の長さ n, m を入力の大きさだとして、これの計算量を n, m で表わせ。

ヒント: 最悪の場合を考えればよい。s が「$\overbrace{\texttt{aa}\cdots\texttt{a}}^{n-1}\texttt{b}$」、p が「$\overbrace{\texttt{aa}\cdots\texttt{a}}^{m-1}\texttt{b}$」という文字列だったときの submatch と match の繰り返し回数を考えよ。

練習 5.16 (同じ誕生日の人達) 同じ年に生まれた n の学生がいる。この中には、同じ誕生日を持つ学生と、そうでない学生がいる。前者が何人いるか数えるアルゴリズムをいくつか考え、その計算量を考えよ。簡単のために、誕生日は 1 月 1 日からの日数 (つまり 0 から 364 の整数) で表わすことにする。また、学生の誕生日は大きさ n の配列 b に入っているが、その順序はデタラメだとする。

アルゴリズム 1: 同じ誕生日を持つ学生の人数を変数 m で表わすことにして、最初は 0 にしておく。学生の誕生日を 1 人ずつとり出し、配列 b のすべての中身と比較して同じ誕生日の学生がいるかどうかを調べ、1 人でも同じ誕生日の者がいた場合は m を 1 増やす。これをすべての学生について行う。

アルゴリズム 2: 同じ誕生日を持つ学生の人数を変数 m で表わすことにして、最初は 0 にしておく。配列 b を併合整列法によって小さい順に並べる。並べ換えた配列を先頭から順に調べてゆき、同じ値が連続する区間の長さ k を数える。k が 2 以上の場合の区間があった場合は、m を k だけ増やす。これを配列の最後まで行う。

アルゴリズム 3: 日付ごとの人数を数えるために、大きさ 365 の配列 c を作り、中身をすべて 0 にしておく。学生の誕生日 d を 1 人ずつとり出し、その日付の人数 c[d] を 1 増やす。これをすべての学生について行う。最後に、c を先頭から調べて 2 以上になっている値の和を求める。