平成10年度(第20回)数学入門公開講座テキスト(京都大学数理解析研究所，平成10年8月3日～8月7日開催)

数学入門公開講座

平成１０年８月３日（月)から８月７日(金)まで

京都大学数理解析研究所

講師及び内容

1．無理数、超越数（6時間15分）

京都大学数理解析研究所・助手　永田　誠

$\sqrt{2}$ は無理数、円周率 π は超越数、というのは良く知られています。$\sqrt{2}$ が無理数であるということはギリシャ時代には既に知られていました。一方 π が超越数であるということが証明されたのは約百年前のことです。現在でもある数が超越数（または無理数）であるかどうかを判定することは大変難しい問題です。

本講座では現在この分野でどのようなことが知られているのかということの紹介等を話題に進めていく予定です。

2．微分方程式と発散級数（6時間15分）

京都大学数理解析研究所・助教授　竹井　義次

「収束」の概念が確立された現代の解析学においては、収束しない発散級数は、取り扱いの難しい厄介な代物と考えられがちです。しかしその一方で、微分方程式を解く過程でしばしば発散級数が現れます。では、こうした発散級数で表される解には何の意味もないのでしょうか。面白いことに事実は全く逆で、収束する解と比べても遜色ないほど多くの情報を発散級数解は含んでいるのです。この講義では、発散級数を論じる際の基本的な道具である「漸近展開」や「Borel総和法」の解説を行いながら、モノドロミー群といった微分方程式の解の大域的な性質と発散級数解との関わりについて考えてみたいと思います。

3．再帰的構造とアルゴリズム（6時間15分）

京都大学数理解析研究所・助手　西村　進

自然数の階乗を求める関数 f は

$$f(n) = \begin{cases} 1 & (n = 0\text{のとき}) \\ n \times f(n-1) & (n > 0\text{のとき}) \end{cases}$$

のように定義できるが、このような定義を、自分自身の定義を使った定義という意味で、再帰的定義という。このような再帰的関数定義は、自然数に限らず、リストや木などの再帰的構造を持つようなデータの処理を、簡潔にかつわかりやすく記述するのに非常に有効である。

本講座では、再帰的プログラミングの入門から始めて、いくつかのアルゴリズム（問題を解く手順）を、再帰的プログラミングの観点から紹介する。

時　間　割

時間＼日	8月3日(月)	4日(火)	5日(水)	6日(木)	7日(金)
10：30～11：45	永田	永田	永田	永田	永田
11：45～13：00	休　憩				
13：00～14：15	竹井	竹井	竹井	竹井	竹井
14：15～14：45	休　憩				
14：45～16：00	西村	西村	西村	西村	西村

平成10年度(第20回)数学入門公開講座テキスト(京都大学数理解析研究所, 平成10年8月3日～8月7日開催)

再帰的構造とアルゴリズム

京都大学数理解析研究所・助　手　　西　村　　進

1998, AUGUST 3,4,5,6,7,　14:45 ~ 16:00

再帰的構造とアルゴリズム

京都大学数理解析研究所　西村 進

1　はじめに

アルゴリズムとは、ある特定の問題に関する回答を得るための具体的な手続きのことをいう。アルゴリズムのうちでも、コンピュータによる計算および手続きによって実現されるものをコンピュータ・アルゴリズムといい、本講義ではアルゴリズムと言えばコンピュータ・アルゴリズムを指すものとする。アルゴリズムの典型的な例としては、与えられた数列から最大値や最小値を見つけ出すアルゴリズムや、与えられたデータを検索するアルゴリズム、数列を並べ替えるアルゴリズムなどがある。

アルゴリズムに関する研究の第一の目的は、アルゴリズムの性能向上である。すなわち、同じ問題に対してより効率良く（より速く）その問題を解くことのできるアルゴリズムを発見することである。そのようなアルゴリズムを発見できれば、そのアルゴリズムを従来のものの代わりに使うことによって、プログラムの効率を改善することができる。

これまでのアルゴリズムの研究において、様々な問題に対していろいろな効率のよいアルゴリズムが発見・提案されてきている。しかし、これらの結果は、一見したところやや難解に見えるためか、残念ながら一般にはあまり使われていなかったり、使われていたとしてもその背景（なぜそのアルゴリズムはより効率よく計算を行えるのか）についてはあまり気にせずに使われているようである。

本講義では、典型的な問題を解くいくつかの代表的なアルゴリズムを紹介するが、個々のアルゴリズムをただ順番に解説していくのが目的ではない。よいアルゴリズムには必ずといっていいほど、背後にそのアルゴリズムのアイディアのもととなっているような数学的原理が存在し、実は多くのアルゴリズムの原理はその中に潜む「再帰的構造」に着目することによって理解することができる。再帰的構造とは、例えば次のようなものである。自然数の階乗 $(n!)$ を求める関数 f は

$$f(n) = \begin{cases} 1 & (n = 0 \text{ のとき}) \\ n \times f(n-1) & (n > 0 \text{ のとき}) \end{cases}$$

のように定義できるが、このような定義を、自分自身の定義を使った定義という意味で**再帰的関数定義**といい、このような再帰的に定義された関数のことを**再帰的関数**という。このような再帰的関数は再帰的構造の一例である。プログラム中に現れる他の重要な再帰的構造としては、あとに説明するようなリストや木などの再帰的なデータ構造がある。

本講義では、このような再帰的構造の観点から各々のアルゴリズムを見つめ直し、各々のアルゴリズムの基礎となっている数学的な原理について解説する。また、そのような数学的な原理が、効率のよい、美しいアルゴリズムの設計に寄与していることが理解していただけると思う。

2 プログラミング言語とデータ構造

コンピュータに対して実行すべきアルゴリズムを正確かつ簡潔に示すためには、**プログラミング言語** と呼ばれる特別な言語を用いる。本講義では、アルゴリズムを記述するためのプログラミング言語としてMinimal[1]という言語を用いる。

アルゴリズムを記述する際に問題となるのは、入力データあるいは出力データをコンピュータ上でどのように表すかということである。コンピュータ上でのデータの表現方法は**データ構造** と呼ばれる。データ構造は、アルゴリズムの効率やアルゴリズムの設計そのものに大きな影響を及ぼす。データ構造については、この章の後半で述べることにする。

2.1 プログラミング言語Minimal

Minimalは**関数型言語** を基本として、それにCやPascalなどのような**手続き型言語**の要素を付け加えたものとした。関数型言語におけるプログラミング言語では、再帰的関数を使ってプログラミングすることにより、プログラムの持つ再帰的構造をより明確に表現することができる。言語の核として関数型を選んだのは、このためである。

第1節で述べたように、本講義の目的は、いくつかのアルゴリズムの数学的な原理を再帰的構造の観点から明らかにすることであるから、本当は再帰的に定義された関数さえ定義できれば十分で、手続き型言語の要素を含める必要はない。しかし、後に再帰的構造を考慮せずに設計したアルゴリズムと比較するために、手続き型的なプログラミングもできるようにしておく。

これからしばらく、Minimalでのプログラミングについて解説していくが、Minimalのすべてについて解説する必要はないので、ごく基本的な部分のみ説明することとする。もし、アルゴリズムを記述する際にこれ以上の事項が必要になった場合はその都度説明することとする。

2.1.1 手続き型言語としてのMinimal

Minimalのごく簡単なプログラムを記述すると、次のようになる。

```
var A = 0;

for i=1 to 100 do
    A <- A+i;
```

上記プログラムにおいて、最初の`var A=0;`は**変数宣言**といわれるもので、コンピュータ内部にひとつ箱を用意し、その箱に`A`という名前をつけ、箱には最初の値として0を入れておく（これを**初期値**という）ということを意味している。このような変数宣言で定義される`A`を**変数**という。次に

```
for i=1 to 100 do
  式;
```

は変数`i`を$1,2,3,\ldots$と増やしながら`i`が100になるまで、100回式を実行することを意味している。最後に、`A <- 式;`というのは変数`A`によって表されるコンピュータ内部の箱に式を計算した値を書き込むことを表す。これを変数`A`に対する**代入**という。上のプログラムでは`A <- A+i`によって、変数`A`に変数`A`の元の値に`i`を足した結果を変数`A`に代入している。なお、複数の式をループの中で実行したいときには、`;`（セミコロン）で区切った式を次の例のように**begin**と**end**で囲めばよい。

[1]プログラミング言語Minimalは、京都大学数理解析研究所で開発した教育用言語である。Minimalという名前はMini Meta Algorithmic Languageに由来する。

```
for i=1 to 100 do
  begin
    A <- A+i;
    A <- A/2
  end;
```

条件による手続きの選択や、手続きの繰り返しを表現するのが**制御構造**とよばれるものである。

手続きの繰り返しを表現する方法には２通りある。ひとつは、いま紹介した for ループであり、もうひとつは、while ループである。[2]

条件によって次に実行する手続きを選択するのが、if 文である。

```
if 条件 then 式1 else 式2;
```

if 文は条件が真であれば式１を実行し、偽であれば式２を実行する。たとえば、

```
if A>0 then A <- A+1 else A <- A-1;
```

は、もし変数 A の値が０より大きければ変数の値を１増やし、そうでなければ逆に１減らす。

なお、プログラム中に**注釈**（プログラムの実行には関係ないが、人間があとでプログラムの動作を思い出したり、ここではプログラムがどういう動作を意図しているのか示すために付け加えるもの）を加えるには注釈を (*と*) で囲めばよい。

2.1.2 Minimal による関数型プログラミング

関数型プログラミングにおける関数というのは、数学における関数とおおよそ同じものである。すなわち、関数 $f(x_1, x_2, \ldots, x_n)$ は n 個のデータ $x_1, x_2, \ldots, x_n$(**引数**) を受け取って、あるひとつの値（**返り値**）を計算して返す。Minimal では、すでに定義されている関数や演算子を組み合わせることによって、自由に関数を定義することができる。たとえば、

```
fun f x = x*x;
```

は整数を受け取って、それの二乗を返す関数 $f(x) = x \times x$ を定義する。Minimal では、このように一度関数を定義すると、その関数をすでに定義されていた関数と全く同じように使うことができる。

```
f 11;
- :  int = 121 (←—答)
```

このように関数に入力を与えて計算させることを**関数適用**あるいは**関数呼び出し**という。n 引数の関数の関数定義と関数適用は一般に次のように表される。

関数定義:　fun 関数名 引数 1 引数 2 ··· 引数 n = 式
関数適用:　関数名 式 1 式 2 ··· 式 n

例えば、２つの整数を引数として受け取り、それらのうちの小さくない方を返す関数max は

```
fun max x y = if x>y then x else y;
```

と定義でき、それを使うと次のような計算ができる。

```
max (f 12) 20;
- : int = 144
```

[2] while ループは通常の手続き型言語のものと同じもの。本講義では出てこないので説明しない。

関数型言語でのプログラミングとは、このようにして必要な関数を定義していくことである。

関数型言語でのプログラミングでは、基本的に変数を使う必要はない。しかし、時には一時的に値を保存しておく変数のようなものがあると便利なことがある。Minimalでは変数宣言に似た、**値宣言** とよばれる宣言がある。たとえば、

```
val 名前 = 値;
```

と宣言すると、それ以降は宣言した名前はその値を表すために使うことができる。ただし、ここで宣言した名前は変数と違い、あとから代入を行って値を書き換えることはできない。したがって、代入の必要がないときは var の代わりに val を使うのがよい。[3]

2.1.3 再帰的な関数定義

再帰的関数とは、その定義の中で自分自身を呼び出している関数のことである。たとえば、1から n までの整数を足し合わせる関数 sum は再帰関数によって次のように定義できる。

```
fun sum n =
  if n=1 then 1 else n+(sum (n-1));
```

関数 sum に例えば 3 を与えると次のような計算が行われる。

$$\begin{array}{ll} & \mathrm{sum}\ 3 \\ \leadsto & 3 + (\mathrm{sum}\ 2) \\ \leadsto & 3 + (2 + (\mathrm{sum}\ 1)) \\ \leadsto & 3 + (2 + 1) \\ \leadsto & 6 \end{array}$$

この再帰的定義は、次のような数学的帰納法による関数定義と対応している。

1. $n = 1$ のとき、$sum(n) = 1$。
2. $sum(n-1)$ が定義されているとすると、$sum(n) = n + sum(n-1)$。

再帰関数によるプログラミングはこのような数学的構造との対応を持っている。一般に、関数型言語でこのような再帰的構造に注目してプログラミングを行うことで、より見通しの良いアルゴリズムを書くことができる。

2.2 データ構造

この章の最初でも述べたように、大量のデータをコンピュータ上で表現するためにどのようなデータ構造を採用するかは、アルゴリズムの記述に関して非常に重要である。Minimalでは、大きく分けて**静的**なデータ構造と、再帰的な構造を持つデータとして定義できる**動的**なデータ構造の2種類のデータ構造を使うことができる。大まかに言って、手続き型プログラミングは静的なデータ構造を扱うのに、関数型プログラミングは動的なデータ構造を扱うのに、それぞれ適している。

2.2.1 配列—静的なデータ構造

静的なデータ構造とは、プログラムの実行中にそのサイズや形が変更できないものをいう。そのようなデータ構造の代表的なものは**配列** である。配列とは、データを一列に並べ、各データに順

[3] このように二種類の宣言が存在するのは、関数型言語の中で代入可能な変数をうまく扱うための工夫である。この話題は、しかし、本講義の範囲を越えるのでここでは扱わない。

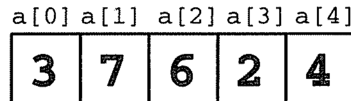

図 1: 配列（コンピュータ内部での表現）

番に番号（インデックスともいう）をつけたものであり、コンピュータ内部では、図1のように、連続した箱（箱を切り離したり、間に新しい箱を入れることはできない）で表される。Minimalでは配列は次のように表す。

```
val a = [| 3, 7, 6, 2, 4 |];
```

このとき、変数aは配列全体を表す変数で、配列のサイズは5で、各データ3,7,6,2,4はインデックス0,1,2,3,4でそれぞれ指し示される。配列aのインデックスiで表される要素を読み出すには

```
a.[i];
```

のように書く。(配列中の前からi番目の要素はインデックス$i-1$で指し示されていることに注意。例えば、

```
a.[2];
- : int = 6
```

となる。)

配列を使った簡単なプログラムの例を示そう。次に示すのは、配列aのすべての配列要素の値を足し合わせた値を変数sumに代入するプログラムである。

```
var sum = 0;

for i=0 to length(a)-1 do
  sum <- sum + a.[i];
```

ここでlength(a)は配列aのサイズを表す。

2.2.2 リスト—動的なデータ構造

配列のような静的なデータ構造では、プログラミングの実行中にデータのサイズや形を変更することはできない。データのサイズや形を変更しなければならない場合は、動的なデータ構造を用いる必要がある。リスト は、このような動的なデータ構造を表すための代表的なデータ構造である。

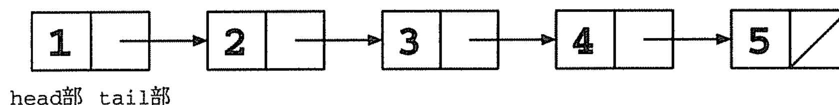

図 2: リスト（コンピュータ内部での表現）

リストはコンピュータ内部では、図2のように表される。リストによるデータ表現においては、リストの各データ要素は、データを保存するための箱と、次の要素を指し示すための矢印（ポイ

ンタという）のための箱の、2つの箱で表される。データを保存するための箱を **head** 部、ポインタのための箱を **tail** 部といい、この2つの箱を合わせて **cons** セルという。

このような表現を取ることによって、リストの先頭の cons セルから tail 部に書き込まれたポインタを順にたどることによって、リストのすべての要素を辿っていくことができる。また、データの削除や追加も可能である。たとえば、リストの先頭にデータ 0 を追加するには、新しい cons セルを表す2つの箱を確保し、head 部に 0、tail 部には元のリストの先頭の cons セルへのポインタを書き込めばよい。

このように、次のデータがどこに置かれているかをポインタで表しているようなデータ構造を一般に、**リンクデータ構造**という。このようなリンクデータ構造を扱うプログラムを手続き型言語で書こうとすると、実はかなりの技術を要する。(プログラミング言語 C でポインタを使ったプログラムを書いたことがあれば、その苦労がよくわかるだろう。) しかし、リンクデータ構造を扱うのに関数型言語を使うと、驚くほど簡単に美しくプログラムを書くことができる。なぜなら、多くのリンクデータ構造は実は再帰的な構造を持っているからである。

2.2.3 再帰データ構造と再帰関数

関数型言語ではふつう、リンクデータ構造をそれと同等の**再帰データ構造**として扱う。Minimal でリストを使いたいときは次のような宣言を行って、リストがどのような再帰的なデータ構造を持つのか定義してやる必要がある。

```
type list = Nil | Cons of int*list ;
```

(int は整数全体を表し、*は2つの値の組、|は「または」をあらわす。) この宣言（**データ型宣言**という）は、list というデータ構造は、整数とほかのリストとの組を表す cons セルであるか、もしくは空のリストを表す Nil である、と定義している。これをもう少し数学的に書くと、

1. Nil はリストである。
2. n が整数、l がリストならば、Cons(n,l) もリストである。

のような定義になる。すなわち、

$$\texttt{Nil} \quad \texttt{Cons}(1,\texttt{Nil}) \quad \texttt{Cons}(3,\texttt{Cons}(1,\texttt{Nil})) \quad \texttt{Cons}(0,\texttt{Cons}(3,\texttt{Cons}(1,\texttt{Nil})))$$

はすべてリストである。

再帰的に定義されたリストが、同じく再帰的な構造を持つ再帰関数によって簡単に処理できることは容易に予想できるだろう。例として、次にリスト中のすべての要素の和を計算する関数 sum を示す。

```
fun sum lst =
  case lst of
      Nil => 0
    | Cons(head,tail) => head + (sum tail)
  end;
```

このプログラムの中で使用している case 文は、パターン・マッチングによる、条件分岐を表している。case 文は一般には次のような文法

```
case 式 of
    パターン 1 => 式1
  | パターン 2 => 式2
    ⋮
  | パターン n
end;
```

リストの表現	元の表現
[]	Nil
head::tail	Cons(head,tail)
[e_1,e_2,...,e_n]	Cons(e_1,Cons(e_2,...,Cons(e_n,Nil)$\cdots$))

表 1: リストを表すための文法

で表される。case 文はまず式の値を求め、その値が各パターンとマッチするかどうか上から順番に調べていく。はじめてパターンがマッチしたところでそのパターンの式（パターン k でマッチしたとすれば、式 k) を求める。上の関数 sum の例では、もし lst が Cons(n,...) の形をしていれば、head が n、tail がリストの残りの部分を表すとして head + (sum tail) を計算する。もし lst が Nil の形をしていれば、0 を返す。このように、再帰データを扱うプログラムを関数型で書くときは、再帰データの再帰的な構造を再帰関数によって辿りながら、データのパターンによって処理を振り分ける、というのが定石である。

リストというデータ構造は非常に頻繁に使われるので、Minimal ではリストは予め定義されており、またプログラムを読みやすくするためにリストを記述するための特別な文法を用意している。(表 1) ここでも以下では、リストのための特別な文法を採用することとする。たとえば、上にあげたリストの和を求める関数 sum は、この文法を使うと次のように書き換えることができる。

```
fun sum lst =
  case lst of
      [] => 0
    | head::tail => head + (sum tail)
  end;
```

再帰関数はリスト上での計算を定義できるばかりでなく、リストそのものに対する操作を定義することもできる。例えば、ふたつのリストをつなぎ合わせたリストを作る関数 append は

```
fun append lst1 lst2 =
  case lst1 of
      [] => lst2
    | head::tail => head::(append tail lst2)
  end;
```

また、リストのデータを逆順に並べたリストを作る関数 reverse は次のように書くことができる。

```
fun make_rev lst revlst =
  case lst of
      [] => revlst
    | head::tail => make_rev tail (head::revlst)
  end;

fun reverse lst = make_rev lst [];
```

のようにして定義できる。

3 計算量とオーダー記法

アルゴリズム同士の性能を比較するにはどのようにしたらよいだろうか。真っ先に思いつく方法はそのアルゴリズムを実際に実行してみることである。しかし、このような方法には限界がある。

ほとんどのプログラムの動作は入力データに依存するから、たとえ何通りかの入力である時間以内に計算を終えることができたとしても、他の入力でもその時間内に計算が終了するとは限らない。データの量を2倍にしたからといって実行時間が2倍になるなどという保証はどこにもない。

いま、同じ問題を解く3つのアルゴリズムA、B、Cがあったとしよう。各アルゴリズムは N 個のデータに関してそれぞれ

アルゴリズムA $\dfrac{10^{-1}}{S} \times 2^N$ μ秒

アルゴリズムB $\dfrac{10}{S} \times N^2$ μ秒

アルゴリズムC $\dfrac{10^6}{S} \times N$ μ秒

で回答を計算することができるとしよう。ただし S は、コンピュータが 1μ秒で実行できる命令の数、すなわちコンピュータの実行速度を表す指数である。(S が大きいほど性能がよい。) いま、仮に $S = 10^3$ としよう。データの数が比較的小さい場合、例えば $N = 10$ のときは、各アルゴリズムはそれぞれ

アルゴリズムA $\dfrac{10^{-1}}{10^3} \times 2^{10} = 0.1024$ μ秒

アルゴリズムB $\dfrac{10}{10^3} \times 10^2 = 1$ μ秒

アルゴリズムC $\dfrac{10^6}{10^3} \times 10 = 10000$ μ秒

のような計算時間を要する。この場合は、計算時間はA $<$ B $<$ Cとなっている。ところが、データの数が比較的大きい場合、例えば $N = 100$ のときは、各アルゴリズムはそれぞれ

アルゴリズムA $\dfrac{10^{-1}}{10^3} \times 2^{100} \approx 10^{26}$ μ秒 $\approx 10^{12.5}$年 ≈ 3.16 兆年

アルゴリズムB $\dfrac{10}{10^3} \times 100^2 = 10^6$ μ秒 $= 1$ 秒

アルゴリズムC $\dfrac{10^6}{10^3} \times 100 = 10^5$ μ秒 $= 0.1$ 秒

のようになり、今度は逆にA $>$ B $>$ Cとなっている。特に、アルゴリズムAの計算時間は爆発的に増えており、少々の計算速度の向上はなんの役にも立たないことがわかる。また、アルゴリズムBに関しても、データの数を $N = 100, 1000, 10000, \cdots$ のように増やしていけば、アルゴリズムCに対する計算時間の比が10倍、100倍、1000倍…と増えていく。以上のことから、N がごく小さいときはアルゴリズムC、B、Aの順に計算時間が少なくてすむが、逆に N がある程度大きければ、しかもデータの数が大きければ大きいほど、計算時間はアルゴリズムA、B、Cの順に少なくなることがわかる。したがって、N がある程度小さい時を除いてはアルゴリズムAよりアルゴリズムBの方が、またアルゴリズムBよりアルゴリズムCの方が、性能がよいといえる。

上記のアルゴリズムAまたはBをアルゴリズムCで置き換えることはプログラムの性能向上に大きく影響する。なぜなら、そもそもプログラムの実行時間のうち、大きなサイズのデータに関する計算は、小さなサイズのデータに関する計算よりも多くの部分を占める。したがって、大き

なサイズのデータの処理をする部分を大きなデータに関してより効率のよいアルゴリズムで置き換えれば、プログラムの性能向上を効率よく達成することができるのである。また、アルゴリズムの置き換えによって、これまでよりも大きなサイズのデータの処理を現在のコンピュータの性能で実行してみることも現実的になる。(たとえば 10 倍のデータ量を処理するのに、アルゴリズムBでは 100 倍、アルゴリズムAでは、約 1000 倍もの時間を要するが、アルゴリズムCは 10 倍の時間で済む。) 以上の理由から、アルゴリズムの研究においては、**ある程度大きなサイズのデータ処理の性能を比較する**ことに注目する。

では、実際どのようにしてアルゴリズム同士の性能を比較すればよいだろうか。

3.1 計算時間の見積もり

例として、2.2.3 節で紹介したリストの和を求めるプログラム

```
fun sum lst =
  case lst of
      [] => 0
    | head::tail => head + (sum tail)
  end;
```

の計算時間を見積もってみよう。関数 sum の各再帰呼び出しそのものに C_1、case 文に関しては、リストが空リストのときは C_2、そうでないときは C_3 だけ時間がかかるとしよう。(C_1, C_2, C_3は定数である。) すると、このプログラムを実行するのにかかる時間は、与えられたリストの長さが n のとき、関数 sum の呼び出しが $n+1$ 回行われ、そのうち n 回はリストが空でない。したがって全体の計算時間は、

$$C_1 \times (n+1) + C_2 + C_3 \times n = (C_1 + C_3) \times n + C_1 + C_2$$

であることがわかる。このようにしてプログラムの実行時間を見積もることはできたが、プログラムのすべての部分の実行にかかる時間を足し合わせるこの解析の仕方は、細か過ぎてプログラムどうしの実行時間の比較などを行なうのが難しい。また、そもそも実行するコンピュータの性能が上がると、定数の値も変わってしまい、意味がなくなってしまう。

ここで、アルゴリズムの性能に大きな影響を与えるのは、データのサイズが大きい時であることを思いだそう。nが十分に大きいとすると、定数$C_1 + C_2$はほとんど無視でき、計算時間はおおよそ

$$C \times n \quad (C \text{ は } C = C_1 + C_3 \text{なる定数})$$

に等しくなる。定数Cは計算機の速度の向上などによって変化するから、その実際の値はあまり意味を持たない。結局、このプログラムで表されるアルゴリズムの実行時間は

ある定数$\times n$

であるということができる。

3.2 オーダー記法による計算時間の漸近的表示

これまでの議論から、計算時間の見積もりは nが十分大きい場合の、定数倍の違いを無視したもので十分なことがわかる。このようにして見積もった計算時間を**漸近的**な計算時間という。

アルゴリズムの漸近的な計算時間を表すためには、**オーダー記法**を用いる。たとえば、上記のリストの和を求めるプログラムの計算量は、オーダー記法を用いると $O(n)$ と表すことができる。

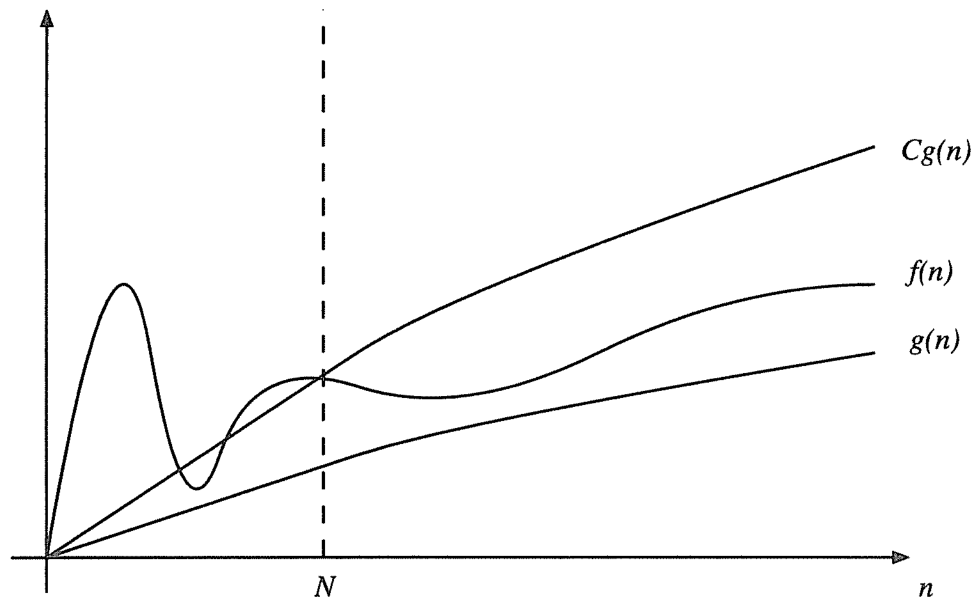

図 3: 漸近的な計算時間

これは、n が十分大きければ、上記のアルゴリズムを実行するのに、高々n の定数倍の時間しかかからないということを意味している。

オーダー記法をもう少し正確に定義しておこう。関数 $f(n)$ が、あるアルゴリズムを表したプログラムのサイズ n の入力データに対する（漸近的でない）計算時間を表しているとしよう。このとき、図 3 のように、適当な正の定数 C があって、十分に大きな n に対しては、$f(n) \leq C \times g(n)$ が常に成り立つとき、$f(n)$ は $g(n)$ のオーダーに属すと言い、$f(n) \in O(g(n))$ または $f(n) = O(g(n))$ と書く。またこのとき、アルゴリズムのオーダーは $g(n)$ である、もしくは $O(g(n))$ のアルゴリズムであると言う。

オーダー表記 $O(f(n))$ をそのオーダーに属する関数の集合とみれば、オーダーには包含関係が成り立つ。$O(f(n)) \subset O(g(n))$ のとき、$O(g(n))$ のアルゴリズムは $O(f(n))$ のアルゴリズムより計算時間がかかると読む。すなわち、オーダー記法はアルゴリズムによるの計算量を表す尺度であると言うことができる。よく使われるオーダーの表示を計算量の低い方から並べると次のようになる。

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \cdots \subset O(2^n) \subset O(n!)$$

($\log n$ は特に断らない限り、$\log_2 n$ を表すものとする。また、$O(1)$ は問題のサイズに依らず、計算が一定時間で終了するような計算量であることを示す。)

アルゴリズムのオーダーを簡単に求めるには、そのアルゴリズムにとってもっとも重要な演算が実行される回数のみを数えるのがよい。そのような演算のことをそのアルゴリズムの基本演算という。ただし、ここでいうもっとも重要な演算とは、そのアルゴリズムによる計算時間を支配する演算、すなわち全体の計算時間が、その演算の計算時間の定数倍以内でおさまるような演算のことをいう。たとえば、3.1 節のプログラムに関して言えば、足し算がそのような演算となる。なぜなら、そのプログラムにおいては、足し算が 1 回行われるときはパターンマッチも再帰関数呼び出しも丁度 1 回ずつ行われるからである。（ここで、足し算の代わりに、パターンマッチや再帰関数呼び出しの実行を基本演算としてもよいのが、基本演算にはそのようなコンピュータの動

作に関わるものではなくて、足し算や大小比較など数学的な概念を表す操作を選ぶのがふつうである。) このような方針に従えば、3.1 節のプログラムの計算量は、サイズ n の入力データに対して行われる足し算の回数は n 回であるから、そのオーダーは $O(n)$ であるというように簡単にオーダーを求めることができる。

なお、同じ問題を解くためのアルゴリズムどうしの計算時間をオーダーで比較したいときは、その問題に関するアルゴリズムすべてに共通の、計算時間を支配する演算を基本演算とし、その基本演算が実行される回数のオーダーを比較する。

3.3 最悪の場合および平均の実行時間

3.1 節のプログラム例では、プログラムの実行時間は入力データのサイズにのみ依存していた。しかし、すべてのアルゴリズムの実行時間が入力データのサイズのみに依存するとは限らない。例えば、以下のプログラム

```
fun find n key lst =
  case lst of
      [] => 0
    | head::tail => if key=head then n
                                else find (n+1) key tail
  end;

fun key_index key lst = find 1 key lst;
```

は、リスト `lst` の要素の中で `key` に一致するものがそのリストの先頭から何番目に最初に現れるかを返す関数 `key_index` を定義している。(リストが `key` と一致する要素を含んでいないときは関数 `key_index` は 0 を返す。)

実行例

```
key_index 0 [3,3,0,4,7,9];
- : int = 3
```

基本演算を比較演算 (`key=head`) として、このアルゴリズムの計算量を考えてみよう。明らかに、比較演算の行われる回数はリストの内容に依存し、`key` と一致する要素がリストの先頭から i 番目に初めて現れたとすれば、比較演算の回数は i 回となる。(一致する要素が含まれていないときは、リストの長さを n とすると、n 回となる。)

このような、入力データの内容によって計算時間の変化するアルゴリズムに関してその計算量に関する議論を行うために、ふたつの異なる尺度を考える。ひとつは、**最悪の場合**の計算量である。サイズ n の入力データのうち、もっとも計算時間のかかるものを考え、その入力データに対してかかる計算量を $W(n)$ で表す。もうひとつの尺度は、**平均**の計算量である。サイズ n のすべての入力が同じ確率で与えられると仮定して、その実行時間の期待値を $A(n)$ で表す。

関数 `key_index` の最悪の場合および平均の実行時間を解析してみよう。リストの長さは n とする。

最悪の場合の解析

もっとも計算時間がかかるのは明らかに、`key` と一致する要素がリストの最後に現れる場合か、リスト中に全く現れない場合で、その場合の比較演算の回数を考えれば、

$$W(n) = n \in O(n)$$

となる。

平均の解析

key が全くリスト中に現れない確率を p とする。すると、**key** が先頭から i 番目 $(1 \leq i \leq n)$ に初めて現れる確率 p_iは、$p_i = \frac{1-p}{n}$ としてよい。したがって、それぞれの場合で何回比較演算を行わなければならないか考えれば、

$$\begin{aligned} A(n) &= \sum_{i=1}^{n}(i \times p_i) + n \times p \\ &= \frac{n(n+1)}{2} \times \frac{1-p}{n} + np \\ &= \frac{1}{2}(n+1)(1-p) + np \\ &= \frac{1}{2}(1+p)n + \frac{1}{2}(1-p) \in O(n) \end{aligned}$$

となる。

この例では、最悪の場合と平均の計算量は一致している。しかし、これらがつねに一致するとは限らない。一般に、最悪の場合の解析を行うのは簡単であるが、平均の解析を行うには期待値を求める必要があるので、やや面倒である。本講義では、特に重要な問題を除いては最悪の場合の解析しか行わないこととする。また、特に断らない限り、アルゴリズムの計算量と言ったときは、最悪の場合の計算量のことを指すものとする。

4 探索アルゴリズム

表としてあらわされたデータの集まりの中から、あるキーワードと合致するデータを探し出すためのアルゴリズムが、**探索アルゴリズム** である。(例えば、電話帳を見て名前から電話番号を調べる等、何らかのデータベースから情報を探し出すのに使うのが典型的な使い方である。)

4.1 表をあらわすデータ構造

ここでは、表に含まれる各データは、探索のためのキーワード（キー）とそのキーに付加された情報の組で表すものとする。ただし、キーに使う値は整数や文字列など、順序がつけられるものに限るものとする。(整数は大小関係、文字列は辞書順 (lexicographical order) で順序がつけられる。) ここでは簡単のため、キーには整数を使うものとする。

このような表をコンピュータ上で表現するデータ構造について考えよう。まず、静的なデータ構造として表す場合は、たとえば次のような配列で表すとよい。(学生番号と、学生の名前の対応表と考えてもらいたい。)

```
var DB = [| {key =  3, info = "Reiko"},
            {key =  7, info = "Naoki"},
            {key =  2, info = "Atsuko"},
            {key =  9, info = "Yasuko"},
            {key =  5, info = "Susumu"},
            {key = 11, info = "Garriko"},
            {key = 14, info = "Makoto"} |];
```

この表DBにおいて、各データは**レコード**とよばれる形で表されている。レコードとは、いくつかのデータをひとまとめに扱うためのものであり、レコード中の各データは**ラベル**によって区別される。レコードは一般的には、

{ ラベル1 = 値1, ラベル2 = 値2, ..., ラベルn = 値n }

のような形をしており、レコード中のあるラベルに結びつけられた値を取り出すには

レコード . ラベル

のように書く。たとえば、

```
{key =  3, info = "Reiko"}.key;
```

```
- : int = 3
```

```
DB.[1].info;
```

```
- : string = "Naoki"
```

のようになる。

表を動的なデータ構造で表すときは、リストで表すとすると、上と同じようにレコードを使って、

```
var DB = [ {key =  3, info = "Reiko"},
           {key =  7, info = "Naoki"},
           {key =  2, info = "Atsuko"},
           {key =  9, info = "Yasuko"},
           {key =  5, info = "Susumu"},
           {key = 11, info = "Garriko"},
           {key = 14, info = "Makoto"} ];
```

のように表すことができる。

4.2 線形探索

もっとも単純素朴な探索アルゴリズムである、**線形探索**アルゴリズムを紹介する。このアルゴリズムは、単に表の先頭から与えられたキーに一致するデータがないか探していくアルゴリズムである。

配列で表された表に対する線形探索は次のようなプログラムで表される。

```
fun linear_find_key i key T size =
  if i<size
    then if T.[i].key=key
           then T.[i].info
           else linear_find_key (i+1) key T size
    else "";

fun linear_search key T = linear_find_key 0 key T (length(T));
```

linear_search は、キーと、配列で表された表を与えられて線形探索を行い、そのキーに付加された情報（そのようなキーがなければ空文字列 "" を）返す関数である。たとえば、DB を節 4.1 に示した配列であらわされた表だとすると、

```
linear_search 9 DB;
```

```
- : string = "Yasuko"
```

のようになる。

また、表がリストであらわされていたとすると、線形探索のアルゴリズムは次のように表される。

```
fun linear_search key T  =
  case T of
      [] => ""
    | head::tail => if head.key=key
                      then head.info
                      else linear_search key tail
  end;
```

DB を、節 4.1 で示した、リストであらわされた表だとすると、

```
linear_search 2 DB;
```

```
- : string = "Atsuko"
```

のようになる。

探索問題における基本演算をキーに関する比較演算とすると、上記の両アルゴリズムとも最悪の場合の計算量は $O(n)$ である。キーが偶然に表の最初の方で現れてくれると計算量は少なくてすむが、平均の計算量を求めてみると、3.3 節と同様の議論により、やはり $O(n)$ であることがわかる。もう少し効率の良い探索方法はないものだろうか？

4.3 二分探索法

探索を高速化するひとつの方法として、**二分探索法** がある。このアルゴリズムでは、表中のデータがキーの大小関係に関してあらかじめ**昇順**、すなわちだんだん大きくなるように並べられているものと仮定する。例えば、4.1 節の配列で表した表は、次のように並べられているものとする。

```
var DB = [| {key =  2, info = "Atsuko"},
            {key =  3, info = "Reiko"},
            {key =  5, info = "Susumu"},
            {key =  7, info = "Naoki"},
            {key =  9, info = "Yasuko"},
            {key = 11, info = "Garriko"},
            {key = 14, info = "Makoto"} |];
```

二分探索法の基本的なアイディアは、表のちょうど真ん中にあるデータを境にして表をサイズが半分ずつの二つの表に分割する、という操作を繰り返すことによって、探索しなければならない範囲を急速に狭めていくというものである。表中のデータがキーの大小関係に関して昇順に並んでいるという仮定から、表のちょうど真ん中のキーと、探索中のキーを比較することによって、分割された二つの表のうち、どちらかの表の中にはそのキーは含まれていないことがすぐわかる。このアイディアをもとに、配列で表された表に対して二分探索法を行うプログラムを書くと次のようになる。

```
fun b_search low high key T =         (* 表の T.[low] から T.[high-1] の部分 *)
  begin                               (* を探索                              *)
    val mid = (low+high)/2;           (* mid = 表の中央 *)

    if high-low=1                     (* 表のサイズが 1 の場合 *)
      then if key=T.[low].key
             then T.[low].info
             else ""                  (* サイズ1なので、一致しなければ探索失敗 *)
      else if key<T.[mid].key         (* keyが表の中央キーより小さい場合 *)
             then b_search low mid key T
                                      (* 表の左半分 (keyがより小さい方) を探索 *)
             else                     (* keyが表の中央のキー以上の場合 *)
                  b_search mid high key T
                                      (* 表の右半分 (keyがより大きい方) を探索 *)
  end;

fun binary_search key T = b_search 0 (length(T)) key T;
```

ここで演算記号/は割り算の商と余りのうち、商を表す。

このアルゴリズムの計算量を、基本演算を整数に関する比較演算として求めてみよう。表のサイズを n とする。

```
binary_search key T
```

を計算するには、

```
b_search 0 (length(T)) key T
```

を計算すればよい。いま、ある **b_search** の再帰関数呼び出しにおいて、$\mathbf{high} - \mathbf{low} = s$ とすると、次の再帰的関数呼び出しにおいては次の命題が成り立つ。

命題 4.1 $\mathbf{high} - \mathbf{low} \leq \dfrac{s+1}{2}$

したがって、関数 **b_search** が $\mathbf{high} - \mathbf{low} = 1$ のとき止まることを考慮すると、関数 **b_search** の再帰呼び出しは高々$O(\log n)$ であることがわかる。したがって、各再帰呼び出しで実行される比較演算の数が高々2 回であることから、二分探索アルゴリズムの計算量は $O(\log n)$ である。

4.4 二分探索木

二分探索法を、そのままリストに適用してもアルゴリズムの効率を改善することはできない。なぜなら、動的なデータ構造（すなわちリンク・データ構造）上では、配列と違って、n 個離れたデータにアクセスするには n 回リンクをたどらなければならないからである。それでは、動的なデータ構造上では線形探索以外に方法がないのかと言うと、そうではない。二分探索に似た探索を可能にするデータ構造として、**二分木** というデータ構造が知られている。

4.4.1 二分木

二分木は概念的には図 4 のような形をしたものをいう。(図を逆さにすれば、木が枝を伸ばしたように見える。) ここで図中の○は木の**節**といい、通常データを置くことができる。各節は、0 個以上 2 個以内の**部分木**を持つことができる。図中では、ある節が部分木を持つとき、その節の右下または左下に部分木を書き、線で結んで表している。この線を木の**枝**と言うことにする。図中、もっとも上にある節をその木の**根**という。部分木を持たない節は**葉**と呼ばれる。また、木の根か

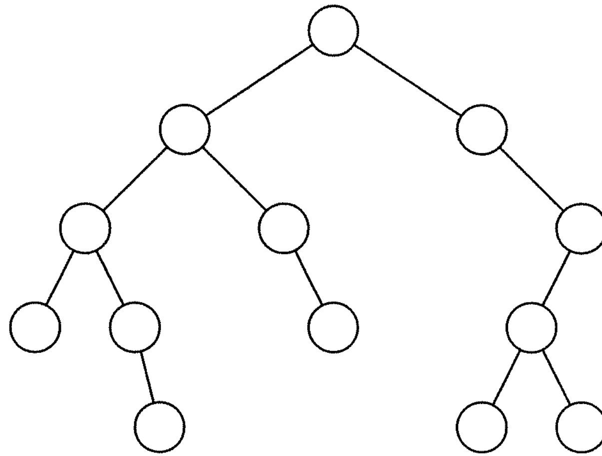

図 4: 二分木

らもっとも離れた葉に辿りつくのに枝を下向きに d 回辿らなければならないとき、その木の深さは $d+1$ であると言う。例えば、図 4 の二分木の深さは 5 である。

二分木を表すデータ構造は次のような宣言により定義できる。

```
type btree = Empty | Node of btree*btree;
```

Node(left,right) と書いた場合、left をその節の左の部分木、right を右の部分木として持つような節をあらわす。ただし、Empty は空の木をあらわし、左（または右）の部分木を持たない節は、Node(right,Empty)（またはNode(left,Empty)）であらわす。したがって、Node(Empty,Empty) は葉をあらわす。

以上のことに注意すれば、図 4 に示した二分木は次のように表現することができる。

```
Node(Node(Node(Node(Empty,Empty),
               Node(Empty,
                    Node(Empty,Empty))),
          Node(Empty,
               Node(Empty,Empty))),
     Node(Empty,
          Node(Node(Node(Empty,Empty),Node(Empty,Empty)),
               Empty)));
```

4.4.2 二分木上での探索

二分木上で探索を行うため、上記の二分木の構造を各節にデータ（キーとそれに付加された情報）を置けるように拡張する。

```
type entry = {key:int, info:string};
type btree = Empty | Node of btree*entry*btree;
```

二分木上にでたらめにデータを置くだけでは、効率よい検索は実現できない。二分木上でのデータの配置は次のような規則にしたがうものとする。

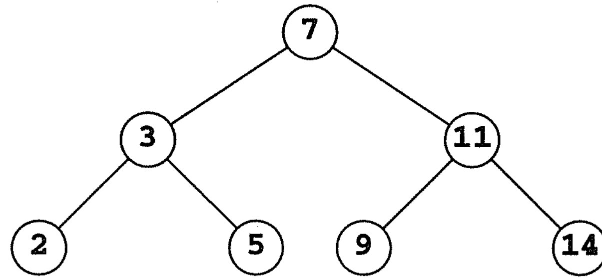

図 5: 二分探索木

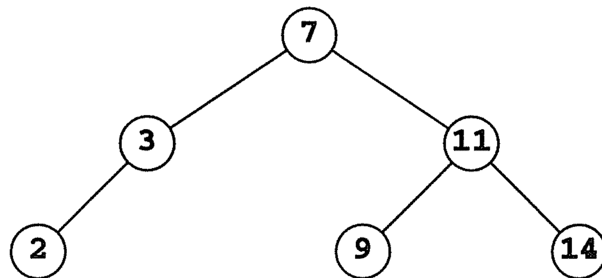

図 6: 葉を間引いた完全木

二分木上の任意の節に置かれたデータのキーの値を key とせよ。このとき、以下の二つの条件が成り立つ。

- その節が左の部分木を持つ場合、その部分木に含まれる全てのデータのキーは key より小さい。
- その節が右の部分木を持つ場合、その部分木に含まれる全てのデータのキーは key より大きい。

例えば、4.1 節でこれまで使ってきた表の例は、二分木上では次のようにあらわされる。

```
var DB =
  Node(Node(Node(Empty,{key =  2, info = "Atsuko"},Empty),
            {key =  3, info = "Reiko"},
            Node(Empty,{key =  5, info = "Susumu"},Empty)),
       {key =  7, info = "Naoki"},
       Node(Node(Empty,{key =  9, info = "Yasuko"},Empty),
            {key = 11, info = "Garriko"},
            Node(Empty,{key = 14, info = "Makoto"},Empty)));
```

これを図にすると、図5のようになる。(各節の数字は、その節に割り当てられたデータのキーを表している。) このような二分木を**二分探索木**という。

図5のように、木の中のどの節に関しても、その節の左右両方の部分木の高さが等しいとき、そのような木を**完全木**という。高さ d の完全木の節の数は $2^d - 1$ であるから、データの個数によっては完全木を作ることができない。しかし、そのような場合は、図6のような、完全木から葉を何枚か取り去った木を考えればよい。ここでは、このような木も完全木と呼ぶことにする。

命題 4.2 n 個の節からなる完全木の高さは

$$\lfloor \log n \rfloor + 1$$

である。(ただし、$\lfloor x \rfloor$ は実数 x を越えない最大の整数を表す。)

二分探索木上での探索のアルゴリズムは次のようなプログラムで表すことができる。

```
fun btree_search key tree =
  case tree of
      Empty => ""                      (* キーは探索木上にない *)
    | Node(left,entry,right) =>
        if key<entry.key               (* その節のキーより小さいとき *)
          then btree_search key left   (* 左の部分木の中を探す *)
        else if key>entry.key          (* その節のキーより大きいとき *)
          then btree_search key right  (* 右の部分木の中を探す *)
        else                           (* キーが一致 *)
          entry.info
  end;
```

このアルゴリズムの計算量は、探索木の高さを d とすると $O(d)$ である。したがって、探索木が完全木ならば、計算量は配列上での二分探索と同じ $O(\log n)$ である。

4.4.3 二分探索木へのデータの登録

二分木は動的なデータ構造として定義されているから、プログラムの実行中に、探索木に新しいデータを登録することができる。次に示すのは、元の探索木にデータ（キー key とそれに付加された情報 info）を一つ追加した探索木を作る関数である。

```
fun add_btree key info tree =
  case tree of
      Empty =>                   (* 新しいデータは木の葉になる *)
        Node(Empty,{key=key,info=info},Empty)
    | Node(left,entry,right) =>
        if key<entry.key then    (* その節のキーより小さいとき *)
          Node(add_btree key info left,entry,right)
                                 (* データを左の部分木に付け加える *)
        else                     (* その節のキーより大きいとき *)
          Node(left,entry,add_btree key info right)
                                 (* データを右の部分木に付け加える *)
  end;
```

この関数は、新しいデータをもとの二分探索木の新しい葉として付け加えた木を返す。すなわち、二分探索木の各節を、付け加えるデータのキーがその節のキーより小さければ左の部分木を、そうでなければ右の部分木を、再帰的にたどることによって、データを付け加えるのにふさわしい位置を探し出している。

二分探索木上の探索は、二分木が完全木ならば効率良く探索を行なうことができるが、そうでない場合はあまり効率が上がらない。特に、n 個の節からなる二分木は最悪の場合 n の高さを持つので、線形探索と同じ $O(n)$ の計算時間を要してしまう。また、たとえもとの二分探索木が完全木だったとしても、新しいデータの登録によって著しく効率の悪い形の探索木ができてしまう可能性がある。

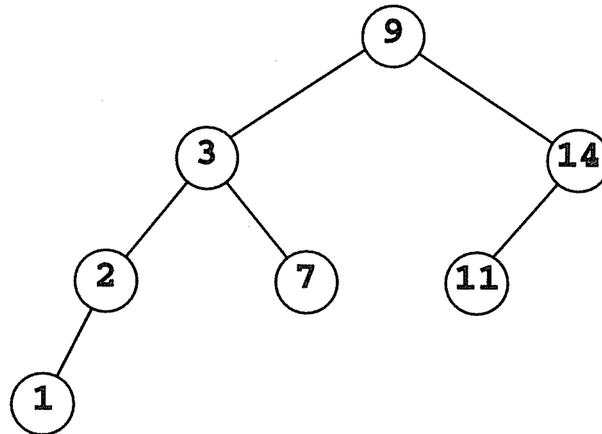

図 7: AVL 木

4.5 平衡木による探索

二分探索木上の探索は、探索木が完全木である場合は効率の良いアルゴリムとなるが、データの追加によって木の形が完全木に保たれるとは限らない。データの追加によっても探索の効率が落ちないように、木の形を保ちつつデータを追加する効率の良いアルゴリズムはないだろうか。

4.5.1 AVL 木

常に完全二分木の形を保つのは計算時間もかかり現実的でない。しかし、完全二分木とは言わないまでも、その探索木上での探索が $O(\log n)$ 程度のオーダーでおさまる程度にバランスのとれた形をしていればよいと考えられる。このような、ある程度のバランスのとれた形を持っている木のことを総称して、**平衡木** という。ここでは、平衡木の中でももっともよく知られたもののひとつである、**AVL 木**と AVL 木へのデータの追加アルゴリズムについて説明する。

AVL 木とは、どの節に関しても、その節の左右の部分木の高さの差が 1 以内であるような木のことを言う。図 7 に AVL 木の例を示す。(この AVL 木は深さ 3 の AVL 木のうち、もっとも節の数が少ないもののひとつである。)

命題 4.3 節の数が n 個の *AVL* 木の深さは高々$O(\log n)$ である。(深さ d の *AVL* 木のうち、もっとも節が少ない木の節の数を N_dとし、N_dを d に関する漸化式で表せばよい。)

4.5.2 AVL 木へのデータの追加

AVL 木上の探索には、4.4.2 節のアルゴリズムがそのまま使える。AVL 木に、AVL 木としての条件を満たすように新しいデータを付け加えるにはどのようにすればよいか考える。

ここで、以降の議論の見通しを良くするために、以下では、木を次のように表現することにしよう。

(左の部分木　根に与えられたデータ　右の部分木)

ただし、空の木は·で表すものとする。たとえば、図 7 に示した AVL 木は、この表記によれば次のように表される。

$$((((\cdot\ 1\ \cdot)\ 2\ \cdot)\ 3\ (\cdot\ 7\ \cdot))\ 9\ ((\cdot\ 11\ \cdot)\ 14\ \cdot))$$

二分探索木の定義から、上記表記において括弧と空の木・ を無視すれば、必ずキーが左側から昇順に並ぶことに注意せよ。また、木の深さは上記表記における括弧のもっとも大きな**入れ子**の数に等しい。

AVL 木にデータを追加するときのアルゴリズムの基本的な流れは以下のようである。まず、二分探索木に追加するときと同じように、データを置くのにふさわしい場所を木の枝を辿りながら探し、データをその木の新しい葉として登録する。つぎに、この新しい葉の追加によって、木の平衡が破られているかもしれないので、今まで辿ってきた道筋を逆に辿りながら、木の再構成を行う。

では、実際平衡が破られていた場合、どのようにして再構成を行って平衡を取り戻せばよいか考えていこう。いま、次のような表記

$$(\alpha\ A\ \gamma)$$

で表される木が、ある AVL 木にデータを追加することによって平衡の破れた最小の木（木全体としては平衡が破れているが、部分木αとγは AVL 木の条件を満たしている）とする。左右は対称に考えられるから、ここでは左の部分木に対して追加を行った結果、平衡が破られたとしよう。$d(\alpha)$と $d(\gamma)$ がそれぞれ、左の部分木の深さと右の部分木の深さを表すとする。AVL 木の条件から、

$$d(\alpha) - d(\gamma) = 2$$

でなければならない。したがって、左の部分木は少なくとも 1 の深さを持つので、$(\alpha\ A\ \gamma)$ は

$$((\alpha'\ B\ \beta)\ A\ \gamma)$$

と表され、α'とβの深さに関して、次の二つの場合が考えられる。

1. $d(\alpha') = d(\gamma) + 1$ かつ $d(\beta) = d(\gamma)$
2. $d(\alpha') = d(\gamma)$ かつ $d(\beta) = d(\gamma) + 1$

まず 1 の場合を考えてみよう。このときは、もとの木 $((\alpha'\ B\ \beta)\ A\ \gamma)$ を

$$(\alpha'\ B\ (\beta\ A\ \gamma))$$

のように再構成する。ここで行ったのは括弧の付け換えだけであるから、再構成された木は明らかに二分探索木の条件を満たす。また、AVL 木としての条件も、各部分木の高さに関する条件により満たされる。このような木の再構成の仕方を **1 重回転** という。

つぎに 2 の場合を考えてみよう。この場合は、木βに関する深さの条件から、木βは少なくとも 1 の高さを持つので、木全体は

$$((\alpha'\ B\ (\beta_1\ C\ \beta_2))\ A\ \gamma)$$

のように表され、木β_1, β_2の深さに関して、つぎのいづれかが成り立つ。

- $d(\beta_1) = d(\gamma)$ かつ $d(\beta_2) = d(\gamma)$
- $d(\beta_1) = d(\gamma) - 1$ かつ $d(\beta_2) = d(\gamma)$
- $d(\beta_1) = d(\gamma)$ かつ $d(\beta_2) = d(\gamma) - 1$

したがって、木 $((\alpha'\ B\ (\beta_1\ C\ \beta_2))\ A\ \gamma)$ を

$$((\alpha'\ B\ \beta_1)\ C\ (\beta_2\ A\ \gamma))$$

のように再構成すると、これも括弧の付け方を変えただけなので、二分探索木の条件を明らかに満たし、また AVL 木としての条件も各部分木の高さに関する条件により満たされる。このような木の再構成の仕方を **2 重回転** という。

```
fun add_AVL key info tree =
  case tree of
      Empty =>                        (* 新しいデータは木の葉になる *)
        Node(Empty,{key=key,info=info,depth=1},Empty)
    | Node(left,entry,right) =>
        if key<entry.key then         (* その節のキーより小さいとき *)
          begin                       (* データを左の部分木に付け加える *)
            val ltree = add_AVL key info left;
            val rtree = right;
            val ldepth = depth ltree;
            val rdepth = depth rtree;

            if (ldepth-rdepth) > 1    (* 平衡が破れたとき *)
              then                    (* 左の部分木の方が深いとして再構成 *)
                rebalance_left (Node(ltree,entry,rtree))
              else                    (* 平衡しているときは深さのみを更新 *)
                Node(ltree,{key=entry.key,info=entry.info,
                            depth=(max ldepth rdepth)+1},rtree)
          end
        else                          (* その節のキーより大きいとき *)
          begin
            val ltree = left;         (* データを右の部分木に付け加える *)
            val rtree = add_AVL key info right;
            val ldepth = depth ltree;
            val rdepth = depth rtree;

            if (rdepth-ldepth) > 1    (* 平衡が破れたとき *)
              then                    (* 右の部分木の方が深いとして再構成 *)
                rebalance_right (Node(ltree,entry,rtree))
              else                    (* 平衡しているときは深さのみを更新 *)
                Node(ltree,{key=entry.key,info=entry.info,
                            depth=(max ldepth rdepth)+1},rtree)
          end
  end;
```

図 8: AVL 木へのデータ追加プログラム

4.5.3 AVL 木を再構成するプログラム

これまでの議論をもとに AVL 木にデータを追加し、平衡が破られていれば木を再構成するプログラムを書いてみよう。平衡が破られているかどうかを確認するためには、左右の部分木の深さを求める必要があるが、木の深さが必要になる度にそれをいちいち計算していると計算量が増大してしまう。そこで、ここでは AVL 木を表すデータ構造を、各節にその節の高さに関する情報を含むよう、次のように定義する。

```
type AVLentry = {key:int, info:string, depth:int};
type AVLtree = Empty | Node of AVLtree * AVLentry * AVLtree;
```

このように定義した場合、木の深さを求める関数は次のように定義できる。

```
fun depth tree =
  case tree of
      Empty => 0
    | Node(left,entry,right) => entry.depth
  end;
```

この関数は、木を再帰的に辿って計算する必要がないので、定数時間でその木の深さを返す。

```
fun rebalance_left tree =
  begin
    val Node(Node(alpha,B,beta),A,gamma) = tree;

    if (depth alpha)>(depth beta) then  (* 1重回転 *)
      Node(alpha,
           {key=B.key,info=B.info,depth=(depth alpha)+1},
           Node(beta,
                {key=A.key,info=A.info,depth=(depth beta)+1},
                gamma))
    else                                (* 2重回転 *)
      begin
        val Node(beta1,C,beta2) = beta;

        Node(Node(alpha,
                  {key=B.key,info=B.info,depth=(depth alpha)+1},
                  beta1),
             {key=C.key,info=C.info,depth=(depth alpha)+2},
             Node(beta2,
                  {key=A.key,info=A.info,depth=(depth gamma)+1},
                  gamma))
      end
  end;
```

図 9: 木の再構成プログラム

AVL 木にデータをひとつ登録し、必要があれば木の再構成を行なう関数 add_AVL を図 8 に示す。この関数は、二分探索木に対するデータの追加と同じようにデータを追加したあと、その追加によって平衡が破られた場合、左の部分木と右の部分木のどちらの深さの方が大きいかによって、関数 rebalance_left または rebalance_right を木を再構成するために呼び出す。左の部分木の深さの方が大きい場合に呼ばれる関数 rebalance_left は 4.5.2 節での議論から、図 9 のように書くことができる。同様にして、右の部分木の深さの方が大きい場合に呼ばれる関数 rebalance_right も書くことができる。

命題 4.4 関数 add_AVL による *AVL* 木へのデータの追加は、もとの *AVL* 木に含まれるデータの数が n のとき、$O(\log n)$ の計算量である。

5 整列アルゴリズム

整列（ソート）アルゴリズムとは、データの列をその大小関係によって並べ換えるためのアルゴリズムのことを指す。整列アルゴリズムは、データの整理などのために使われる基本的アルゴリズムである。

なお、以下では話を簡単にするため、整列アルゴリズムは整数の列を昇順に整列するもののみを考える。

5.1 配列上での整列

データの列が配列として与えられた場合の整列アルゴリズムの基本的な操作は、配列中の 2 つのデータを比較してその大小関係が逆転していれば、それらのデータを入れ換えるというものである。配列では、どんなに離れた要素でもインデックスによって定数時間でアクセスできるので、

このデータを入れ換えるという操作も常に定数時間でおこなうことができる。配列 a のインデックス i と j で指し示されるデータを入れ換える関数 swap は

```
fun swap a i j =
  begin
    val tmp = a.[i];
    a.[i] <- a.[j];
    a.[j] <- tmp
  end;
```

のように定義できる。

5.1.1 単純な整列アルゴリズム

まず手始めに、素朴な発想で整列アルゴリズムを作り、その計算量を検討してみよう。整列アルゴリズムの基本演算として採用すべき候補には、比較演算とデータの入れ換え操作の二つがあるが、データの入れ換えをする前には必ず比較演算を行なう必要があることから、以下では特に断らない限り、比較演算を基本演算とし、整列されるデータの長さは n として計算量を算出する。

```
fun selection_sort a =
  begin
    val size = length(a);        (* size = 配列のサイズ *)

    for i=0 to size-1 do         (* 小さいデータから左に詰めていく *)
      begin
        var min_index=i;

        for j=i to size-1 do     (* 残ったデータの中で最小のものを探す *)
          if a.[j]<a.[min_index] then min_index <- j;

        swap a i min_index       (* 最小のものを詰める *)
      end;
    a
  end;
```

図 10: 選択ソート

図10に示すのは、**選択ソート**とよばれるアルゴリズムである。このアルゴリズムのアイディアは、整列を行なうには、配列の中で最小のデータを最初の要素に、2番目に小さいデータを次の要素に…といったふうに順に最小値を探していけばよいというものである。

このアルゴリズムの計算量を求めてみよう。このアルゴリズムは入れ子になったふたつのループからなっている。外側のループは $\mathtt{i}=0$ から $\mathtt{i}=n-1$ まで繰り返し実行され、内側のループは $\mathtt{j}=\mathtt{i}$ から $\mathtt{j}=n-1$ まで繰り返し実行される。この内側のループが1回実行されるごとに比較が1回行なわれるので、結局このアルゴリズムの計算量は、

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

であることがわかる。なお、このアルゴリズムの計算量は、入力されるデータのサイズのみに依存し、データの内容には依存しない。

選択ソートのように、最大値（または最小値）を見つけるという操作をアルゴリズムの核としこの操作をできるだけ高速化する、という方針で突き進むことによって、アルゴリズムの性能をあげ

ることも可能である。実際、そのようなアルゴリズムとしてヒープソートが知られている。ヒープソートの計算量は最悪、平均ともに $O(n \log n)$ であるが、アルゴリズムに含まれる操作自体がかなり複雑なものになってしまうため、期待したほどの効率化は得られない。本当に効率の良いアルゴリズムを得るには、方針を変えなければならない。

5.2 リスト上での整列

これからは、大小関係にしたがって配列要素の入れ替えを行うなどの局所的な見方ではなくて、もっと大局的な見方でアルゴリズムの設計を行う。やや概念的な説明で言うと、あるデータを整列するのに、それをいくつかのより小さな部分問題に分割して整列を行い、その結果を後でまとめることによって全体の整列を得るという方針を採ることにする。このように、大きな問題をより小さな問題に分割して解くことを、一般に**分割統治法**という。

実は、このような分割統治法によるアルゴリズムは配列上よりもリスト上でプログラムを書いた方がより自然にアルゴリズムを表現することができる。これは、リストの持つ再帰的構造が、分割統治法が持っている再帰的な計算構造とうまく調和しているからである。

5.2.1 クイックソート

クイックソートは、最悪の場合の計算量こそ $O(n^2)$ であるが、平均の場合の計算量は $O(n \log n)$ となる高速な整列アルゴリズムである。ヒープソートは最悪の場合でも計算量が $O(n \log n)$ となるが、クイックソートの方がアルゴリズムが単純なので、実際の実行時間を考えると平均的には有利なので、実際のプログラムで使用されるのはこちらの方が多い。

図11にクイックソートのプログラムを示す。クイックソートのアイディアはごく明解である。ある整数のリストを整列するには、まずリスト中の適当な値をえらんで基準値とし、基準値以下のデータと、基準値より大きいデータとに分割する。すなわち、リストは基準値より小さい部分列と大きい部分列とに分割される。あとはそれぞれの部分列を整列して、整列した二つのリストをつなげればよい。部分列を整列するにはさらに部分列を分割していけばよい。分割を繰り返して、部分列が空になったとき、空列は明らかに整列された列であるから、ここで部分列の分割を終わることができる。

図11のクイックソートのプログラム中では、リストを分割するのに、関数 `divide` を使っている。この関数は与えられたリストを、基準値（`pivot` であらわされる）以下のデータのリスト（`less`）と基準値より大きいリスト（`greater`）に分割する。分割したリストは (`less`, `greater`) の形、すなわちふたつのリストの組として返される。クイックソートのプログラムの本体である関数 **quick_sort** は、整列すべきリストの先頭の値を基準値として、リストを二つに分割する。そして、それぞれの分割されたリストに対して自分自身を呼び出して整列を行い、整列されたリストをつなげる。リストをつなげるには、関数 **append** を使っている。関数 **append** は次のように定義すればよい。

```
fun append lst1 lst2 =
  case lst1 of
      [] => lst2
    | head::tail => head::(append tail lst2)
  end;
```

クイックソートの計算量を求めよう。クイックソートの主な手続きはリストの分割であるが、その他にも関数 **append** によるリストのつなぎ合わせの操作がある。配列の場合と違って、このようなリストの操作にかかる時間もちゃんと考慮にいれるべきである。関数 **append** によるリストのつ

```
fun divide pivot lst =            (* リスト lst を基準値 pivot で分割 *)
  case lst of
      [] => ([],[])               (* 空リストは二つの空リストに分割 *)
    | head::tail =>
        begin                     (* 先頭を除いたリストを分割 *)
          val (less,greater) = divide pivot tail;

          if (head>pivot)         (* 先頭のデータを基準値によって *)
                                  (* どちらかの部分リストに加える *)
            then (less,head::greater)
            else (head::less,greater)
        end
  end;

fun quick_sort lst =
  case lst of
      [] => []                    (* 空リストは既に整列されている *)
    | head::tail =>
        begin                     (* 先頭を基準値として残りのリストを分割 *)
          val (less,greater) = divide head tail;
                                  (* 二つの部分リストを整列し、つなぎ合わせる *)
          append (quick_sort less) (head::(quick_sort greater))
        end
  end;
```

図 11: クイックソート

なぎ合わせにかかる時間は明らかに、プログラム中の引数 lst1 に対応するリストの長さに比例する。しかし、クイックソートのプログラム自体を見るとわかるように、append は関数 divide によって分割されたリストのつなぎあわせを行っているのだから、リストのつなぎ合わせに要する時間は、高々divide の実行によって行われる比較演算にかかる時間の定数倍程度であると言える。したがって、ここでは関数 divide が行う比較演算を基本演算として計算量を求める。

それでは、クイックソートの最悪の場合の計算時間を考えてみよう。クイックソートにおけるデータの分割が常に極端に片寄っていた場合、すなわち、分割によって得られる二つの部分列のうち一方が空である場合がもっとも計算時間を要することはすぐにわかる。したがって、このような場合、長さ m の部分列の分割に要する計算時間が $O(m)$ であることから、全体の計算時間が $O(n^2)$ であることがわかる。

逆に理想的に分割が行われた場合のクイックソートの計算時間について概算してみよう。理想的な分割とは、部分列がちょうど等しいサイズに分割されるときである。分割される前の配列の長さを L、分割された後の部分列の長さを L'とすると、

$$L = 2 \times L' + 1$$

これを変形して、

$$L + 1 = 2 \times (L' + 1)$$

であるから、入力される整数列の長さは $n = 2^m - 1$ とあらわされると考えてよい。整数列の分割を行なうごとに、さらに分割しなければならない部分列の数が倍になることに注意すると、分割に必要な比較演算の回数の総計は、

$$\sum_{i=1}^{m} 2^{m-i}(2^i-2) = 2^m\sum_{i=1}^{m}1 - 2\sum_{i=1}^{m}2^{m-i} = m2^m - 2\sum_{i=0}^{m-1}2^i$$
$$= m2^m - 2(2^m-1) = (n+1)\log(n+1) - 2n = O(n\log n)$$

となる。したがって、最善の場合、クイックソートの計算量は $O(n\log n)$ であることがわかる。

クイックソートの最悪の場合の計算量は $O(n^2)$ だが、平均ではクイックソートはそれよりずっと効率の良いアルゴリズムである。(実は、平均の計算量は最前の場合と同じ計算量 $O(n\log n)$ である。)

平均の計算量を求めるため、長さ n の整数列の分割に必要な比較演算の回数を Q_n と書こう。すると、一般的には次のような等式が成り立つ。

$$Q_n = n-1+Q_a+Q_b$$

ここで、Q_a と Q_b は長さ n の列を分割した後、新しくできた二つの列をさらに分割するのに必要な比較演算の回数なので、$a+b+1=n$ の関係が成り立つ。

整数列の分割が等確率でおこったとすると、必要な回数の平均の漸近式は次の通りである。

$$Q_n = n-1+\frac{1}{n}\sum_{i=0}^{n-1}(Q_i+Q_{n-1-i}) = n-1+\frac{2}{n}\sum_{i=0}^{n-1}Q_i$$

さらに、$Q_0=0$、$Q_1=0$ となっている。

上の漸近式を解こう。まず $S_n = Q_n - 2$ と置く。

$$S_n = n-3+\frac{2}{n}\sum_{i=0}^{n-1}(S_i+2) = n+1+\frac{2}{n}\sum_{i=0}^{n-1}S_i$$

$$nS_n = n(n+1)+2\sum_{i=0}^{n-1}S_i$$

同じ式の n を $n-1$ で取ると、

$$(n-1)S_{n-1} = (n-1)n+2\sum_{i=0}^{n-2}S_i$$

これらを辺々引くと、

$$nS_n-(n-1)S_{n-1} = 2n+2S_{n-1}\ nS_n = (n+1)S_{n-1}+2n$$

$n(n+1)$ で割ると、

$$\frac{S_n}{n+1} = \frac{S_{n-1}}{n}+\frac{2}{n+1} = \frac{S_{k-1}}{k}+\sum_{i=k}^{n}\frac{2}{i+1} = S_0+\sum_{i=1}^{n}\frac{2}{i+1}$$

これを調和級数 $H_n = 1+\frac{1}{2}+\ldots+\frac{1}{n}$ を元に書きなおすと、

$$\frac{S_n}{n+1} = -2+2(H_{n+1}-1) = 2(H_{n+1}-2)$$

$\lim_{n\to\infty}\frac{H_n}{\log n} = 1$ であることが知られているので、

$$Q_n = 2(n+1)(H_{n+1}-2) = O(n\log n)$$

となる。よって、クイックソートの平均の計算量は $O(n\log n)$ である。

このように、クイックソートは、平均の計算量がヒープソートの最悪の計算量と同じであるから、平均的には十分高速なアルゴリズムである。しかもアルゴリズムが単純なので、実際はたいていヒープソートより速く実行できる。このため、クイックソートは実際のプログラムでもっともよく使われるアルゴリズムの一つとなっている。

ただし、クイックソートによって分割が効率よく行われるかどうかは、与えられたリストにデータがどのように並んでいるかによることに注意しなければならない。ここであげたプログラムのように、先頭のデータを常に基準値として選ぶと、リストが初めから整列されていた場合が最悪の計算量を必要とする場合になる。これはすなわち、リストがもともと（完全にではなくともおおよそ）整列されているときは、整列にかなり時間がかかってしまうことを示している。したがって、リスト上でのデータの整列に関しては、データがもともとある程度整列されていることがわかっている場合は次に上げるような他のアルゴリズムを使った方が良い。

5.2.2　マージソート

マージソートは、クイックソートと同じ分割統治法に基づく整列アルゴリズムである。しかも、その最悪の場合の計算量はヒープソートと同じ $O(n\log n)$ である。

マージソートのアルゴリズムを、図12に概念的に示す。マージソートは、クイックソートとは逆に、初めからデータの列をひとつずつのデータに分解して、それらの列を組み合わせていくことで整列を行う。最初、ひとつずつのデータはそれ自身長さ1の（したがって明らかに整列された）列をあらわしている。次に、この整列された列を二つずつ組み合わせて（マージという）、新しい整列された列を作っていく。この二つずつマージする操作を繰り返せば、最終的には1つの整列された列を得ることができる。

図13にマージソートのプログラムを示す。マージソートでもっとも重要なのは、2つのすでに整列されたリストをマージして整列したリストを得る操作である。この操作は関数 `merge` としてあらわされている。マージのアルゴリズムは簡単で、2つのリストの先頭の要素を比較して、小さい方だけを、新しいリストに先に出現するよう付け加える、という操作をどちらかのリストが空になるまで続ければよい。あとは、この関数 `merge` を使ってリストを2つずつマージしていけばよいのだが、その前に、与えられたリストの要素をひとつずつ分解して長さ1のリストの列を作らなければならない。このようなことを行うのが、関数 `init_merge` である。`init_merge` はリ

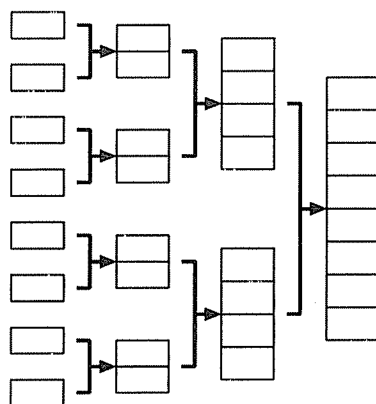

図 12: マージソートの概念

```
fun merge lst1 lst2 =            (* 整列された二つのリストをマージ *)
  case (lst1,lst2) of
      (lst1,[]) => lst1          (* 一方が空ならもう一方を返す *)
    | ([],lst2) => lst2
    | (hd1::tl1,hd2::tl2) =>     (* 二つのリストの先頭を比べて *)
        if hd1<hd2               (* 小さい方を先にする         *)
          then hd1::(merge tl1 (hd2::tl2))
          else hd2::(merge (hd1::tl1) tl2)
  end;

fun init_merge lst =             (* リストをその各データ1つからなる *)
  case lst of                    (* リストのリストに変換する         *)
      [] => []
    | head::tail => [head]::(init_merge tail)
  end;

fun combine lsts =               (* 先頭から2つずつマージする *)
  case lsts of
      [] => []                   (* リストの要素が1個以下の時は *)
    | [lst] => [lst]             (* マージしない                 *)
    | lst1::lst2::tail => (merge lst1 lst2)::(combine tail)
  end;

fun msort lsts =                 (* マージすべきリストがひとつになるまで *)
  case lsts of                   (* マージを続ける                       *)
      [lst] => lst
    | lst::tail => msort (combine lsts)
  end;

fun merge_sort lst = msort (init_merge lst);
```

図 13: マージソート

スト $[e_1, e_2, \ldots, e_n]$ を受け取ると、それをもとにリストのリスト $[[e_1], [e_2], \ldots, [e_n]]$ を作る関数である。このようにこのプログラム中では、整数列の列をリストのリストとしてあらわしている。

マージソートの計算量を算出してみよう。ここでは簡単のため、入力されるリストの長さ n は、$n = 2^m$ であらわされるとしよう。まず関数 merge の計算量について考えてみよう。明らかに、関数 merge が行う比較演算の数は高々(マージする2つのリストの長さの和) − 1 回である。また、マージソートのアルゴリズムでは、長さ 2^i のリストを、それぞれ 2^{m-i-1} 組マージするから、マージソートの最悪の場合の計算量は、

$$\sum_{i=0}^{m-1}(2^{i+1}-1)2^{m-i-1} = \sum_{i=0}^{m-1}(2^m - 2^{m-i-1}) = m2^m - 2^{m-1}\frac{1-2^m}{1-2}$$

$$= m2^m + 2^{m-1} - 2^{2m-1} = O(n \log n)$$

より、ヒープソートと同じ $O(n \log n)$ であることがわかる。