卒業論文

最長部分列問題の解法に関する研究

指導教官　胡 振江 講師

2000年2月14日

氏名　林 俊介

目次

1 序論 **1**

1.1 研究の目的と背景 1

1.2 論文の構成 1

2 最長部分列問題 **3**

2.1 用語 3

2.2 演算子と関数の表記 4

2.2.1 列の基本操作 5

2.2.2 列集合を返す関数 5

2.2.3 その他の関数 6

2.2.4 アルゴリズムを操作する宣言 6

2.3 最長部分列問題を求める関数 7

3 述語の閉属性に着目した解法 **8**

3.1 述語の閉属性 8

3.2 述語が *prefix-closed* である場合の解法 8

3.2.1 述語が *overlap-closed* でもある場合 9

3.2.2 Windowing 手法 10

3.3 列分割を適用した解法 11

3.3.1 列分割の定義 11

3.3.2 欲張り法 11

3.3.3 欲張り法と合わせて用いる命題 13

3.3.4 最長 p_R 部分列問題の解法 14

3.4 $p(x)$ の計算時間 15

3.5 閉属性を持つ述語の例 15

4 最左最右関係問題 **16**

4.1 関係 R の性質 16

4.2 線形解法 17

4.2.1 付随列の定義 17

4.2.2 不変式 18

4.2.3 解法 18

4.3 他の問題への応用 19

4.3.1 部分列和の正負に関する問題 21

4.3.2 部分列平均に関する問題 21

4.3.3 部分列和と定数の比較に関する問題 21

4.4 関数化への拡張 22
4.4.1 付随列の定義 22
4.4.2 不変式 22
4.4.3 解法 23
4.5 応用例 25
4.5.1 最左要素と最右要素の積に関する問題 25
4.5.2 部分列和と定数の比較に関する問題 26

5 計算機上でのプログラミング **29**
5.1 Haskell の特徴 29
5.2 最左最右関係問題のプログラム 29
5.3 改善法 30
5.3.1 対リスト構造を用いた列の表記法 30
5.3.2 対リスト構造に用いる関数の定義 30
5.4 結果 31

6 結論 **32**

A 最左最右関係問題の Haskell プログラム (改善前) **33**

B 対リスト構造に用いる関数 **35**

C 最左最右関係問題の Haskell プログラム (改善後) **37**

D 部分列和と定数の比較に関する問題の Haskell プログラム **39**

謝辞 **41**

参考文献 **42**

第 1 章

序論

1.1 研究の目的と背景

例えば, ラーメン屋の売上を例に挙げてみよう. 勿論, 一日中どの時間帯も一定してお客さんが入る訳ではない. お昼時や夕方はたくさんお客さんが来るだろうし, 逆に夜中や早朝はお客さんは殆んどいない. そこで, ラーメン屋の親父としては, たくさんお客さんが来る時間帯だけ商売をして, それ以外の時間帯は人件費等の節約上店を閉めることにしたいと考えるのは当然であろう. しかし, 時間平均にしてある一定以上のお客さんが来る時間帯はなるべく長くお店を開けていたい. このようなときどうすればよいだろうか.

まずはこの問題を数学的に置き換えてみる. 例えば, お客さんの来る数を 10 分ごとに平均して離散化したデータを時系列データとして並べてみよう. 一日 24 時間だから $24 \times 6 = 144$ 個のデータが数列として出てくるはずだ. そして, この数列の中の連続する部分列で, 平均 C 以上のものの中から最も要素数の多いものを選ぶというのが, 数学的なモデルである.

ところで, この問題は NP 問題（指数オーダー）ではない. 何故なら, 部分列に連続しているという条件が与えられているからだ. n 個の要素を含む列の中に, 連続する部分列は ${}_nC_2 + {}_nC_1 + {}_nC_0$ 個, すなわち約 $n^2/2$ 個程度であるから, 一つの部分列の平均を求めるのにその要素数と同じ数の演算が必要としても, 全体としては $O(n^3)$ で済んでしまう.

昨今のコンピューターの進歩の具合は凄まじいものがある. よって, $O(n^3)$ 程度では大した計算量ではないと思われるかもしれない. しかし, 膨大なデータを扱う状況においては, $O(n)$ と $O(n^3)$ では計算にかかる時間大きな差があるのは否めないし, また, 電話回線のような時間と共に流れていくようなデータに対してリアルタイムにその時どきの最適解を見つけていくというような観点から見ても, $O(n)$ で問題を解くアルゴリズムというのは非常に重宝すると言えよう.

よって , この論文では, ラーメン屋の売上の例でも挙げたように, ある一定の条件を満たす連続する部分列の中で最も要素の多いものを求める問題, すなわち, 最長部分列問題にテーマを絞り, そのある一定の条件というのが, どのようなときならば, $O(n)$ にできるか, そして, それはどのようなアルゴリズムで実現できるかと言うことについて研究の成果を述べていきたい.

1.2 論文の構成

本論文は, 最長部分列問題ということで, 普通の数学で用いるような演算とは違った, 数列に関する演算を行なう. よって, 第 2 章では, まずその数列に関する演算の定義, および用語の解説といった, この論文を読むにあたって必要な予備知識について述べて, その予備知識を元に最長部分列問題の本質に迫りたいと思う. 次の第 3 章では, 述語（語彙は 2.1 節で説明）が閉属性というある性質を満たす場合のアルゴリズムについて説明する. なお, この第 3 章で述べていることは, 論文 [1] をほぼ参考にしてある. 第 4 章では, 閉属性とは違った性質を持つ最左最右関係問題をテーマに論じてある. 最左最右関

係問題を $O(n)$ で解くアルゴリズムに関しては, 論文 [1] を参考にしているが, その問題の応用, 及び関数化の可能性については, 自分なりの研究成果が得られた．また, この問題における関係 R の条件についても, 論文 [1] では詳しく述べられていないため, その条件の必要性と詳しい証明を付しておいた. 第 5 章は, アルゴリズムを実際に関数型言語 Haskell を用いて, コンピューター上で実行し, 机上のアルゴリズムと実際の計算におけるギャップなどをどう補っていくかといった内容になっている．なお, この章では, Haskell の構造及びリストの応用の仕方など, [2], [4] を参考にさせて頂いた.

第 2 章

最長部分列問題

この章では, 本論文を読むにあたって必要な予備知識を 2.1 節, 2.2 節に示し, それらを用いて 2.3 節で最長部分列問題を表記すると共に, 問題解決の基本的な式を示す.

2.1 用語

この論文で用いられる用語についての説明を以下に行う.

- 列と要素 (string and element)

  列とは簡単に言うと, 同じ種類の要素を決まった順番に一次元的に並べたものである. 基本的に要素は, 文字, 数字, 記号など何でも良いのだが, 本論文中では数字を要素とする列, すなわち数列に限って議論する. よって, 以降は列と言うと, その要素は数字であると思って頂いて差し支えない.

  なお, 列を表すのに, 例えば

  $$x = [3, 5, 7, 6, 3, 8, 5],\ \ y = [5, 2, 6]$$

  のように, 要素を [ ] で囲んで表す. ここで, x, y は列であり, 大括弧内の $3, 5, 7, ...$ はその列の要素である. また, 要素を持たない列として空列 [ ] を定義しておく.

  なお, この論文では特別な注意が無い限り, 以降 $x, y, z, ...$ は列, $a, b, c, ...$ は要素, $X, Y, Z, ...$ は列の集合, というふうに表すものとする.

- 列の長さ

  列の長さとはすなわちその列の要素の数を意味する. また, 列 x の長さは $\mathit{length}\ x$ で表す. 例を挙げると,

  $$\mathit{length}\ [5, 3] = 2,\ \ \mathit{length}\ [3, 7, 5, 1] = 4$$

  なお, 空列に対しては,

  $$\mathit{length}\ [\ ] = 0$$

  である.

- 部分列 (segment)

  部分列とは, 元の列の中から連続する一部分をとりだしたものである. また, 空列及び元の列も部分列に含まれる. 例えば,

$x = [1,2,3,4,5,6,7]$ のとき,

$[\,], [3], [2,3,4], [4,5,6,7], [1,2,3,4,5,6,7]$: x の部分列
$[2,2], [3,5,6], [5,4,3,2]$: x の部分列ではない

- 述語 (predicate)

  述語とは, 列に対して或る演算を行い, その結果を真偽値で返すものであり, 本論文中ではしばしば p で表される. 例えば,

  $p(x) \equiv x$ の長さが偶数　のとき,

  $p[2,5,3] = \mathit{False},\ p[6,3,7,6] = \mathit{True}$

- 最長 p 部分列 (longest p segment)

  最長 p 部分列とは, 元の列 z の部分列 x の中で, $p(x) = \mathit{True}$ となるようなもののうち, 長さが最大のもののことである. 例えば,

  $p(x) \equiv x$ の最左要素と最右要素が等しい　のとき,

  $[2,5,3,7,5,9,3,1,5,6]$ の最長 p 部分列は, $[5,3,7,5,9,3,1,5]$ である.

- 線形アルゴリズム (linear algorithm)

  線形アルゴリズムとは, 列の長さに対して $O(n)$ の時間で解けるアルゴリズムのことである.

- on-line アルゴリズム

  on-line アルゴリズムとは, 例えば列を x まで読んで, その時点での最適解を出した時, $x +\!\!+ [a]$ の最適解は, x までの情報を用いて求めることが出来るというものである. すなわち, データをある一定方向に読んでいく場合, 先読みをしなくても, 最適解が出せるというものである.

- real-time アルゴリズム

  real-time アルゴリズムとは, on-line アルゴリズムの一種で, on-line アルゴリズムにおいて, x までの最適解から $x +\!\!+ [a]$ の最適解を求める際に, それにかかる時間が x の長さに依存しないというものである. なお, real-time アルゴリズムは必ず線形アルゴリズムでもある.

2.2 演算子と関数の表記

この論文ではアルゴリズムを表すのに, 関数型言語である Haskell を意識した形の表記法を用いる. それを以下に示す. （但し, 以下に書かれた表記法は全て Haskell で使えるという訳ではない.）

2.2.1 列の基本操作

- : (列に対する要素の添加)

  この演算子は列に要素を添加するのに用いられる. 例えば,

  $x = [4, 2, 8],\ a = 7$ のとき, $a : x = [7, 4, 2, 8]$

- $+\!\!+$ (列と列の連結)

  この演算子は列と列を連結するのに用いられる. 例えば,

  $x = [1, 2, 3],\ y = [4, 5, 6]$ のとき, $x +\!\!+ y = [1, 2, 3, 4, 5, 6]$

- *leftmost* と *rightmost* (列の最左要素, 最右要素)

  この関数は, 与えられた列の最左要素, 及び最右要素を返す. 例えば,

  $\mathit{leftmost}\ [a_1, a_2, ..., a_n] = a_1$

  $\mathit{rightmost}\ [a_1, a_2, ..., a_n] = a_n$

- *tail* と *init*

  この関数は, 列から最左要素を除いたもの, 及び列から最右要素を除いたものを返す.
  例えば,

  $\mathit{tail}\ [a_1, a_2, ..., a_n] = [a_2, a_3, ..., a_n]$

  $\mathit{init}\ [a_1, a_2, ..., a_n] = [a_1, a_2, ..., a_{n-1}]$

2.2.2 列集合を返す関数

- *tails* と *inits*

  この関数は, 与えられた列から, 以下のような列集合を返す.

  $\mathit{tails}\ [a_1, a_2, ..., a_n] = \{[a_1, a_2, ..., a_n], [a_2, a_3, ..., a_n], ..., [a_{n-1}, a_n], [a_n], [\ ]\},$

  $\mathit{inits}\ [a_1, a_2, ..., a_n] = \{[\ ], [a_1], [a_1, a_2], ..., [a_1, a_2, ..., a_{n-1}], [a_1, a_2, ..., a_n]\}.$

- *segs* (部分列集合)

  この関数は, 与えられた列の全ての部分列を列集合にして返す. 例えば,

  segs [1,2,3,4]={[ ],[1],[2],[3],[4],[1,2],[2,3],[3,4], [1,2,3],[2,3,4],[1,2,3,4]}

  また, 関数 *segs* は以下のように帰納的に定義される.

  $\mathit{segs}\ [\ ] = \{[\ ]\}$

  $\mathit{segs}\ (x +\!\!+ [a]) = \mathit{segs}\ x \cup \mathit{tails}\ (x +\!\!+ [a])$

2.2.3 その他の関数

- *length* (列の長さ)

  この関数は, 与えられた列の要素の数, すなわち長さを返す. 例えば,

  $$length\ [4,3,7,9,2] = 5$$

- *longest* (最長列)

  この関数は, 与えられた列集合の中から, 最も長さの長い列を返す. 例えば,

  $$longest\ \{[1,3],[\ ],[1],[2,3,1,4,2],[5,2,4]\} = [2,3,1,4,2]$$

- *longer* (列長比較)

  この関数は, 与えられた列のペアから, 長さの長い方の列を返す. ただし, 両方とも同じ長さの場合は前方の列を返す. 例えば,

  $$longer\ ([6,2,7],[1,3,2,2]) = [1,3,2,2]$$
  $$longer\ ([4,6,9],[5,0,2]) = [4,6,9]$$

- *sum* (要素の総和)

  この関数は, 与えられた列の要素の総和を返す. 例えば,

  $$sum\ [1,2,3,4,5] = 15$$

- *max* と *min* (列の最大要素, 最小要素)

  この関数は, 与えられた列の中から, 最大および最小の要素を返す. 例えば,

  $$max\ [3,8,-4,1,-6,7] = 8$$
  $$min\ [3,8,-4,1,-6,7] = -6$$

2.2.4 アルゴリズムを操作する宣言

- **if then else** (条件分岐)

  if の後の条件式が真であるときは **then** の直後の値を返し, 偽であるときは **else** の直後の値を返す. 例えば,

  $calc\ a = \textbf{if}\ \ a \geq 5\ \ \textbf{then}\ \ 10\ \ \textbf{else}\ \ 0$　のとき,
  $calc\ 7 = 10 \quad calc\ 3 = 0$

- **let in**

  let と **in** の間にある関係式を全て適用した上で **in** の直後の値を返す. 例えば,

  $calc\ (a,b) =$
  　$\textbf{let}\ \ c = a \times b$
  　　　$d = a + b$
  　$\textbf{in}\ (c+d, c \times d)$　のとき,

  $calc\ (2,4) = (14,48) \quad calc\ (1,3) = (7,12)$

2.3 最長部分列問題を求める関数

この節では, 2.2 節で述べて来た定義を元に, 最長部分列問題を求める関数である lps をどのように定義していくかを述べる. まずは, 列集合 X のうち, 述語 p を満たす列の集合を以下のような表記法で定義する.

$$p \lhd X = \{x \in X \mid p(x) = True\}$$

すると, この表記法を用いることにより, x の最長 p 部分列が

$$lps\ x = longest\ (p \lhd segs\ x)$$

というふうに表すことが出来るのは議論の余地はあるまい. また, 2.2 節 で述べた $segs$ の定義より,

$$\begin{aligned}
lps\ (x +\!\!+ [a]) &= longest\ (p \lhd (segs\ x \cup tails\ (x +\!\!+ [a]))\\
&= longest\ ((p \lhd segs\ x) \cup (p \lhd tails\ (x +\!\!+ [a])))\\
&= longer\ (longest\ (p \lhd segs\ x),\ longest\ (p \lhd tails\ (x +\!\!+ [a])))
\end{aligned}$$

が言えるので, 結局, 最長 p 部分列は以下のように帰納的に表すことが出来る.

$$\begin{aligned}
&lps\ [\ ] = [\ ]\\
&lps\ (x +\!\!+ [a]) = longer\ (lps\ x,\ longest\ (p \lhd tails\ x +\!\!+ [a]))
\end{aligned}$$

第 3 章

述語の閉属性に着目した解法

この章では, 論文 [1] を参考に, 述語に何らかの閉属性がある場合について, 最長部分列を線形時間で求める方法を与える. なお, この章では, tail , init , $\mathit{leftmost}$, $\mathit{rightmost}$, length , $+\!\!+$ といった演算は定数時間で出来るものと仮定して話を進めていく. また, $p(x)$ については, 3.3 節までは, とりあえず定数時間で計算できるものとし, 3.4 節で, 実際にはそれをどのように定数時間で計算すれば良いかを議論する.

3.1 述語の閉属性

述語の閉属性として, 以下の 4 つをこれより定義する. なお, これらの閉属性を持つ述語の例は 3.5 節にまとめて示してあるので, そちらも参照されたい.

- 述語 p が *prefix-closed* であるとは, 任意の列 x, y に対し,

$$p(x +\!\!+ y) = \mathit{True} \implies p(x) = \mathit{True}.$$

- 述語 p が *postfix-closed* であるとは, 任意の列 x, y に対し,

$$p(x +\!\!+ y) = \mathit{True} \implies p(y) = \mathit{True}.$$

- 述語 p が *segment-closed* であるとは,

$$p \text{ は } \mathit{prefix\text{-}closed} \text{ かつ } \mathit{overlap\text{-}closed}.$$

  これを言い換えると, 任意の列 x, y, z に対し,

$$p(x +\!\!+ y +\!\!+ z) = \mathit{True} \implies p(y) = \mathit{True}.$$

- 述語 p が *overlap-closed* であるとは, 任意の列 x, y, z に対し,

$$y \neq [\,] \wedge p(x +\!\!+ y) = \mathit{True} \wedge p(y +\!\!+ z) = \mathit{True} \implies p(x +\!\!+ y +\!\!+ z) = \mathit{True}.$$

3.2 述語が *prefix-closed* である場合の解法

この節では, p が *prefix-closed* である場合の解法を示すのが最終目的であるが, その前にいくつかの命題を与える.

命題 1 p が *prefix-closed* のとき, $t = longest\ (p \lhd tails\ x)$ とおくと,

$$longest\ (p \lhd tails\ (x +\!\!+ [a])) = longest\ (p \lhd tails\ (t +\!\!+ [a]))$$

となる.

この命題は, 列集合 $tails\ x$ の中で p を満たす最長の列が t であるとしたとき, 列集合 $tails\ (x +\!\!+ [a])$ の中で, $t +\!\!+ [a]$ よりも長い列が p を満たすことはありえない, ということを述べている. 何故なら, 図 3.1 において, t よりも長い $t' \in tails\ x$ に対して, $p(t' +\!\!+ [a]) = True$ だったとすると, *prefix-closed* の性質より, $p(t') = True$ となってしまい, t が $p \lhd tails\ x$ の中で最長であるという事実に反するからである.

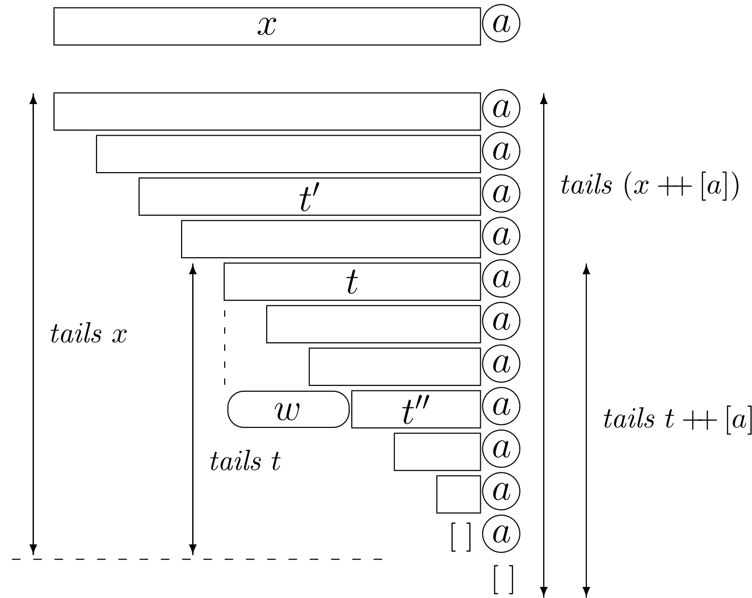

図 3.1　命題 1, 2

命題 2 p が *prefix-closed* かつ *overlap-closed* であるとき, $t = longest\ (p \lhd tails\ x)$ とおくと, $longest\ (p \lhd tails\ (x +\!\!+ [a]))$ は, $t +\!\!+ [a]$, $[a]$, $[\ ]$ のいずれかである.

証明　まず, t よりも長い $t' \in tails\ x$ に対しては, 必ず $p(t' +\!\!+ [a]) = False$ となることは, 命題 1 で示した通りである. 次に, 図 3.1 にあるように, ある t よりも短い（但し空でない）$t'' \in tails\ x$ に対して, $t'' +\!\!+ [a] = longest\ (p \lhd tails\ (x +\!\!+ [a]))$ となると仮定する. すると, t も t'' も $tails\ x$ に含まれることから, ある空でない列 w を用いて, $t = w +\!\!+ t''$ とできることが, 図 3.1 からも分かるだろう. ここで, $p(w +\!\!+ t'') = True \wedge p(t'' +\!\!+ [a]) = True$ より, p が *overlap-closed* であることから, $p(w +\!\!+ t'' +\!\!+ [a]) = True$, すなわち, $p(t +\!\!+ [a]) = True$ が言える. ところが, これは $t'' +\!\!+ [a]$ が $p \lhd tails\ (x +\!\!+ [a])$ の中で最長であるとした仮定に矛盾する. よって, $t'' +\!\!+ [a] = longest\ (p \lhd tails\ (x +\!\!+ [a]))$ とした仮定は誤りであり, $longest\ (p \lhd tails\ (x +\!\!+ [a]))$ となりうるのは, $t +\!\!+ [a]$, $[a]$, $[\ ]$ のいずれかである. ■

3.2.1　述語が *overlap-closed* でもある場合

p が *prefix-closed* でも *overlap-closed* でもある場合は, 以上の 2 つの命題を用いることにより, 最長 p 部分列を求める real-time アルゴリズムを比較的容易に導くことが出来る. そのアルゴリズムを以下に示す.

$lps\ x = \textbf{let}\ (s,t) = lps'\ x\ \textbf{in}\ s$

$lps'\ [\,] = ([\,],[\,])$
$lps'\ (x +\!\!+ [a]) =$
$\quad \textbf{let}\ \ (s,t) = lps'\ x$
$\qquad t' = \textbf{if}\ \ p(t +\!\!+ [a]) = True\ \ \textbf{then}\ \ t +\!\!+ [a]$
$\qquad t' = \textbf{if}\ \ p(t +\!\!+ [a]) = False \wedge p[a] = True\ \ \textbf{then}\ \ [a]$
$\qquad t' = \textbf{if}\ \ p(t +\!\!+ [a]) = False \wedge p[a] = False\ \ \textbf{then}\ \ [\,]$
$\qquad s' = longer\ (s,t')$
$\quad \textbf{in}\ (s',t')$

3.2.2 Windowing 手法

それでは, p が *overlap-closed* ではなく, ただ単に *prefix-closed* だけである場合はどうすれば良いであろうか. それを解くために, windowing 手法というテクニックを用いる必要がある. まずはその方法について簡単に説明しておこう. この手法では, 不変式として,

$$s = longest\ (p \triangleleft segs\ x) \wedge u \in tails\ x \wedge length\ u = length\ s$$

が常に成り立っている. ここで, u の長さが x の最長 p 部分列の長さと同じということは, $p \triangleleft tails\ x$ の中で最長の列の長さも $length\ u$ を越えることはないということは議論の余地はあるまい. よって, 命題 1 より, $longest\ (p \triangleleft tails\ (x +\!\!+ [a]))$ の長さは $(length\ u) + 1$ を越えることは無い. よって, $p(u +\!\!+ [a])$ が真であるときだけ, s を $u +\!\!+ [a]$ に更新すれば良く, 偽のときは s はそのままで, u だけを $tail\ (u +\!\!+ [a])$ に更新すれば良い.

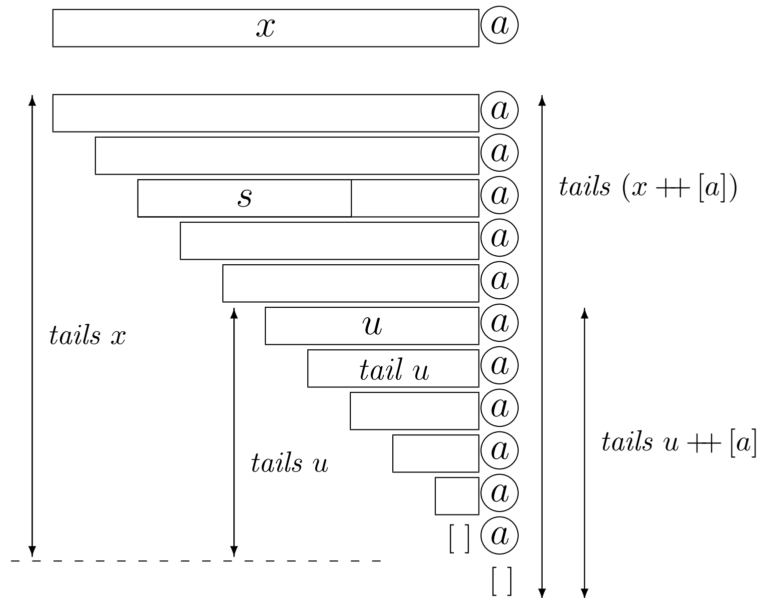

図 3.2 Windowing 手法

p が *prefix-closed* の場合の最長部分列問題に関して, windowing 手法を用いた real-time アルゴリズムを以下に示す.

$$lps\ x = \textbf{let}\ (s,u) = lps'\ x\ \textbf{in}\ s$$

$$lps'\ [\,] = ([\,],[\,])$$
$$\begin{aligned}
&lps'\ (x +\!\!+ [a]) = \\
&\quad \textbf{let}\ \ (s,u) = lps'\ x \\
&\quad\quad s' = \textbf{if}\ \ p(u +\!\!+ [a]) = \mathit{True}\ \ \textbf{then}\ \ u +\!\!+ [a]\ \ \textbf{else}\ \ s \\
&\quad\quad u' = \textbf{if}\ \ p(u +\!\!+ [a]) = \mathit{True}\ \ \textbf{then}\ \ u +\!\!+ [a]\ \ \textbf{else}\ \ \mathit{tail}\ (u +\!\!+ [a]) \\
&\quad \textbf{in}\ (s',u')
\end{aligned}$$

3.3 列分割を適用した解法

ここでは, 列分割という考え方を用いた最長部分列問題の解法について述べたいと思う. 実際, この列分割を用いた解法は, p が *postfix-closed* の場合などに深く関係している.

3.3.1 列分割の定義

ここでは, 列分割と, 極大列分割について定義する.

- 列分割 (partition)

  列分割とは, 列をいくつかの部分列に分けたもののことである. 例えば, $x = x_1 +\!\!+ x_2 +\!\!+ x_3 +\!\!+ x_4$ のとき, 列分割は, $xs = [x_1, x_2, x_3, x_4]$ などのように, 列を要素として持つような列として表される. ただし, 列分割の要素として空列は含まないものとする.

- 極大列分割 (maximal partition)

  極大列分割とは, その列分割の要素である列自体は全て p を満たすが, それらを連結させたものに対しては, いずれも p を満たさないような列分割のことである. 例えば, $x = x_1 +\!\!+ x_2 +\!\!+ x_3 +\!\!+ x_4$ としたとき, $p(x_1) = p(x_2) = p(x_3) = p(x_4) = \mathit{True}$ なのに対し, $p(x_1 +\!\!+ x_2) = p(x_2 +\!\!+ x_3) = p(x_3 +\!\!+ x_4) = \mathit{False}$ となるようなものが極大列分割である. なお, この極大列分割は, ただ一つしか存在しないとは限らない.

3.3.2 欲張り法

次に, 極大列分割を求めるアルゴリズムとして, 欲張り法 (greedy algorithm) を紹介する. このアルゴリズムの内容について, 図 3.3 と照らし合わせながら説明しよう.

まず, 要素が一つだけの列 $[c]$ に対する極大列分割は $[[c]]$ である. そして, x に対する極大列分割を $xs = [x_1, x_2, ..., x_{n-1}, x_n]$ としたとき, $x +\!\!+ [a]$ の極大列分割は次のようにして帰納的に求める. まず $x_n +\!\!+ [a]$ が p を満たすがどうか調べ, p を満たさないなら, それを新しい列分割の要素として加えて, $x +\!\!+ [a]$ の極大列分割とする. 一方, p を満たすなら, 次に $x_{n-1} +\!\!+ x_n +\!\!+ [a]$ が p を満たすかどうか調べ, それが p を満たさないなら, $x +\!\!+ [a]$ の極大列分割は $[x_1, x_2, ..., x_{n-1}, x_n +\!\!+ [a]]$ となり, 一方それが p を満たすなら, 次は $x_{n-2} +\!\!+ x_{n-1} +\!\!+ x_n +\!\!+ [a]$ が p を満たすかどうか調べていく. このようにして, p を満たさなくなるか, または $xs = [x +\!\!+ [a]]$ になるまで, 右から繋げていくという操作を行なうというものである.

以下に, 欲張り法を用いた, 極大列分割を求めるアルゴリズムを記述しておく.

$maxpart\ [a] = [[a]]$

$maxpart\ (x \mathbin{+\!\!+} [a]) =$
  let $xs = maxpart\ x$
  $(xs', y') = absorb\ (xs, [a])$
  in $xs' \mathbin{+\!\!+} [y']$

$absorb\ (xs, y) =$
  if $xs \neq [\ ] \wedge p((rightmost\ xs) \mathbin{+\!\!+} y) = True$
  then $absorb\ (init\ xs\ ,\ (rightmost\ xs) \mathbin{+\!\!+} y)$
  else (xs, y)

さて, この欲張り法というアルゴリズムだが, 一回のループで最大 n 回の演算を行なうため, 一見 $O(n^2)$ はありそうに見えるだろう. しかし, 実際は $O(n)$ であることが分かっている. ここではその詳しい証明はしないが, 直観的な説明としては, 欲張り法の一回のループでたくさんの連結操作を行なったときは, 次のループでは列分割の要素数は減っているため, 連結操作の数も少なくなるという感じである.

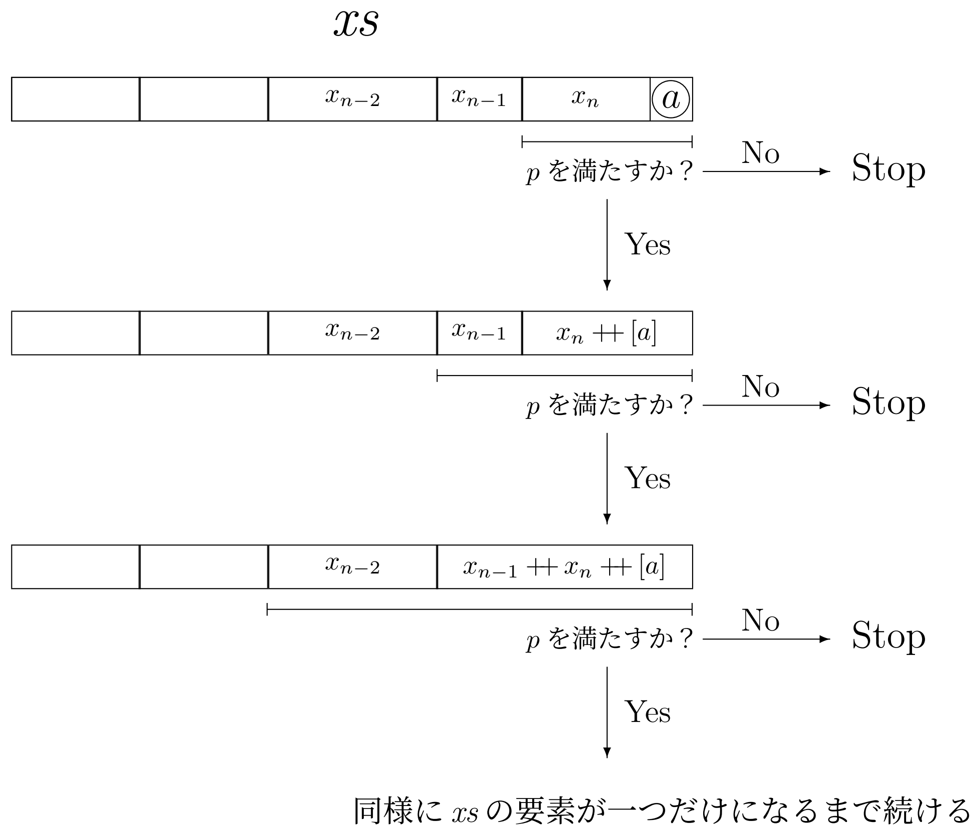

図 3.3　欲張り法

3.3.3 欲張り法と合わせて用いる命題

極大列分割や欲張り法を最長部分列問題に適用するに当たって, いくつか述べておかねばならない命題があるので, それらをここに示す. まず, p_R という述語を次のように定義する. (図 3.4 参照)

$$p_R(x) \equiv \forall a \in init\ x : aR(rightmost\ x) = True$$

例えば, R が '$\leq$' のときは, $p_{\leq}(x) \equiv rightmost\ x = max\ x$ である. なお, この述語が *postfix-closed* であることは, ほぼ自明であろう. これに関する命題を以下に示す.

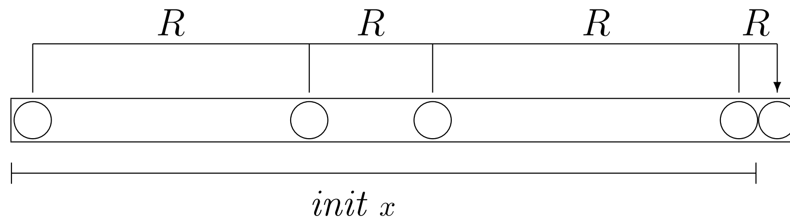

図 3.4 述語 p_R

命題 3 R が推移律を満たすならば, p_R は *overlap-closed* である.

証明 x, y, z を $p_R(x \mathbin{+\!\!+} y) = p_R(y \mathbin{+\!\!+} z) = True$ を満たす任意の空でない列, また, $b = rightmost\ y$, $c = rightmost\ z$ とする. そのとき, $p_R(y \mathbin{+\!\!+} z) = True$ より, $\forall d \in init\ (y \mathbin{+\!\!+} z) : dRc$ が言える. また, $p_R(x \mathbin{+\!\!+} y) = True \wedge bRc$, そして R の推移性より, $\forall a \in init\ (x \mathbin{+\!\!+} y) : aRc$ が言える. よって, $p_R(x \mathbin{+\!\!+} y \mathbin{+\!\!+} z) = True$ が言えるので, p_R は *overlap-closed* である. ■

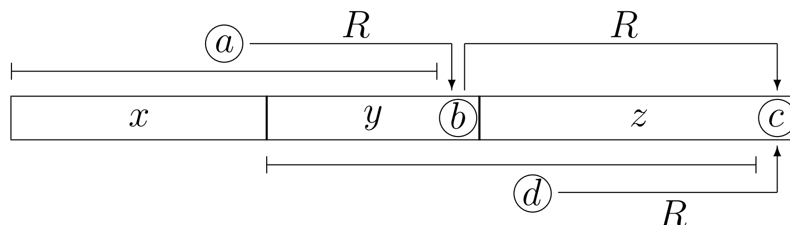

図 3.5 命題 3

命題 4 R が推移律を満たす関係で, u, v が p_R を満たす空でない列であるとき,

$$p_R(u \mathbin{+\!\!+} v) \equiv (rightmost\ u)R(rightmost\ v)$$

である.

証明 $a = rightmost\ u$, $b = rightmost\ v$ とする. まず, $aR^C b$ のときだが, このときは, $p_R(u \mathbin{+\!\!+} v) = False$ であることは自明である. 次に aRb のときだが, $\forall d \in v$ に対しては仮定より cRb は明らか. また, $\forall d \in u$ に対しては, u が p_R を満たすので, cRa であることと R が推移的であることより, cRb が言える. よって, $p_R(u \mathbin{+\!\!+} v) = True$ が言える. ■

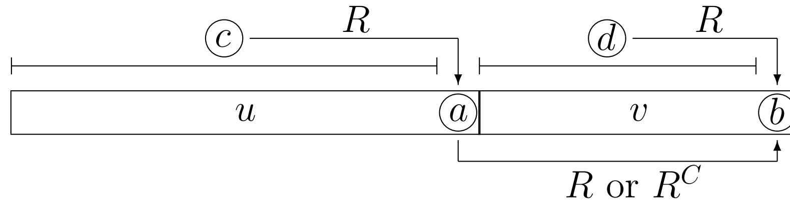

図 3.6　命題 4

命題 5 p を *postfix-closed* かつ *overlap-closed*, xs を x の p に対する極大列分割としたとき,

$$longest\ (p \lhd tails\ x) = rightmost\ xs$$

となる.

証明　$w = longest\ (p \lhd tails\ x)$, $v = rightmost\ xs$, $u = rightmost\ (init\ xs)$ とおく. まず, $u +\!\!+ v \subset w$ のとき (図 3.7 w_1) は, $p(w) = True$ かつ, p は *postfix-closed* より, $p(u +\!\!+ v) = True$ となり, xs の極大性に矛盾する. 次に, $v \subset w \subset u +\!\!+ v$ のとき (図 3.7 w_2) は, $p(u) = True \wedge p(w) = True$ で, p は *overlap-closed* より, $p(u +\!\!+ v) = True$ となり, これも矛盾. 最後に, $w \subset v$ のとき (図 3.7 w_3) であるが, これは $p(v) = True \wedge v \in tails\ x$ より, w の最長性に矛盾. よって, $w = v$ すなわち, $longest\ (p \lhd tails\ x) = rightmost\ xs$ である. ■

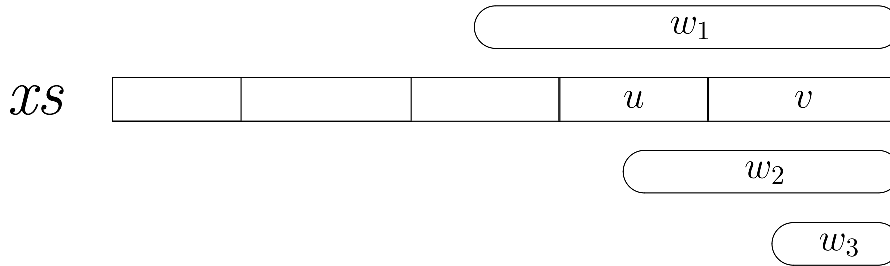

図 3.7　命題 5

3.3.4　最長 p_R 部分列問題の解法

R が推移的でかつ一定時間で計算できるならば, 先ほどの三つの命題を利用することができるので, 欲張り法を用いて, 最長 p_R 部分列を求める線形な on-line アルゴリズムが導ける. そのアルゴリズムを以下に示す.

```
lps x = let (xs, s) = lps' x  in s

lps' [a] = ([[a]], [a])
lps' (x ++ [a]) =
    let  (xs, s) = lps' x
         (xs', y') = absorb (xs, [a])
         s' = longer (s, y')
    in (xs' ++ [y'], s')
```

$$
\begin{aligned}
&absorb\ (xs, y) = \\
&\quad \textbf{if}\ \ xs \neq [\ ] \wedge (rightmost\ (rightmost\ xs))R(rightmost\ y) \\
&\qquad \textbf{then}\ \ absorb\ (init\ xs\ ,\ (rightmost\ xs) \mathbin{+\!\!+} y) \\
&\qquad \textbf{else}\ \ (xs, y)
\end{aligned}
$$

なお, この最長 p_R 部分列問題の解法に関して, p_R が *postfix-closed* だということは, 元の列を逆にすれば, *prefix-closed* の場合と同じだから, *prefix-closed* の場合と同じようにすれば良いように一見思われるかもしれない. しかし, 実際問題として, データの転送及びそれを読む際は向きが一方的であるため, アルゴリズム自体もオンラインであることが望ましい. よって, 最長 p_R 部分列問題を解くには列分割を用いた解法の方がより望ましいと言える.

3.4 $p(x)$ の計算時間

今までは $p(x)$ が定数時間で計算できるものとして話を進めてきたが, 実際にそうなのだろうか. 例えば, $p(x) \equiv leftmost\ x = min\ x$ などといった述語を考えてみよう. この述語を計算するのに定数時間で出来るだろうか. 当然答えは否である. 何故なら, 与えられた列に対して最小値を計算するためには, 全ての要素を比較していかなければならないため, 要素数と同じだけの演算が必要だからだ. しかし, 以下のことは比較的容易に言えるであろう.

$$
min\ (x \mathbin{+\!\!+} [a]) = \begin{cases} min\ x, & \text{when}\ \ min\ x \leq a \\ a, & \text{when}\ \ min\ x \geq a \end{cases}
$$

これをみれば明らかなように, $min\ x \mathbin{+\!\!+} [a]$ は $min\ x$ の答えが分かっていれば, それを用いて定数時間で計算できるのである. つまり, $p(x)$ を定数時間で計算するためには, 下の式において 'something' の部分が定数時間で計算できるという条件が必要になってくると言えよう.

$$
p(x \mathbin{+\!\!+} [a]) = p(x) \wedge \text{'something'}
$$

3.5 閉属性を持つ述語の例

以下の表に閉属性を持つ述語の例をいくつか挙げておく.

表 3.1 述語の例

閉属性	述語
prefix-closed $\wedge$ *overlap-closed*	$leftmost\ x = min\ x\ (max\ x)$
postfix-closed $\wedge$ *overlap-closed*	$rightmost\ x = max\ x\ (min\ x)$
segment-closed	$length\ x < min\ x$
segment-closed	$max\ x - min\ x \leq C$
segment-closed	$sum\ x \leq C\ (\forall c \in x : c \geq 0)$
overlap-closed	$length\ x > max\ x$
overlap-closed	$leftmost\ x = min\ x \wedge rightmost\ x = max\ x$

第 4 章

最左最右関係問題

最左最右関係問題 (Leftmost at most rightmost problem) とは, ある条件を満たす関係 R に対して,

$$p(x) \;\equiv\; (\mathit{leftmost}\ x)R(\mathit{rightmost}\ x) \vee x = [\,]$$

で定義される述語 p を満たすような最長部分列を求める問題である. ただし, この述語 p は, 前章で述べた 4 つの閉属性のいずれも満たしていないので, この章では最左最右関係問題を閉属性とは別の, 付随列を用いた観点から解いていく.

なお, 最左最右関係問題を付随列を用いて $O(n)$ で解くという方法は論文 [1] を参考にしているが, そのアルゴリズムが正しいことの証明, および 4.4 節で述べる最左最右関係問題の拡張については, 本研究で得られた成果である. また, この章でも第 3 章と同様, tail , init , $\mathit{leftmost}$, $\mathit{rightmost}$, length , $+\!\!+$ といった演算は定数時間で出来るものと仮定して話を進めていく

4.1 関係 R の性質

最左最右関係問題において, 関係 R に要求される条件は, R が反対称かつ R^C が推移的というものである. 但し, R^C とは

$$aR^Cb \;\equiv\; \neg aRb$$

で定義される関係である. また $R^C_=$ という関係についても, 論文中に用いるので, その定義が

$$aR^C_=b \;\equiv\; aR^Cb \vee a = b$$

であるということを, ここに示しておく.

次に反対称および推移的という単語の意味について説明する.

- 反対称 (antisymmetric)

  関係 R が反対称であるとは, 全ての要素 a, b に対して,

  $$aRb \wedge bRa \implies a = b$$

  ということである.

- 推移的 (transitive)

  関係 R が推移的であるとは, 全ての要素 a, b, c に対して,

  $$aRb \wedge bRc \implies aRc$$

  ということである.

ちなみに, 全ての要素と言っても, 例えば '$\leq$' に対する複素数のような, R の比較の対象とならない要素については除くものとする. なお, この論文では基本的に要素といえば実数を指しているので, 全ての要素とある場合には, それは全ての実数と読み替えても差し支えない.

次に, R が反対称のときに, 以下の命題が成り立つので, それについても触れておく.

命題 6 関係 R が反対称のとき, 全ての要素 a, b に対して,

$$aRb \implies bR^C_= a$$

である.

証明 先述の反対称の定義について対偶をとると, $a \neq b \Rightarrow aR^C b \vee bR^C a$. 故に, aRb が成り立つならば, $a \neq b \Rightarrow bR^C a$ が言えて, これはすなわち, $aRb \Rightarrow bR^C a \vee a = b$ と同値である. ■

なお, R^C が推移的であるとき, $R^C_=$ も推移的であるが, これは自明であろう.

4.2 線形解法

4.2.1 付随列の定義

最左最右関係問題を $O(n)$ で解くために, ここでは x に付随する列 $auxList\ x$ を利用する. それは以下のようなアルゴリズムで帰納的に定義される. なお, ここで問題になるのが, $rightmost\ [\,]$ をどのように定義すれば良いかということだが, これに関しては, 番兵として $\forall a \in x : \omega R^C a$ となるような ω を, $rightmost\ [\,] = \omega$ と定義しておけば問題はない. (図 4.1 参照)

$auxList\ [\,] = [\,]$

$auxList\ (x \mathbin{+\!\!+} [a]) =$
  let $y = auxList\ x$
    $a' =$ **if** $(rightmost\ y)Ra$ **then** $rightmost\ y$
    $a' =$ **if** $(rightmost\ y)R^C a$ **then** a
  in $y \mathbin{+\!\!+} [a']$

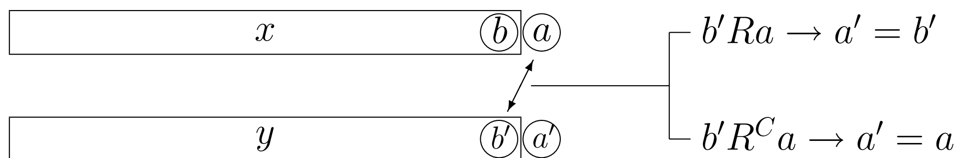

図 4.1 付随列 ($y = auxList\ x$)

これだけだと分かりにくいので, 例を挙げてみよう. R が '$\leq$' のとき, x に対して $y = auxList\ x$ としたら,

$$x = [7, 9, 6, 8, 6, 5, 4, 6, 3, 7, 2, 1, 8, 5]$$
$$y = [7, 7, 6, 6, 6, 5, 4, 4, 3, 3, 2, 1, 1, 1]$$

となる.

すなわち上の例の場合, x の新しい要素が, 現在の y の最右要素がよりも等しいか大きければ, y の新しい要素は今までの要素を維持し, 小さければその x の新しい要素をそのまま y の新しい要素とする.

4.2.2 不変式

与えられた $x = x_1 +\!\!+\, x_2$ に対して, $y_1 +\!\!+\, y_2 = auxList\,(x_1 +\!\!+\, x_2) \wedge length\ x_1 = length\ y_1 \wedge length\ x_2 = length\ y_2$ を満たす全ての x_1, x_2, y_1, y_2 に対して, 以下の 2 つの不変式が言える.

不変式 1 $\forall c \in x_1 : cR^C_= rightmost\ y_1$

不変式 2 $y_1 \neq [\ ] \wedge rightmost\ y_1 \neq leftmost\ y_2 \implies leftmost\ x_2 = leftmost\ y_2$

証明 $x_1 = [a_1, a_2, \ldots, a_i], x_2 = [a_{i+1}, a_{i+2}, \ldots, a_n], y_1 = [b_1, b_2, \ldots, b_i], y_2 = [b_{i+1}, b_{i+2}, \ldots, b_n]$ とおく. そのとき, 付随列の定義の仕方より, 1 以上 $n-1$ 以下の任意の整数 k に対して以下のことが言える.

$$b_k R a_{k+1} \implies b_{k+1} = b_k \tag{4.1}$$

$$b_k R^C a_{k+1} \implies b_{k+1} = a_{k+1} \tag{4.2}$$

ここで, (4.1) 式の対偶をとれば, $b_{k+1} \neq b_k \Rightarrow b_k R^C a_{k+1}$ が言えて, それと (4.2) 式より, $b_{k+1} \neq b_k \Rightarrow b_{k+1} = a_{k+1}$ が言える. よって, $k = i$ を代入することで, まず不変式 2 は証明された.

次に, 不変式 1 の証明をする. まずは, b_k と b_{k+1} の関係だが, $b_k R a_{k+1}$ の場合は, (4.1) 式より, $b_{k+1} = b_k$ となり, また $b_k R^C a_{k+1}$ の場合は, (4.2) 式より, $b_k R^C b_{k+1}$ となるので, 全ての k に対し, $b_k R^C_= b_{k+1}$ であることが言える. 次に a_k と b_k の関係だが, $b_{k-1} R^C a_k$ のときは, (4.2) 式より, 容易に $a_k = b_k$ が言え, また, $b_{k-1} R a_k$ のときは, (4.1) 式より, $b_k = b_{k-1}$ なので $b_k R a_k$ となり, R の反対称性から, 命題 6 より $a_k R^C_= b_k$ が言える. よって, 以上より $a_k R^C_= b_k$ と $b_k R^C_= b_{k+1}$ が言えたので, $R^C_=$ の推移性より, 不変式 1 が言える. ■

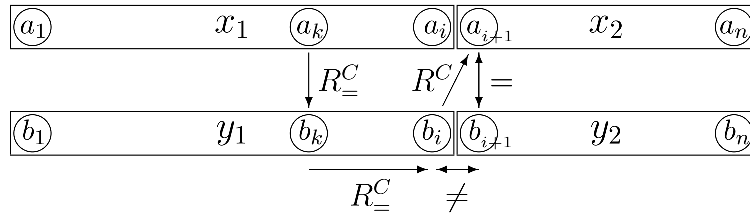

図 4.2 不変式

4.2.3 解法

先述の不変式は x_1 と x_2 の決め方に依らず成り立つ. よって, この不変式から導かれる *slide* 関数と, 前章の windowing 手法を用いることにより, 最左最右関係問題を線形時間で解くアルゴリズムを求めることができる.

windowing は新しい要素を取り入れる毎に一回だけ行なう操作で, 図 4.3 のように,

$$x_1' = x_1 +\!\!+ [leftmost\ x_2],\ x_2' = tail\ x_2 +\!\!+ [a]$$
$$y_1' = y_1 +\!\!+ [leftmost\ y_2],\ y_2' = tail\ y_2 +\!\!+ [a]$$

というふうにして, 一旦列の境目を右に動かす. この際 x_2, y_2 の長さは変わらないことに注意されたい.

次に, *slide* 関数についてだが, この関数は, y_1 の最右要素と新しく取り込んだ要素である a とを比較し, その関係が R を満たしてたら, x_1 と x_2 及び y_1 と y_2 を区切る敷居を一つずつずらしていくと

いう作業を行なう. また, この作業は関係が R^C になるか, x_1, y_1 が空列になるまでずっと続けられる. (図 4.4 参照)

さて, この windowing 手法と *slide* 関数を用いたアルゴリズムの内容を以下に示す.

$$
\begin{array}{l}
\mathit{lps}\ x = \mathbf{let}\ (x_1, x_2, y_1, y_2, s) = \mathit{lps'}\ x\ \mathbf{in}\ s \\
\mathit{lps'}\ [\,] = ([\,],[\,],[\,],[\,],[\,]) \\
\mathit{lps'}\ (x +\!\!+ [a]) = \\
\quad \mathbf{let}\ \ (x_1, x_2, y_1, y_2, s) = \mathit{lps'}\ x \\
\qquad a' = \mathbf{if}\ \ (\mathit{rightmost}\ y_2) R a\ \ \mathbf{then}\ \ \mathit{rightmost}\ y_2 \\
\qquad a' = \mathbf{if}\ \ (\mathit{rightmost}\ y_2) R^C a\ \ \mathbf{then}\ \ a \\
\qquad (x_1', x_2', y_1', y_2') = \mathit{window}\ (x_1, x_2, y_1, y_2, a, a') \\
\qquad (x_1'', x_2'', y_1'', y_2'') = \mathit{slide}\ (x_1', x_2', y_1', y_2', a) \\
\qquad s' = \mathit{longer}\ (s, x_2'') \\
\quad \mathbf{in}\ (x_1'', x_2'', y_1'', y_2'', s') \\
\mathit{window}\ (x_1, x_2, y_1, y_2, a, b) = \\
\quad \mathbf{let}\ \ x_2' = x_2 +\!\!+ [a] \\
\qquad y_2' = y_2 +\!\!+ [b] \\
\qquad x_1'' = x_1 +\!\!+ [\mathit{leftmost}\ x_2'] \\
\qquad x_2'' = \mathit{tail}\ x_2' \\
\qquad y_1'' = y_1 +\!\!+ [\mathit{leftmost}\ y_2'] \\
\qquad y_2'' = \mathit{tail}\ y_2' \\
\quad \mathbf{in}\ (x_1'', x_2'', y_1'', y_2'') \\
\mathit{slide}\ (x_1, x_2, y_1, y_2, a) = \\
\quad \mathbf{if}\ \ y_1 \neq [\,] \wedge \mathit{rightmost}\ y_1 R a \\
\qquad \mathbf{then}\ \ \mathit{slide}\ (\mathit{init}\ x_1, [\mathit{rightmost}\ x_1] +\!\!+ x_2, \mathit{init}\ y_1, [\mathit{rightmost}\ y_1] +\!\!+ y_2) \\
\qquad \mathbf{else}\ \ (x_1, x_2, y_1, y_2)
\end{array}
$$

以上のアルゴリズムが正しいことを次頁の図 4.5 を用いて, 簡単に示しておこう. そのためには, x_2 に新たに付け加わることになる要素 a に対して, 次の式が正しいことを言ってやればよい.

$$(\mathit{rightmost}\ y_1) R^C a \wedge (\mathit{leftmost}\ y_2) R a \implies x_2 +\!\!+ [a] = \mathit{longest}\ (p \triangleleft \mathit{tails}\ (x +\!\!+ [a]))$$

証明 まずは, 先程と同様に $x_1 = [a_1, a_2, \ldots, a_i]$, $x_2 = [a_{i+1}, a_{i+2}, \ldots, a_n]$, $y_1 = [b_1, b_2, \ldots, b_i]$, $y_2 = [b_{i+1}, b_{i+2}, \ldots, b_n]$ とおく. そのとき, $b_i R^C a \wedge b_{i+1} R a$ より, 明らかに $b_i \neq b_{i+1}$ なので, 先程の不変式 2 より $a_{i+1} = b_{i+1}$ が成り立ち, 故に $a_{i+1} R a$ が言える. また, 不変式 1 より, $\forall k \leq i : a_k R^C_{=} b_i$ が言えて, $b_i R^C a$ と $R^C_{=}$ の推移性より, $\forall k \leq i : a_k R^C a$ となる. よって, x_1 の全ての要素は, a に対して R^C の関係であり, また x_2 の最左要素は, a に対して R の関係であるから, $x_2 +\!\!+ [a] = \mathit{longest}\ (p \triangleleft \mathit{tails}\ (x +\!\!+ [a]))$ であることが示された. ■

4.3 他の問題への応用

最左最右関係問題は列の左右の要素を比較するだけに留まらず, さらに発展した問題に対しても応用が効く. その例をいくつか挙げてみよう.

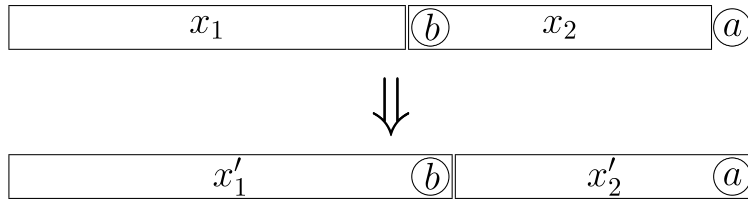

図 4.3　windowing

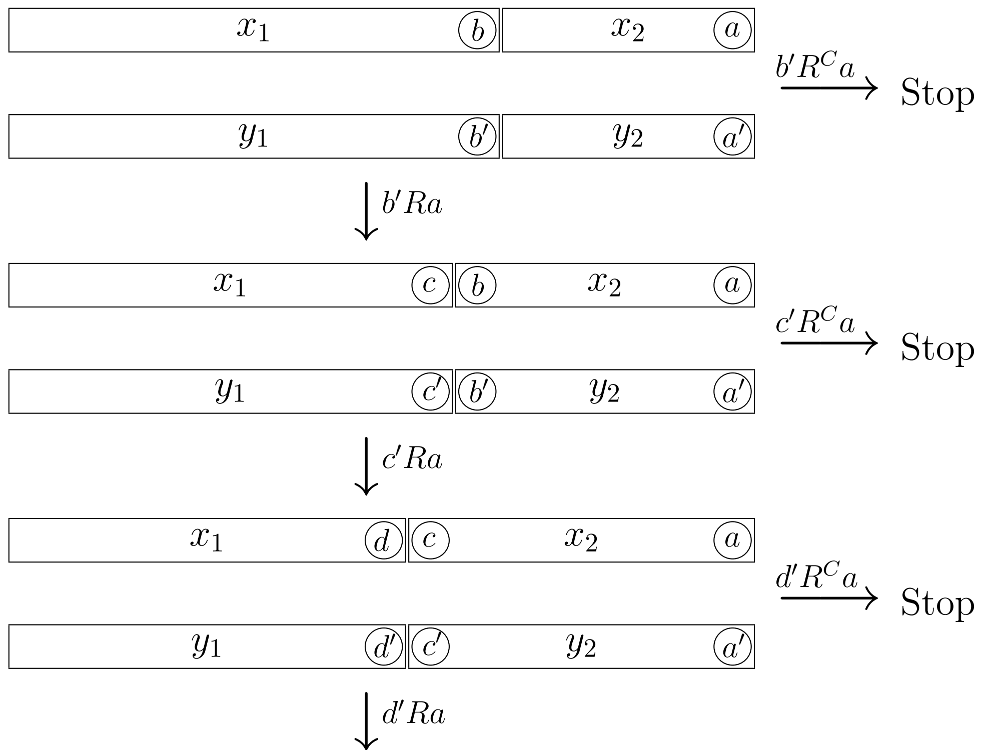

図 4.4　*slide* 関数

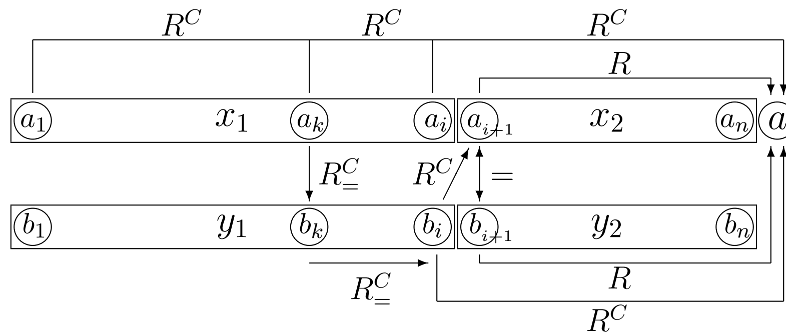

図 4.5 アルゴリズム

4.3.1 部分列和の正負に関する問題

まず, ここで扱う問題は, 列の和が非負であるような, 最長部分列を求めるというものである. すなわち, 述語 p が

$$p(x) \;\equiv\; sum\ x \geq 0$$

で表されるというものである. 基本的な考え方としては, 元の列を $x = [a_1, a_2, ..., a_n]$ としたとき, それを付随的な列

$$x' = [0, a_1, a_1 + a_2, ..., \sum_{i=1}^{n} a_i]$$

に写像して, その付随列 x' に対して R が '$\leq$' の場合の最左最右関係問題を解けば良い.

4.3.2 部分列平均に関する問題

次に扱う問題は, 列の平均がある定数 C 以上であるような, 最長部分列を求めるというものである. この場合, 述語 p は

$$p(x) \;\equiv\; \frac{sum\ x}{length\ x} \geq C$$

で表される. これについての解法は, まず列の全ての要素からそれぞれ C を引いておき, $x' = [a_1 - C, a_2 - C, \ldots, a_n - C]$ に対して, 先ほどの, 列の和が非負であるような最長部分列を求める問題に置き換えればよい. なお, この問題に関しては, [3] に別の線形解法が載っているので, それも参照されたい.

4.3.3 部分列和と定数の比較に関する問題

部分列和が非負となる問題については, 先程述べた通りであるが, これを応用して部分列和がある定数 C を越えないという問題を考えたい. すなわち, 述語 p が

$$p(x) \;\equiv\; sum\ x \leq C$$

で表されるといったものである. そのとき, 元の列 $x = [a_1, a_2, ..., a_n]$ を付随列 $x' = [0, a_1, a_1 + a_2, ..., \sum_{i=1}^{n} a_i]$ に写像するまでは先程の問題と同じで良いのだが, それを最左最右関係問題に適用する方法が先程の問題と異なってくる. つまり,

$$p'(x') \;\equiv\; leftmost\ x' \geq rightmost\ x' - C$$

という風に, 不等号の向きが変わるだけでなく, 最右要素に定数演算が加わったものの比較となってしまうのだ. まず, 不等号の向きだが, これはどちら向きでも R の条件を満たすので, 問題はない. しかし, 左右の要素に演算を加えたものを R で比較するということを最左最右関係問題と同じようなアルゴリズムで解くことが出来るのだろうか. それを次の節で検証してみたい.

4.4 関数化への拡張

前節で述べた $p'(x) \equiv \mathit{leftmost}\ x \geq \mathit{rightmost}\ x - C$ や, または, $p(x) \equiv (\mathit{leftmost}\ x)(\mathit{rightmost}\ x)^2 \leq C$ などといった述語に対する最長部分列問題は解けないだろうか.

ここでは, そのような問題を解決するために, 左右の要素を関数化した場合, すなわち, 述語が

$$p(x) \equiv \phi(\mathit{leftmost}\ x) R \psi(\mathit{rightmost}\ x)$$

の場合について, 最左最右関係問題を拡張していけるのかどうか検証してみる. なお, この節は 4.2 節とほぼ同じ形の構成で進めているので, 両者を比べてみると分かりやすいであろう.

4.4.1 付随列の定義

まずは, 付随列 $\mathit{auxList}\ x$ をどのように作るかということが問題になってくる. 何故なら, 通常の最左最右関係問題のときの解法でも分かるように, 付随列 $\mathit{auxList}\ x$ は最長部分列の最左要素を決定するのに欠かせないからだ. しかし, 最左要素に ϕ という関数がかかっている以上, 単純に要素同士を比較する訳には行かない. よって, ここではまず, 以下のように関数 ϕ を掛けた形で, 付随列 y を定義しておく. (図 4.6 参照)

$$
\begin{aligned}
&\mathit{auxList}\ [\,] = [\,] \\
&\mathit{auxList}\ (x +\!\!+ [a]) = \\
&\quad \mathbf{let}\ \ y = \mathit{auxList}\ x \\
&\quad\quad a' = \mathbf{if}\ \ (\mathit{rightmost}\ y) R \phi(a)\ \ \mathbf{then}\ \ y \\
&\quad\quad a' = \mathbf{if}\ \ (\mathit{rightmost}\ y) R^C \phi(a)\ \ \mathbf{then}\ \ \phi(a) \\
&\quad \mathbf{in}\ y +\!\!+ [a']
\end{aligned}
$$

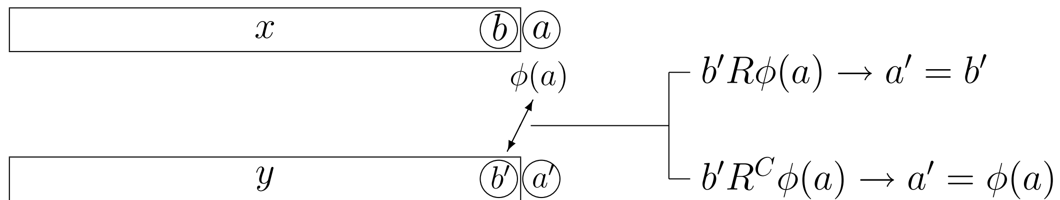

図 4.6 付随列 ($y = \mathit{auxList}\ x$)

4.4.2 不変式

次に不変式がどのように変わるか考えてみる.

与えられた $x = x_1 +\!\!+ x_2$ に対して, $y_1 +\!\!+ y_2 = \mathit{auxList}\,(x_1 +\!\!+ x_2) \wedge \mathit{length}\ x_1 = \mathit{length}\ y_1 \wedge \mathit{length}\ x_2 = \mathit{length}\ y_2$ を満たす全ての x_1, x_2, y_1, y_2 に対して, 以下の 2 つの不変式が言える.

不変式 1' $\forall c \in x_1 : \phi(c) R^C_= \mathit{rightmost}\ y_1$

不変式 2' $y_1 \neq [\,] \wedge \mathit{rightmost}\ y_1 \neq \mathit{leftmost}\ y_2 \implies \phi(\mathit{leftmost}\ x_2) = \mathit{leftmost}\ y_2$

証明　$x_1 = [a_1, a_2, \ldots, a_i], x_2 = [a_{i+1}, a_{i+2}, \ldots, a_n], y_1 = [b_1, b_2, \ldots, b_i], y_2 = [b_{i+1}, b_{i+2}, \ldots, b_n]$ とおく. そのとき, 列 y の定義の仕方より, 1 以上 $n-1$ 以下の任意の整数 k に対して以下のことが言える.

$$b_k R \phi(a_{k+1}) \implies b_{k+1} = b_k \tag{4.3}$$

$$b_k R^C \phi(a_{k+1}) \implies b_{k+1} = \phi(a_{k+1}) \tag{4.4}$$

ここで, (4.3) 式の対偶をとれば, $b_{k+1} \neq b_k \Rightarrow b_k R^C \phi(a_{k+1})$ が言えて, それと (4.4) 式より, $b_{k+1} \neq b_k \Rightarrow b_{k+1} = \phi(a_{k+1})$ が言える. よって, $k = i$ を代入することで, まず不変式 2' は証明された.

次に, 不変式 1' の証明をする. まずは, b_k と b_{k+1} の関係だが, $b_k R \phi(a_{k+1})$ の場合は, (4.3) 式より, $b_{k+1} = b_k$ となり, また $b_k R^C \phi(a_{k+1})$ の場合は, (4.4) 式より, $b_k R^C b_{k+1}$ となるので, 全ての k に対し, $b_k R^C_= b_{k+1}$ であることが言える. 次に $\phi(a_k)$ と b_k の関係だが, $b_{k-1} R^C \phi(a_k)$ のときは, (4.4) 式より, 容易に $\phi(a_k) = b_k$ が言え, また, $b_{k-1} R \phi(a_k)$ のときは, (4.3) 式より, $b_k = b_{k-1}$ なので $b_k R \phi(a_k)$ となり, R の反対称性から, 命題 6 より $\phi(a_k) R^C_= b_k$ が言える. よって, 以上より $\phi(a_k) R^C_= b_k$ と $b_k R^C_= b_{k+1}$ が言えたので, $R^C_=$ の推移性より, 不変式 1' が言える. ■

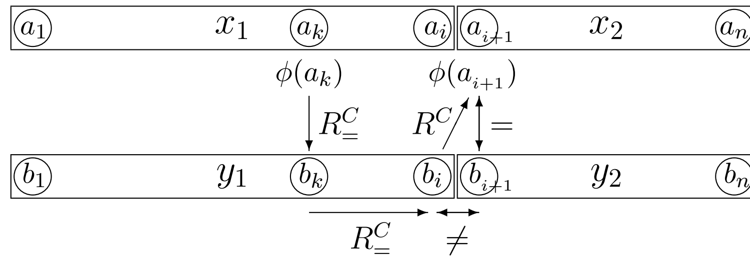

図 4.7　不変式

4.4.3　解法

先述の不変式が証明できたことにより, 関数 ϕ を掛けた場合についても, 同様に付随列 *auxList* x が, 意味を持つものとして定義できることが分かった. 次は, 最右要素に ψ を掛けたものをどのように付随列と比較させて関数化した最左最右関係問題を解いて行くかを考えていく.

まず, windowing 手法に関してだが, これは関数化しない場合と全く同じなので, ここでは省略する. 次に, *slide* 関数についてだが, これも関数化しない場合と殆んど同じであるが, ただ一つ違うところは, 比較の対象が a ではなく, $\psi(a)$ であることだ. すなわち, y_2 の最左要素と $\psi(a)$ とを比較して, その関係が R である場合は境目を一つずらしてまた同じことを繰り返し, R^C であるときはそこで止める. (次頁, 図 4.8 参照)

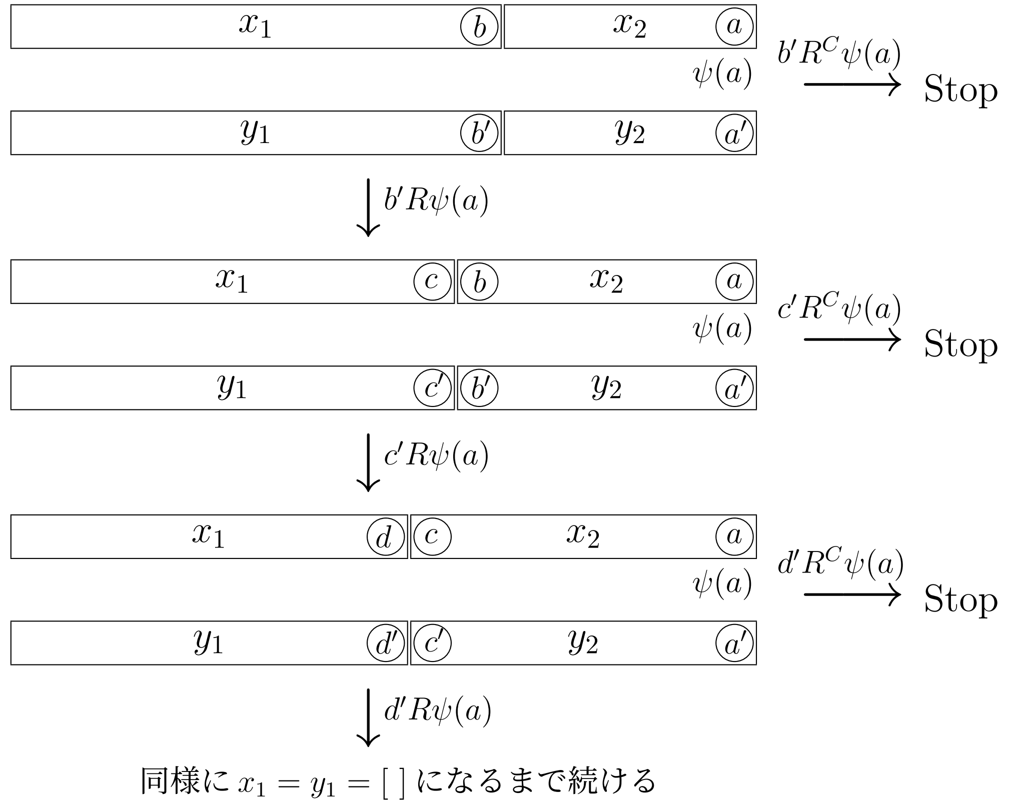

図 4.8　*slide* 関数

これらの方法を用いて導いたアルゴリズムを以下に示す.

$$
\mathit{lps}\ x = \mathbf{let}\ (x_1, x_2, y_1, y_2, s) = \mathit{lps'}\ x\ \mathbf{in}\ s
$$

$$
\begin{aligned}
&\mathit{lps'}\ [\,] = ([\,],[\,],[\,],[\,],[\,])\\
&\mathit{lps'}\ (x +\!\!+ [a]) =\\
&\quad \mathbf{let}\ \ (x_1, x_2, y_1, y_2, s) = \mathit{lps'}\ x\\
&\quad\quad a' = \mathbf{if}\ \ (\mathit{rightmost}\ y_2) R \phi(a)\ \ \mathbf{then}\ \ \mathit{rightmost}\ y_2\\
&\quad\quad a' = \mathbf{if}\ \ (\mathit{rightmost}\ y_2) R^C \phi(a)\ \ \mathbf{then}\ \ \phi(a)\\
&\quad\quad (x_1', x_2', y_1', y_2') = \mathit{window}\ (x_1, x_2, y_1, y_2, a, a')\\
&\quad\quad (x_1'', x_2'', y_1'', y_2'') = \mathit{slide}\ (x_1', x_2', y_1', y_2', \psi(a))\\
&\quad\quad s' = \mathit{longer}\ (s, x_2'')\\
&\quad \mathbf{in}\ (x_1'', x_2'', y_1'', y_2'', s')
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{window}\ (x_1, x_2, y_1, y_2, a, b) =\\
&\quad \mathbf{let}\ \ x_2' = x_2 +\!\!+ [a]\\
&\quad\quad y_2' = y_2 +\!\!+ [b]\\
&\quad\quad x_1'' = x_1 +\!\!+ [\mathit{leftmost}\ x_2']\\
&\quad\quad x_2'' = \mathit{tail}\ x_2'\\
&\quad\quad y_1'' = y_1 +\!\!+ [\mathit{leftmost}\ y_2']\\
&\quad\quad y_2'' = \mathit{tail}\ y_2'\\
&\quad \mathbf{in}\ (x_1'', x_2'', y_1'', y_2'')
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{slide}\ (x_1, x_2, y_1, y_2, a) = \\
&\quad \mathbf{if}\ \ y_1 \neq [\,] \wedge \mathit{rightmost}\ y_1 R a) \\
&\qquad \mathbf{then}\ \ \mathit{slide}\ (\mathit{init}\ x_1, [\mathit{rightmost}\ x_1] +\!\!+\, x_2, \mathit{init}\ y_1, [\mathit{rightmost}\ y_1] +\!\!+\, y_2) \\
&\qquad \mathbf{else}\ \ (x_1, x_2, y_1, y_2)
\end{aligned}
$$

以上のアルゴリズムの略証を図 4.9 を用いて, 以下に示す. そのためには, 先程と同様に, 要素 a の関数値 $\psi(a)$ に対して, 次の式が成り立つことを証明すればよい.

$$(\mathit{rightmost}\ y_1) R^C \psi(a) \wedge (\mathit{leftmost}\ y_2) R \psi(a) \implies x_2 +\!\!+\, [a] = \mathit{longest}\ (p \triangleleft \mathit{tails}\ (x +\!\!+\, [a]))$$

証明　まずは, $x_1 = [a_1, a_2, \ldots, a_i]$, $x_2 = [a_{i+1}, a_{i+2}, \ldots, a_n]$, $y_1 = [b_1, b_2, \ldots, b_i]$, $y_2 = [b_{i+1}, b_{i+2}, \ldots, b_n]$ とおく. そのとき, $b_i R^C \psi(a) \wedge b_{i+1} R \psi(a)$ より, 明らかに $b_i \neq b_{i+1}$ なので, 先程の不変式 2' より $\phi(a_{i+1}) = b_{i+1}$ が成り立ち, 故に $\phi(a_{i+1}) R \psi(a)$ が言える. また, 不変式 1' より, $\forall k \leq i : \phi(a_k) R^C_= b_i$ が言えて, $b_i R^C \psi(a)$ と $R^C_=$ の推移性より, $\forall k \leq i : \phi(a_k) R^C \psi(a)$ となる. よって, x_1 の全ての要素 c に対して $\phi(c) R^C \psi(a)$ であるのに対し, x_2 の最左要素 b に対しては $\phi(b) R \psi(c)$ であるので, $x_2 +\!\!+\, [a] = \mathit{longest}\ (p \triangleleft \mathit{tails}\ (x +\!\!+\, [a]))$ である. ■

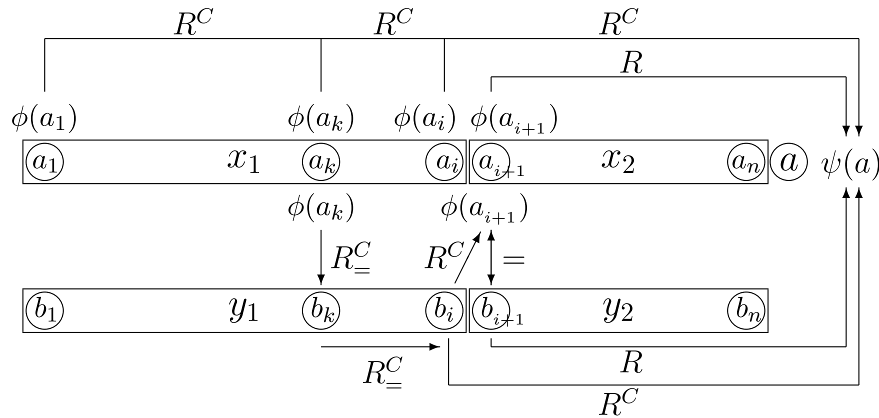

図 4.9　アルゴリズム

よって, 最左最右関係問題を関数化した場合でも, その関数が定数時間で計算できさえすれば, 単調性などの性質に関わらず, 先に述べた手法で最長部分列を求めることが出来ることが分かった. よって, 以下の定理が導けたことになる.

定理 1 関係 R が反対称かつ R^C が推移的であるとき, 定数時間で計算できる全ての実関数 ϕ, ψ に対して,

$$p(x) \ \equiv\ \phi(\mathit{leftmost}\ x) R \psi(\mathit{rightmost}\ x)$$

なる述語に対する最長部分列問題を解く線形な on-line アルゴリズムが存在する.

4.5　応用例

上記の定理 1 を応用した例を以下に二つ挙げる.

4.5.1　最左要素と最右要素の積に関する問題

まず, 応用例の一つ目として最左要素と最右要素の積に関する問題について考えてみたい. この最長部分列問題の述語は

$$p(x) \ \equiv\ (\mathit{leftmost}\ x)(\mathit{rightmost}\ x) \geq C \quad (\forall c \in x : c > 0)$$

で表される. 但し, ここで要素が正であるとしたのは, 両辺を負の数で割ると不等号の向きが逆転するからである. この問題に対して, 先ほどの拡張した最左最右関係問題を用いると, R が '$\leq$' で, $\phi(a) = C/a,\ \psi(a) = a$ となるので, アルゴリズムは以下のように書ける.

$$
\begin{array}{l}
\textit{lps}\ x = \textbf{let}\ (x_1, x_2, y_1, y_2, s) = \textit{lps'}\ x\ \textbf{in}\ s \\[6pt]
\textit{lps'}\ [\,] = ([\,], [\,], [\,], [\,], [\,]) \\
\textit{lps'}\ (x \mathbin{+\!\!+} [a]) = \\
\quad \textbf{let}\ \ (x_1, x_2, y_1, y_2, s) = \textit{lps'}\ x \\
\qquad\quad a' = \textbf{if}\ \ (\textit{rightmost}\ y_2) \leq C/a\ \ \textbf{then}\ \ \textit{rightmost}\ y_2 \\
\qquad\quad a' = \textbf{if}\ \ (\textit{rightmost}\ y_2) > C/a\ \ \textbf{then}\ \ C/a \\
\qquad\quad (x'_1, x'_2, y'_1, y'_2) = \textit{window}\ (x_1, x_2, y_1, y_2, a, a') \\
\qquad\quad (x''_1, x''_2, y''_1, y''_2) = \textit{slide}\ (x'_1, x'_2, y'_1, y'_2, a) \\
\qquad\quad s' = \textit{longer}\ (s, x''_2) \\
\quad \textbf{in}\ (x''_1, x''_2, y''_1, y''_2, s') \\[6pt]
\textit{window}\ (x_1, x_2, y_1, y_2, a, b) = \\
\quad \textbf{let}\ \ x'_2 = x_2 \mathbin{+\!\!+} [a] \\
\qquad\quad y'_2 = y_2 \mathbin{+\!\!+} [b] \\
\qquad\quad x''_1 = x_1 \mathbin{+\!\!+} [\textit{leftmost}\ x'_2] \\
\qquad\quad x''_2 = \textit{tail}\ x'_2 \\
\qquad\quad y''_1 = y_1 \mathbin{+\!\!+} [\textit{leftmost}\ y'_2] \\
\qquad\quad y''_2 = \textit{tail}\ y'_2 \\
\quad \textbf{in}\ (x''_1, x''_2, y''_1, y''_2) \\[6pt]
\textit{slide}\ (x_1, x_2, y_1, y_2, a) = \\
\quad \textbf{if}\ \ y_1 \neq [\,] \wedge \textit{rightmost}\ y_1 \leq a \\
\qquad \textbf{then}\ \ \textit{slide}\ (\textit{init}\ x_1, [\textit{rightmost}\ x_1] \mathbin{+\!\!+} x_2, \textit{init}\ y_1, [\textit{rightmost}\ y_1] \mathbin{+\!\!+} y_2) \\
\qquad \textbf{else}\ \ (x_1, x_2, y_1, y_2)
\end{array}
$$

4.5.2 部分列和と定数の比較に関する問題

二つ目の例として, 4.3 節でも述べた部分列和と定数の比較に関する問題について挙げてみよう. この問題は, 述語が

$$p(x)\ \equiv\ \textit{sum}\ x \leq C$$

で表されるもので, それを解くためには, 元の列を $x = [a_1, a_2, \ldots, a_n]$ とおいたとき,

$$u = [0, a_1, a_1 + a_2, ..., \sum_{i=1}^{n} a_i]$$

という, 要素がそれまでの和で表されるような列に写像して, その最長部分列を

$$p'(u)\ \equiv\ \textit{leftmost}\ u \geq \textit{rightmost}\ u - C$$

という述語に対して求めてやれば良かった. しかし, 一度列を写像して, それに対してもう一度最左最右関係問題を解くというやり方では, $O(n)$ にはなるものの, on-line にはならない. そこで, on-line アルゴリズムにするためには, x を読みとっていくのと同時に u を作成していく必要が出てくる.

そのためには, $x +\!\!+ [a]$ に対して, u にはどんな要素を付け加えればよいかという問題がまず浮かんでくる. しかし, これは $s_{n+1} = s_n + a_{n+1}$ という関係に注意すれば, u の新しい要素として, $(rightmost\ u) + a$ を付け加えてやればよいことがすぐに分かるであろう.

もう一つの問題は, u を u_1 と u_2 に分割する際に, x はどのように分割されていれば良いかということだ. まず,

$$u = [s_0, s_1, s_2, \ldots, s_n] \quad (s_0 = 0,\ s_i = \sum_{k=1}^{i} a_i)$$

とおく. (ここで, u の長さが x よりも 1 だけ長いのに注意されたい.) すると,

$$a_{i+1} + a_{i+2} + \cdots + a_j = s_j - s_i$$

となることは簡単に分かるであろう. よって, $u_2 = [s_i, s_{i+1}, \ldots, s_n]$ のとき, $x_2 = [a_{i+1}, a_{i+2}, \ldots, a_n]$ となっていれば良く, 故に,

$$\begin{aligned} length\ u_1 &= length\ x_1 \\ length\ u_2 &= length\ x_2 + 1 \end{aligned}$$

が常に成り立つように列を分割していけば良いことが分かるだろう. よって, これらのことに注意しつつ, アルゴリズムを構築していけばよい. そのアルゴリズムを以下に示す. なお, 実際に Haskell で動くプログラムも付録 D に付しておいたので, そちらも参照されたい.

$lps\ x = \mathbf{let}\ (x_1, x_2, u_1, u_2, v_1, v_2, s) = lps'\ x\ \mathbf{in}\ s$

$lps'\ [\,] = ([\,], [\,], [\,], [0], [\,], [0], [\,])$
$lps'\ (x +\!\!+ [a]) =$
$\quad \mathbf{let}\ \ (x_1, x_2, u_1, u_2, v_1, v_2, s) = lps'\ x$
$\quad\quad a' = (rightmost\ u_2) + a$
$\quad\quad a'' = \mathbf{if}\ \ (rightmost\ v_2) \geq a'\ \ \mathbf{then}\ \ rightmost\ v_2$
$\quad\quad a'' = \mathbf{if}\ \ (rightmost\ v_2) < a'\ \ \mathbf{then}\ \ a'$
$\quad\quad (x'_1, x'_2, u'_1, u'_2, v'_1, v'_2) = window\ (x'_1, x'_2, u'_1, u'_2, v'_1, v'_2, a, a', a'')$
$\quad\quad (x''_1, x''_2, u''_1, u''_2, v''_1, v''_2) = slide\ (x'_1, x'_2, u'_1, u'_2, v'_1, v'_2, a' - C)$
$\quad\quad s' = longer\ (s, x''_2)$
$\quad \mathbf{in}\ (x''_1, x''_2, u''_1, u''_2, v''_1, v''_2, s')$

$window\ (x_1, x_2, u_1, u_2, v_1, v_2, a, b, c) =$
$\quad \mathbf{let}\ \ x'_2 = x_2 +\!\!+ [a]$
$\quad\quad u'_2 = u_2 +\!\!+ [b]$
$\quad\quad v'_2 = v_2 +\!\!+ [c]$
$\quad\quad x''_1 = x_1 +\!\!+ [leftmost\ x'_2]$
$\quad\quad x''_2 = tail\ x'_2$
$\quad\quad u''_1 = u_1 +\!\!+ [leftmost\ u'_2]$
$\quad\quad u''_2 = tail\ u'_2$
$\quad\quad v''_1 = v_1 +\!\!+ [leftmost\ v'_2]$
$\quad\quad v''_2 = tail\ v'_2$
$\quad \mathbf{in}\ (x''_1, x''_2, u''_1, u''_2, v''_1, v''_2)$

$$
\begin{array}{l}
\mathit{slide}\ (x_1, x_2, u_1, u_2, v_1, v_2, b) = \\
\quad \textbf{if}\ \ v_1 \neq [\,]\ \wedge\ \mathit{rightmost}\ v_1 \geq b \\
\qquad \textbf{then}\ \ \mathit{slide}\ (\mathit{init}\ x_1, [\mathit{rightmost}\ x_1] +\!\!+\, x_2, \mathit{init}\ u_1, [\mathit{rightmost}\ u_1] +\!\!+\, u_2, \\
\qquad\qquad\qquad \mathit{init}\ v_1, [\mathit{rightmost}\ v_1] +\!\!+\, v_2) \\
\qquad \textbf{else}\ \ (x_1, x_2, u_1, u_2, v_1, v_2)
\end{array}
$$

第 5 章

計算機上でのプログラミング

この章では, Haskell という関数型言語を用いて, 実際のプログラムにおける問題点について考える. 実際, プログラムを本当の意味で $O(n)$ にするのは結構難しいことであって, 定数オーダーで出来ると思ってた演算が $O(n)$ かかってしまうということがしばしばある. そこらへんについて, [2], [4] を参考にしつつ, この章では折に触れて考えていきたい.

5.1 Haskell の特徴

Haskell についてはたくさんの能力があるが, 本論文中で用いるべき要素はその一部分に過ぎないので, Haskell の細かいことには触れず, 大まかな特徴だけを述べたいと思う.

- 左開き

  今まで本論文中では, 列に対して左右どちらからも, 定数時間で要素の出し入れができるものとしてアルゴリズムを説明してきた. しかし, 実際に Haskell で扱うリストは要素を左から取り出したり, 左に付け加えたりするような左開きの構造になっている. 勿論, リストの右からも要素の出し入れは出来るが, 左開きの構造上, その作業に $O(n)$ もの時間がかかってしまう.

- 関数型言語

  Haskell は関数型言語であるため, 代入操作などは存在しない. Haskell のプログラムにおいては, 代入操作は別の変数との等号関係でもって表現することになる. 例えば, $x \leftarrow x +\!\!+ [a]$ という代入操作をしたいときは, $x' = x +\!\!+ [a]$ としておき, 結果として x' を用いるという形を取る.

- 関数の再帰的定義

  Haskell においても, 論文中で示したアルゴリズムと同様, 再帰的な関数定義ができる. ただし, 今まで本論文中では, $lps\ x$ の返す値を用いて, $lps\ (x +\!\!+ [a])$ を定義していくといった, 列に対して要素を右から付け加えていくというやり方でアルゴリズムを説明してきた. しかし, Haskell は左開きであるため, $lps\ (a : x) = \mathbf{let}\ \ldots$ のように, 左から付け加える形でしか, 関数の再帰的定義はできない.

5.2 最左最右関係問題のプログラム

まずは, 左右いずれにも開いてると考えて関数定義した Haskell のプログラムを付録 A に示す. 但し, 一つのリストを用いた再帰的定義に関しては, 右開きの定義が出来ないので, このプログラムでは

やむなく左右を逆にして書いている. よって, *leftmost* と *rightmost, init* と *tail* が逆になっているのに注意されたい.

5.3 改善法

付録 A に, 第 4 章で述べた最左最右関係問題のアルゴリズムを Haskell で書いたプログラムを記した. しかし, これは本当は $O(n)$ を実現していない. 何故なら, 論文中で定数時間で計算できるとしてきたのにもかかわらず, Haskell では $O(n)$ の時間がかかってしまう演算が存在するからだ. それは, *rightmost, init, length,* および列の右に要素を付け加える演算である. これらの操作を何とか定数時間で行なえないかということについて, この節で述べる.

5.3.1 対リスト構造を用いた列の表記法

まず, 列を表記するのにあたって, 一見一つのリストを用いることが自然であるように思われるかもしれない. 実際, そちらの方がプログラムも見やすくなるし, どういう演算を行なっているかもよく分かるであろう. しかし, これには重大な欠点がある. それは先にも述べた通り, リストは左側にしか開いてないということだ. そのため, 列の最右要素に関する演算が定数時間で出来ないことになる. 特に最左最右関係問題では, 列の最右要素を取り出すという演算がループ内で何度も行なわれるため, 最右要素を定数時間で取り出せるようにしたい. ならば, どうすればよいか. それを解決するために, 一つの列を表記するのに二つのリストを使うという対リスト構造を用いることにした. (two-end-list [2] 参照)

例えば, 列 $[a_1, a_2, a_3, a_4, a_5]$ を表記するのに, $([a_1, a_2], [a_5, a_4, a_3])$ といったリストのペアを用いる. ここで, リストの二番目の要素は順番が逆になっていることに注意されたい. このリストのペアを一つの列とみなして計算すれば, 列の最左要素を取り出すときは, 左側のリストの最左要素を, 列の最右要素を取り出すときは, 右側のリストの最左要素を取り出せば, それにかかる時間は定数時間で済む.

それでは, リストのペアのうち片方が空になった場合どうするか. これに対しては *lpush, rpush* という関数を定めておいて, それを用いることで対処する. *lpush* の方は, 左側のリストに要素があるときは何もせず, 左側のリストが空になったときだけ, 右側のリストをリバースして全て左側に移すという作業を行ない, *rpush* はそれの左右が逆のものである. この両関数はアルゴリズムの性質によって, どこに使うかなどが変わってくるが, 基本的に右から要素をどんどん受けとるタイプの関数では, *lpush* しか使わくてよいことが多い. なおリバースして移すという作業には $O(n)$ の時間がかかるが, 多くの場合ペアのいずれのリストにも要素が存在しているため, このリバースという作業が全体のオーダーに及ぼす影響は殆んどないと言える.

だが, これだけではまだ十分とは言えない. 列の長さを求めるコマンドもループの中に入っているからだ. これを解決するには, 列を表記する際に長さも記憶しておく必要がある. そうすれば, 毎回長さを計算せずとも, 記憶を取り出せば定数時間でその列の長さも分かるという訳である.

以上のことを踏まえた上で, 列を Haskell 上では以下のように表記することにした.

$$[a_1, a_2, \ldots, a_n] \;\equiv\; ([a_1, a_2, \ldots, a_i], [a_n, a_{n-1}, \ldots a_{i+1}], n)$$

5.3.2 対リスト構造に用いる関数の定義

付録 A で *leftmost* や *rightmost* の定義をしているが, 勿論これは対リスト構造の列表記には使えない. *init* や *tail* に関しても同じである. よって対リスト構造に対する関数を定めてやる必要がある. 細かい定義は付録 B のプログラムの内容を見て頂きたい. 基本的には, *init'* や *rightmost'* のように右上にプライムの付いてる関数は列の対リスト構造に対する関数だと思って頂きたい.

5.4 結果

改善前のプログラムを付録 A に, 改善後のプログラムを付録 C に付しておいた. 両者を比較した場合, 一見改善前のプログラムの方が他の関数定義も少なく, 列の構造も単純なので, 比較的すっきりしてるかのように思われる. しかし, 実際に両方のプログラムを実行してリダクション数を調べたら, その違いは歴然と出た. 以下の表とグラフでは, ある数列を n 回連結させたものに対して, lps 関数を実行し, その n と, リダクションおよびセルの値とを, 比較した結果を表している. それを見ると, 改善前はリダクション, セル共に二乗のオーダーで増えていのに対し, 改善後は線形オーダーに押えられているのが分かるだろう.

表 5.1　リダクションとセルの変化

n	改善前		改善後	
	reductions	cells	reductions	cells
1	2663	4374	1941	5064
2	9769	14582	3896	10078
3	21352	30665	5919	15252
4	37405	52598	7925	20379
5	57928	80381	9931	25506
6	82921	114014	11937	30633
7	112384	153497	13943	35760
8	146317	198830	15949	40887
9	184720	250013	17955	46014
10	227448	306623	19961	51141

図 5.1　リダクションとセルの変化

第 6 章

結論

最長部分列問題にテーマを絞って, 半年間研究をしてきたのだが, まず $O(n)$ で解けるか解けないかという分岐点の一つとして, 述語に用いられる関係 R の性質が関わっていると思われる. その中でも特に推移性や反対称性は $O(n)$ で解けるための重要なファクターであることが多い. 逆に, 論文 [1] の中でも述べられていたが, '=' などの同値関係では $O(n)$ の解法が存在しない場合が多いようだ. (最左最右関係問題においても, R が '=' のときには線形解法が存在しないことが分かっている.)

まず, 第 3 章で述べた述語の閉属性に関しての解法だが, *overlap-closed* だけしか成り立たない述語についての $O(n)$ の解法については, まだ本論文では解決法を示していないので, これを解くことが今後の課題の一つになるだろう.

また, 第 4 章で述べた最左最右関係問題についての解法およびその応用については, 最左要素および最右要素を関数化して比較することができるという結論に至ったことは当研究の収穫であったし, 他の最長部分列問題に対しても広く応用が効くものと思われる. 例えば, 最左要素と最右要素の和や積に関する関係なども, $p(x) = \phi(\mathit{leftmost}\ x)R\psi(\mathit{rightmost}\ x)$ で書けさえすれば, $O(n)$ で解くことが出来るということになる.

なお, 今後の課題としては先に述べた *overlap-closed* のときの解法や, その他諸々の閉属性で括られない述語に対する最長部分列問題の解法を見つけていくこと, または列の要素が数以外のデータの場合に対して応用していくことなどが挙げられる.

付録 A

最左最右関係問題の Haskell プログラム (改善前)

ここでは, 列を表すのに一つのリストだけを用いた場合の最左最右関係問題を解く Haskell のプログラムを紹介する. なお, このプログラムは実際に Haskell 上では, $O(n)$ にはならないので, 注意されたい.

```
lps :: [Int] -> [Int]
lps x = let (x1,x2,y1,y2,s) = lps' x in s

lps' :: [Int] -> ([Int],[Int],[Int],[Int],[Int])
lps' [] = ([],[],[],[],[])
lps' (a:x) =
  let (x1,x2,y1,y2,s) = lps' x
      a'= if leftmost y2 <= a then leftmost y2 else a
      (x1',x2',y1',y2') = window (x1,x2,y1,y2,a,a')
      (x1'',x2'',y1'',y2'') = slide (x1',x2',y1',y2',a)
      s' = longer (s,x2'')
  in (x1'',x2'',y1'',y2'',s')

window :: ([Int],[Int],[Int],[Int],Int,Int) -> ([Int],[Int],[Int],[Int])
window (x1,x2,y1,y2,a,b) =
  let x2' = a:x2
      y2' = b:y2
      x1'' = (rightmost x2'):x1
      x2'' = init x2'
      y1'' = (rightmost y2'):y1
      y2'' = init y2'
  in (x1'',x2'',y1'',y2'')

slide :: ([Int],[Int],[Int],[Int],Int) -> ([Int],[Int],[Int],[Int])
slide (x1,x2,y1,y2,a) =
  if (y1 /= [])&&(leftmost y1 <= a)
    then slide (tail x1,x2++[leftmost x1],tail y1,y2++[leftmost y1],a)
    else (x1,x2,y1,y2)
```

```
leftmost :: [Int] -> Int
leftmost [] = 9999
leftmost (a:x) = a

rightmost :: [Int] -> Int
rightmost [] = 9999
rightmost [a] = a
rightmost (a:x) = rightmost x

longer :: ([Int],[Int]) -> [Int]
longer (x,y) = if length x < length y then y else x
```

付録 B

対リスト構造に用いる関数

対リスト構造に用いる関数をここでまとめて定義しておく. なお, omega の値はプログラムによって異なるため, プログラムの本体の方で定義する. また, 本文で説明の無かった関数として, lattach', rattach', remake' があるが, lattach' と rattach' はそれぞれ, 要素を列の左および右に取り込む関数で, remake' は対リスト構造の列を一つのリストに戻す関数である.

```
leftmost :: [Int] -> Int
leftmost [] = omega
leftmost (a:x) = a

rightmost :: [Int] -> Int
rightmost [] = omega
rightmost [a] = a
rightmost (a:x) = rightmost x

leftmost' :: ([Int],[Int],Int) -> Int
leftmost' ([],y,len) = rightmost y
leftmost' (a:x,y,len) = a

rightmost' :: ([Int],[Int],Int) -> Int
rightmost' (x,[],len) = rightmost x
rightmost' (x,b:y,len) = b

lattach' :: (Int,([Int],[Int],Int)) -> ([Int],[Int],Int)
lattach' (a,(x,y,len)) = (a:x,y,len+1)

rattach' :: (([Int],[Int],Int),Int) -> ([Int],[Int],Int)
rattach' ((x,y,len),a) = (x,a:y,len+1)

rpush' :: ([Int],[Int],Int) -> ([Int],[Int],Int)
rpush' (x,y,len) = if y == [] then ([],reverse x,len) else (x,y,len)

lpush' :: ([Int],[Int],Int) -> ([Int],[Int],Int)
lpush' (x,y,len) = if x == [] then (reverse y,[],len) else (x,y,len)
```

```
length' :: ([Int],[Int],Int) -> Int
length' (x,y,len) = len

longer' :: (([Int],[Int],Int),([Int],[Int],Int)) -> ([Int],[Int],Int)
longer' (x,y) = if length' x < length' y then y else x

tail' :: ([Int],[Int],Int) -> ([Int],[Int],Int)
tail' ([],y,len) = ([],init y,len-1)
tail' (x,y,len) = (tail x,y,len-1)

init' :: ([Int],[Int],Int) -> ([Int],[Int],Int)
init' (x,[],len) = (init x,[],len-1)
init' (x,y,len) = (x,tail y,len-1)

remake' :: ([Int],[Int],Int) -> [Int]
remake' (x,y,len) = x ++ reverse y
```

付録 C

最左最右関係問題の Haskell プログラム (改善後)

対リスト構造を用いた最左最右関係問題の Haskell プログラムを以下に示す. なお, 基本操作を行なう関数は付録 B に記してあるので, それらと併せて実行されたい.

```
lps :: [Int] -> [Int]
lps x = let (x1,x2,y1,y2,s) = lps' ([],reverse x,length x) in remake' s

omega :: Int
omega = 9999

lps' :: ([Int],[Int],Int)
     -> (([Int],[Int],Int),([Int],[Int],Int),
         ([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int))
lps' ([],[],0) = (([],[],0),([],[],0),([],[],0),([],[],0),([],[],0))
lps' ([],a:x,len) =
  let (x1,x2,y1,y2,s) = lps' ([],x,len-1)
      a' = if rightmost' y2 <= a then rightmost' y2 else a
      (x1',x2',y1',y2') = window' (x1,x2,y1,y2,a,a')
      (x1'',x2'',y1'',y2'') = slide' (x1',x2',y1',y2',a)
      s' = longer' (s,x2'')
  in (x1'',x2'',y1'',y2'',s')

window' :: (([Int],[Int],Int),([Int],[Int],Int),
            ([Int],[Int],Int),([Int],[Int],Int),Int,Int)
        -> (([Int],[Int],Int),([Int],[Int],Int),
            ([Int],[Int],Int),([Int],[Int],Int))
window' (x1,x2,y1,y2,a,b) =
  let x2' = rattach' (x2,a)
      y2' = rattach' (y2,b)
      x2'' = lpush' x2'
      y2'' = lpush' y2'
      x1''' = rattach' (x1,(leftmost' x2''))
      x2''' = tail' x2''
      y1''' = rattach' (y1,(leftmost' y2''))
```

```
      y2''' = tail' y2''
  in (x1''',x2''',y1''',y2''')

slide' :: (([Int],[Int],Int),([Int],[Int],Int),
           ([Int],[Int],Int),([Int],[Int],Int),Int)
       -> (([Int],[Int],Int),([Int],[Int],Int),
           ([Int],[Int],Int),([Int],[Int],Int))
slide' (x1,x2,y1,y2,a) =
  if (y1 /= ([],[],0))&&(rightmost' y1 <= a)
    then slide' (init' x1,lattach' (rightmost' x1,x2),
                 init' y1,lattach' (rightmost' y1,y2),a)
    else (x1,x2,y1,y2)
```

付録 D

部分列和と定数の比較に関する問題の Haskell プログラム

$p(x) \equiv sum\ x \leq C$ に対する最長部分列問題を Haskell で解くプログラムを以下に示す．なお, このプログラムも対リスト構造を用いているため, 付録 B に記してある基本操作を行なう関数と併せて実行されたい.

```
lps :: [Int] -> [Int]
lps x = let (x1,x2,u1,u2,v1,v2,s) = lps' ([],reverse x,length x) in remake' s

omega :: Int
omega = -9999

c :: Int
c = 20

lps' :: ([Int],[Int],Int)
     -> (([Int],[Int],Int),([Int],[Int],Int),
         ([Int],[Int],Int),([Int],[Int],Int),
         ([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int))
lps' ([],[],0) = (([],[],0),([],[],0),([],[],0),([],[0],0),
                  ([],[],0),([],[0],0),([],[],0))
lps' ([],a:x,len) =
  let (x1,x2,u1,u2,v1,v2,s) = lps' ([],x,len-1)
      a'= rightmost' u2 + a
      a'' = if rightmost' v2 >= a' then rightmost' v2 else a'
      (x1',x2',u1',u2',v1',v2') = window' (x1,x2,u1,u2,v1,v2,a,a',a'')
      (x1'',x2'',u1'',u2'',v1'',v2'') = slide' (x1',x2',u1',u2',v1',v2',a'-c)
      s' = longer' (s,x2'')
  in (x1'',x2'',u1'',u2'',v1'',v2'',s')

window' :: (([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int),
            ([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int),Int,Int,Int)
        -> (([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int),
            ([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int))
window' (x1,x2,u1,u2,v1,v2,a,b,c) =
```

```
  let x2' = rattach' (x2,a)
      u2' = rattach' (u2,b)
      v2' = rattach' (v2,c)
      x2'' = lpush' x2'
      u2'' = lpush' u2'
      v2'' = lpush' v2'
      x1''' = rattach' (x1,(leftmost' x2''))
      x2''' = tail' x2''
      u1''' = rattach' (u1,(leftmost' u2''))
      u2''' = tail' u2''
      v1''' = rattach' (v1,(leftmost' v2''))
      v2''' = tail' v2''
  in (x1''',x2''',u1''',u2''',v1''',v2''')

slide' :: (([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int),
           ([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int),Int)
       -> (([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int),
           ([Int],[Int],Int),([Int],[Int],Int),([Int],[Int],Int))
slide' (x1,x2,u1,u2,v1,v2,a) =
  if (v1 /= ([],[],0))&&(rightmost' v1 >= a)
    then slide' (init' x1,lattach' (rightmost' x1,x2),
                 init' u1,lattach' (rightmost' u1,u2),
                 init' v1,lattach' (rightmost' v1,v2),a)
    else (x1,x2,u1,u2,v1,v2)
```

謝辞

まずは, この論文を書くにあたって, 一生懸命ご指導下さった担当教官の胡振江講師に心から感謝します. 先生には, 最長部分列問題と言うテーマを与えて下さったばかりでなく, 研究中に分からないところがあったときなど, 快く懇切丁寧に質問に答えて下さったり, 順調に研究が進んでいるか常に気にかけて下さったりと, 多岐に渡って非常にお世話になりました.

また, 武市正人教授, 岩崎英哉助教授, 尾上能之助手にもミーティングの際に, 研究内容に関して, 一緒になって考えて下さったり, 内容についてのご指摘を下さったりと, 様々なご尽力の程を心よりお礼申し上げます.

また, 分からないことについて色々と教えて下さった研究室の諸先輩方にも感謝の意を尽くせません.

参考文献

[1] H. Zantema, *Longest Segment Problems*, Science of Computer Programming 18, 1992 39-66.

[2] Chris Okazaki, *Purely Functional Data Structures*, Cambridge University Press, 1998.

[3] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Mining optimized association rules for numeric attributes. In *Proceeding of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1996.

[4] 武市正人, “プログラミング言語,” 岩波書店, 1994.