Longest segment problems

H. Zantema

RUU-CS-90-28
August 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Longest segment problems

H. Zantema

Technical Report RUU-CS-90-28
August 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

Longest segment problems

H. Zantema
Department of Computer Science
University of Utrecht
P.O. box 80.089
3508 TB Utrecht
The Netherlands

Abstract

The following problem is considered:

> Given a predicate p on strings. Determine the longest segment of a given string that satisfies p.

This paper is an investigation of algorithms solving this problem for various predicates. The predicates considered are expressed in simple functions like the size, the minimum, the maximum, the leftmost and the rightmost element of the segment. The algorithms are linear in the length of the string.

1 Introduction

Consider the following problem:

> Given a predicate p on strings. Find an algorithm to determine the longest segment (= consecutive substring) of a given string of length n that satisfies p. The algorithm has to be linear in n.

A problem of this shape we call a *segment problem.* If nothing is known about p the only way to solve this problem is summing up all segments of the string, determine for each segment whether p holds for it or not, and keep track of the longest segment for which p holds. Since the number of segments is about $n^2/2$, this algorithm requires $O(n^2)$ computations of p, and is clearly not linear. However, if p has a particular shape, or has some nice properties, then often a strong optimization can be made, resulting in a linear algorithm. This paper is an investigation of possible optimizations of this type, together with the required properties. For each set of required properties a number of examples is given, yielding some dozens of solutions of segment problems throughout the paper.

One of the simplest examples is the *longest plateau problem:* given a string, find the longest segment of which all of the elements are equal. It is a standard undergraduate programming exercise to find a linear solution for this problem, see e.g. [2], section 16.3. However, most of our examples are less simple. A whole range of segment problems is solved in [3]. In a different way all of them are solved in this paper too, but not the other way around. An example of a non-trivial problem of which we present a solution is the following: given a string of integers, find in linear time the longest segment of which the length is smaller than the sum of the maximum and the minimum of that segment.

One can wonder why these segment problems are interesting. On the one hand one may argue that a solution of a non-trivial segment problem is interesting as an algorithm itself. On the other hand most segment problems have simple formulations, they have short solutions that only need very simple data structures, while these solutions are difficult to find. This makes them very suitable as testcases for programming methodologies. For example, various problems discussed by M. Rem in his regular column *Small Program Exercises* in the journal *Science of Computer Programming* are instances of segment problems.

In this paper we do not fix to one particular programming methodology. However, in developing our programs we follow

- some of the notation of the Bird–Meertens formalism;
- the driving force of choosing invariants and obtaining correctness proofs for free, as in the method of programming proposed by Dijkstra and Gries;
- the generalization paradigm from mathematics: after finding a proof or derivation of a particular case, examine which assumptions are essential in each of the steps, and present it in the most general case.

In section 2 we present the notation we use in this paper and define initial segments, tail segments and general segments. In section 3 we show how for some very simple predicates the corresponding segment problems have no solutions of complexity less than $O(n \log n)$. In the next section some basic properties of programs like 'on-line' and 'real-time' are discussed, and some basic properties of predicates, like 'prefix-closed'. In section 5 the first algorithms appear. The algorithms are still rather simple. On the one hand they are included for the sake of completeness: they are the starting point for the classification of segment problems presented in this paper. On the other hand they are interesting in the way they are parametrized and therefore general applicable.

The most interesting part of this paper is section 6: the application of partitions. Except for the notation introduced in section 2 this section hardly depends on the first part of the paper. Lots of non-trivial solutions of segment problems are derived here, for which the shape of the predicate does not suggest to apply partitions at all. The main result is proposition 11, which provides the key for many solutions, and is also applicable for other problems than segment problems. Roughly speaking, the approach can be described as follows: problems that lack some required monotonicity can be solved by introducing an additional data structure in which this monotonicity can be forced. Some of this approach also can be found in [3] and [4]. One difference is that we tend to on-line algorithms and avoid preprocessing.

2 Notation

In a lot of presentations on segments, the problem is expressed in terms of arrays. However, in the formulation of a segment problem no indexing occurs, and we prefer to avoid introducing indexing. This can be done by describing some basic string operations; throughout the paper we shall only refer to these basic string operations.

Although we do not follow the way of program development as suggested in [1], we borrow some of the notation from that paper. The notational convention is as follows:

elements	$a, b, c, \ldots$
strings of elements	x, y, z, s, t, u, v, w
strings of strings of elements	xs, ys, zs
sets of strings of elements	X, Y, Z
predicates on strings	p, q, r

When no confusion is possible, parentheses of function application are often omitted, so fx means $f(x)$. The string of n elements $a_1, a_2, \ldots a_n$ respectively is denoted by $[a_1, a_2, \ldots a_n]$; concatenation of strings is denoted by $+\!\!+$:

$$[a_1, a_2, \ldots a_n] +\!\!+ [b_1, b_2, \ldots b_m] \;=\; [a_1, a_2, \ldots a_n, b_1, b_2, \ldots b_m].$$

The binary operators $\ll$ and $\gg$ are universally defined:

$$a \ll b = a \quad \text{and} \quad a \gg b = b.$$

If a total order $\leq$ has been defined on the elements, then a maximum operator $\uparrow$ and a minimum operator $\downarrow$ are defined:

$$a \uparrow b = \begin{cases} a & \text{if } b \leq a \\ b & \text{if } a \leq b \end{cases} \qquad a \downarrow b = \begin{cases} a & \text{if } a \leq b \\ b & \text{if } b \leq a \end{cases}$$

The size function on strings is denoted by $\#$:

$$\#[a_1, a_2, \ldots, a_n] \;=\; n.$$

For any associative binary operator $\oplus$ we can define the operator $\oplus/$ on non-empty strings:

$$\oplus/[a_1, a_2, \ldots, a_n] \;=\; a_1 \oplus a_2 \oplus \cdots \oplus a_n.$$

For example, $\ll/x$ denotes the leftmost element of the string x, and $\uparrow/x$ denotes the value of the greatest element of the string x. If the operator $\oplus$ is also commutative and idempotent, then $\oplus/$ is also defined on sets instead of strings.

For any function f on elements we can define the function f^* on strings:

$$f^*[a_1, a_2, \ldots a_n] \;=\; [fa_1, fa_2, \cdots, fa_n].$$

To be able to express segment problems in this notation, we need the notion of the longest segment of a set of segments; we choose the notation $\uparrow_\#$ for the longer of two segments, and the notation $\uparrow_\#/$ for the longest of a string of segments or a set of segments. One question immediately arises: what to do if this is not unique? Do we want to deliver all of them, or only one of them, and in the latter case, which one of them? This question has caused a lot of troubles; we prefer to consider it as an implementation detail. We only assume the availability of a binary operator $\uparrow_\#$, and an operator $\uparrow_\#/$ on non-empty sets of strings in such a way that for all non-empty sets of strings X and Y we have

$$\uparrow_\#/(X \cup Y) \;=\; (\uparrow_\#/X) \uparrow_\# (\uparrow_\#/Y). \tag{1}$$

The operator $\uparrow_\#$ is supposed to be computable in constant time.

Next we define the *filter*. Let p be a predicate on some type of elements, then $p\triangleleft$ (pronounced as p filter) is defined on sets of that type of elements as follows:

$$p \triangleleft X \;=\; \{x \in X \mid px\}.$$

An immediate consequence of this definition is

$$p \triangleleft (X \cup Y) \;=\; (p \triangleleft X) \cup (p \triangleleft Y) \tag{2}$$

for all sets of strings X and Y.

If *segs* x denotes the set of segments of the string x, we can describe segment problems in this notation: given a predicate p find an algorithm that computes

$$\uparrow_\#/(p \triangleleft \mathit{segs}\, x)$$

for any given string x. In this description x occurs as a dummy; omitting parentheses we can also ask for the computation of the function $\uparrow_{\#}/p \triangleleft segs$. We still have to give a formal description of *segs*. One way to do this is by introducing tails and inits:

$$tail\,[a_1, a_2, \ldots a_n] \;=\; [a_2, a_3, \ldots a_n],$$
$$init\,[a_1, a_2, \ldots a_n] \;=\; [a_1, a_2, \ldots a_{n-1}].$$

For example, for any non-empty string x we have

$$[\ll/x] +\!\!+ (tail\,x) \;=\; x \;=\; (init\,x) +\!\!+ [\gg/x].$$

The set consisting of $x, tail\,x, tail\,(tail\,x)$ and so on will be written as $tails\,x$, and similarly $inits\,x$, so

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

Now we can give an inductive definition of *segs*:

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

Applying equations (1) and (2) to this definition, we obtain

$$\uparrow_{\#}/p \triangleleft segs\,(x +\!\!+ [a]) \;=\; (\uparrow_{\#}/p \triangleleft segs\,x)\;\uparrow_{\#}\;(\uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a])). \qquad (3)$$

For many predicates p this is the property we need for the derivation of an algorithm for computing $\uparrow_{\#}/p \triangleleft segs$. In section 6.2 however, we shall need a generalization.

We prefer to present our algorithms in an imperative style:

do *not eof* $\rightarrow$ *read(a)*
; ……
od

If z is the total string to be considered, x is the part of z already read, and y is the part of z still to be read, we have as an invariant

$$z \;=\; x +\!\!+ y.$$

Here *eof* and *read(a)* can be considered as abbreviations of

$$y = [\,]$$

and

$$
\begin{array}{ll}
 & a := \ll / y \\
; & x := x +\!\!+ [a] \\
; & y := tail\, y,
\end{array}
$$

respectively. The initialization

$$
\begin{array}{ll}
 & x := [\,] \\
; & y := z
\end{array}
$$

is left implicit.

3 Lower bounds on the complexity

This paper is on deriving and presenting linear time algorithms for segment problems. One can wonder whether linear time algorithms always exist for simply shaped predicates. The answer is no: there are very simply shaped predicates for which we can prove that no linear algorithm exists. Of course the meaning of this statement depends on the complexity model. We follow the model in which the number of element comparisons is counted. More precisely, a total order on the elements is assumed, and one basic step is the following: compare two elements a and b, then the result of the comparison is either $a > b$, or $a = b$, or $a < b$.

An interesting example is achieved by taking the predicate p defined by

$$p x \;\equiv\; \ll / x \;=\; \gg / x.$$

From this definition of p it is immediate that

$$\#(\uparrow_{\#} / p \triangleleft segs\, x) \begin{cases} = 1 & \text{if all elements of } x \text{ are distinct} \\ > 1 & \text{if they are not.} \end{cases}$$

So if there is a linear algorithm to compute $\uparrow_{\#} / p \triangleleft segs$, then there is also a linear algorithm checking whether all elements of a given string are distinct or not. In the literature this problem is called the 'element uniqueness problem', and it has been proven that it takes at least $O(n \log n)$ steps, where n is the length of the string, see [5]. (In [5] any linear test is allowed, but the difference with our complexity model is not essential.)

An easier argument of this result has been given by D. Gordon, which can be sketched as follows. Any element uniqueness checking algorithm can be transformed into a sorting algorithm using only an extra constant time for each comparison; since sorting takes at least $O(n \log n)$ steps, the same holds for element uniqueness.

If indexing in an array of which the index type equals the type of the string elements is also considered as a basic step, then a linear element uniqueness algorithm can be given. In our model, however, such indexing is not a basic step.

Variations of this method give more proofs of the nonexistence of linear solutions of segment problems, in particular if the predicate consists of an equality. For example, if

$$px \equiv \ll/x + \gg/x = C$$

for some constant C, or if

$$px \equiv +/x = C$$

for some constant C, then one can prove that there is no linear algorithm computing $\uparrow_{\#}/p \triangleleft segs$, provided that both positive and negative numbers are allowed to occur in the string.

4 Basic properties

4.1 On-line and real-time

In this paper we consider algorithms that inspect the elements of a given string from left to right, and each element is inspected only once. Such an algorithm has the following shape:

$$\begin{array}{ll} & value := initvalue \\ ; & \textbf{do } not\ eof \rightarrow read(a) \\ & \qquad\qquad ; \quad value := \phi(value, a) \\ & \textbf{od} \\ ; & result := \pi(value). \end{array}$$

The functions ϕ and π are supposed not to depend on the string that is read. If the function π is computable in constant time, then this algorithm is called *on-line.* If the functions ϕ and π are both computable in constant time, then this algorithm is called *real-time.*

In other words, an algorithm for the computation of $f(z)$ is called on-line if for each $x \in inits\, z$ the result $f(x)$ is available before elements in the string behind x have been inspected. An algorithm is called *real-time* if it is on-line and the time between any two consecutive inspections is majorated by a constant.

For example, an algorithm containing some preprocessing is not on-line, since in such an algorithm no result on any init is available before all elements have been inspected.

Clearly a real-time algorithm is always linear. However, real-time is not the same as linear on-line. In the latter case the average time between any two consecutive inspections is constant (or the cost is amortized constant as it is sometimes called in the literature), which is a weaker requirement than in the case of real-time.

In [1] the infix notation $value \oplus a$ is chosen instead of $\phi(value, a)$; the on-line algorithm is then written as $\pi \oplus \not\leftarrow s$, and is called a *directed reduction.*

Back to segment problems. From the definition of $\triangleleft$ it is immediate that

$$(p \vee q) \triangleleft X \;=\; (p \triangleleft X) \cup (q \triangleleft X)$$

for all predicates p and q and sets of strings X. Combining this with equation (1) we obtain

$$\uparrow_{\#}/(p \vee q) \triangleleft segs\, x \;=\; (\uparrow_{\#}/p \triangleleft segs\, x) \uparrow_{\#} (\uparrow_{\#}/q \triangleleft segs\, x).$$

Given two linear algorithms computing $\uparrow_{\#}/p \triangleleft segs$ and $\uparrow_{\#}/q \triangleleft segs$, we can construct a linear algorithm computing $\uparrow_{\#}/(p \vee q) \triangleleft segs$ by joining them together and the $\uparrow_{\#}$ of both results. If the join is inside the loop, the same holds if we replace the word 'linear' by 'on-line', or by 'real-time'.

Given two algorithms computing $\uparrow_{\#}/p \triangleleft segs$ and $\uparrow_{\#}/q \triangleleft segs$, one can wonder if there is a construction giving a similar algorithm computing $\uparrow_{\#}/(p \wedge q) \triangleleft segs$. The answer is *no*. For example, define the predicates p and q by

$$px \;\equiv\; \ll/x \;=\; \downarrow/x,$$

$$qx \;\equiv\; \gg/x \;=\; \uparrow/x.$$

In section 5.1 we construct a real-time algorithm for the longest p segment; in section 6 we construct an on-line algorithm for the longest q segment. Finding the longest $p \wedge q$ segment turns out to be more difficult; in section 6.2 we define $box = p \wedge q$ and construct a linear algorithm for the longest box segment which is not on-line. Of course, this does not yet prove that an on-line algorithm for the longest $p \wedge q$ segment does not exist.

More convincing is the following example: define the predicates p and q by

$$px \;\equiv\; \ll/x \leq \gg/x,$$

$$qx \;\equiv\; \ll/x \geq \gg/x.$$

In section 7 we shall give linear algorithms for the longest p segment and the longest q segment. However, in section 3 we have seen that a linear algorithm for the longest $p \wedge q$ segment does not exist.

How about negation: given an algorithm computing $\uparrow_{\#}/p \triangleleft segs$ is there a construction giving a similar algorithm computing $\uparrow_{\#}/\neg p \triangleleft segs$? The answer is again *no*. For example, let p be defined by

$$px \;\equiv\; \ll/x \neq \gg/x.$$

A linear algorithm for the longest p segment is easy to find; even a linear on-line algorithm is possible as we shall see in section 7. For the non-existence of a linear algorithm for the longest $\neg p$ segment we again refer to section 3.

4.2 Closedness properties

The following properties of segment predicates often occur:

- A segment predicate p is called *prefix-closed* if

$$p(x +\!\!+ y) \Rightarrow p(x)$$

  for all strings x and y.

- A segment predicate p is called *postfix-closed* if

$$p(x +\!\!+ y) \Rightarrow p(y)$$

  for all strings x and y.

- A segment predicate p is called *segment-closed* if it is both prefix-closed and postfix-closed, i.e.,

$$p(x +\!\!+ y +\!\!+ z) \Rightarrow p(y)$$

  for all strings x, y and z.

- A segment predicate p is called *overlap-closed* if

$$(y \neq [\,] \ \wedge \ p(x +\!\!+ y) \ \wedge \ p(y +\!\!+ z)) \ \Rightarrow \ p(x +\!\!+ y +\!\!+ z)$$

  for all strings x, y and z.

For example, the predicate defined by

$$\#x \ < \ \downarrow /x$$

for segments x of integers is segment-closed but not overlap-closed. On the other hand, the predicate *low* defined by

$$\#x \ > \ \uparrow /x$$

for segments x of integers is overlap-closed but not segment-closed.

As is easily verified, each of these four classes of predicates is closed under conjunction; also the three classes of prefix-closed, postfix-closed and segment-closed predicates are closed under disjunction. However, the class of overlap-closed predicates is not closed under disjunction. For example, both 'ascending' and 'descending' are overlap-closed, but 'ascending or descending' is not overlap-closed.

Before deriving real segment problem algorithms one remark has still to be made. What do we mean by $\uparrow_{\#}/p \triangleleft segs\ x$ if p doesn't hold for any segment of x, for example if p is defined to be always false? We can define it to be an abstract element, or the empty set or something like that. However, it doesn't matter. If we define p' for any predicate p by

$$p'x \equiv px \ \ \vee \ \ x = [\,],$$

then $\uparrow_{\#}/p \triangleleft segs\ x$ can be computed in constant time from $\uparrow_{\#}/p' \triangleleft segs\ x$, and $p'\,[\,] = true$. So we may, and shall, assume without loss of generality that $p\,[\,] = true$.

5 Solutions for prefix-closed predicates

As a universal invariant of all our loops we have that x is the part of the string that has been read. Further choose

$$s \;=\; \uparrow_{\#}/p \triangleleft segs\, x$$

as an invariant. Due to equation (3) then the following program is correct:

$s := [\,]$
; **do** *not eof* $\rightarrow$ *read(a)*
; $s := s \uparrow_{\#} (\uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a]))$
od

How to compute $\uparrow_{\#} /p \triangleleft tails\,(x +\!\!+ [a])$? If we also keep track of $\uparrow_{\#} /p \triangleleft tails\, x$ it remains to compute $\uparrow_{\#} /p \triangleleft tails\,(x +\!\!+ [a])$ from $\uparrow_{\#} /p \triangleleft tails\, x$. The following proposition states that for prefix-closed predicates indeed $\uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a])$ only depends on $\uparrow_{\#}/p \triangleleft tails\, x$. It can also be found as lemma 3.1 in [6].

Proposition 1 *Let p be a prefix-closed predicate. Then*

$$\uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a]) \;=\; \uparrow_{\#}/p \triangleleft tails\,((\uparrow_{\#}/p \triangleleft tails\, x) +\!\!+ [a]).$$

Proof: Let $t = \uparrow_{\#}/p \triangleleft tails\, x$. Then $tails\,(t +\!\!+ [a]) \subseteq tails\,(x +\!\!+ [a])$, so also

$$p \triangleleft tails\,(t +\!\!+ [a]) \;\subseteq\; p \triangleleft tails\,(x +\!\!+ [a]).$$

For proving the converse of this inclusion assume that $z \notin p \triangleleft tails\,(t +\!\!+ [a])$ for some $z \in p \triangleleft tails\,(x +\!\!+ [a])$. Then pz holds and $z \in tails\,(x +\!\!+ [a])$ and $z \notin tails\,(t +\!\!+ [a])$. So z can be written as $w +\!\!+ t +\!\!+ [a]$ for some non-empty string w. Since pz holds and p is prefix-closed, we see that $p(w +\!\!+ t)$ holds. So $w +\!\!+ t \in p \triangleleft tails\, x$, which contradicts the definition of t. We conclude that

$$p \triangleleft tails\,(x +\!\!+ [a]) \;=\; p \triangleleft tails\,(t +\!\!+ [a]),$$

so also

$$\uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a]) \;=\; \uparrow_{\#}/p \triangleleft tails\,(t +\!\!+ [a]),$$

which we had to prove. □

In fact, also the converse of this proposition holds: if p is any predicate that is not prefix-closed, it is not difficult to construct a string x and an element a such that

$$\uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a]) \;=\; x +\!\!+ [a] \;\neq\; \uparrow_{\#}/p \triangleleft tails\,((\uparrow_{\#}/p \triangleleft tails\, x) +\!\!+ [a]).$$

A consequence of the proposition is the following. Let p be a prefix-closed predicate, then choosing

$$s \;=\; \uparrow_{\#}/p \triangleleft segs\, x \;\;\wedge\;\; t \;=\; \uparrow_{\#}/p \triangleleft tails\, x$$

as an invariant leads to the following program:

$s := [\,]$
; $t := [\,]$
; **do** *not eof* $\rightarrow$ *read(a)*
; $t := \uparrow_{\#}/p \triangleleft tails\,(t +\!\!+ [a])$
; $s := s \uparrow_{\#} t$
od

5.1 Prefix-closed and overlap-closed

In order to derive a complete program, we still have to be able to compute

$$\uparrow_{\#}/p \triangleleft tails\,(t +\!\!+ [a]).$$

If p is also overlap-closed, this is easy according to the following proposition.

Proposition 2 *Let p be a predicate which is both prefix-closed and overlap-closed. Let $t = \uparrow_{\#}/p \triangleleft tails\, x$. Then*

$$\uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a])$$

is either

$$t +\!\!+ [a], \quad or \quad [a], \quad or \quad [\,].$$

Proof: Let $z = \uparrow_{\#}/p \triangleleft tails\,(x +\!\!+ [a])$. According to proposition 1 we have

$$z \;=\; \uparrow_{\#}/p \triangleleft tails\,(t +\!\!+ [a]),$$

so there exists a string w such that $t +\!\!+ [a] = w +\!\!+ z$. If $z = [\,]$ then we are done, so assume that $z \neq [\,]$. Then we can write $z = y +\!\!+ [a]$ for $y = init\, z$. Both pt and pz hold, and $t = w +\!\!+ y$ and $z = y +\!\!+ [a]$. Since p is overlap-closed, we conclude that either $y = [\,]$ or $p(w +\!\!+ y +\!\!+ [a])$ holds. In the former case we obtain $z = [a]$. In the latter case we have $p(t +\!\!+ [a])$, so

$$z \;=\; \uparrow_{\#}/p \triangleleft tails\,(t +\!\!+ [a]) = t +\!\!+ [a].$$

□

As a consequence, if p is both prefix-closed and overlap-closed, we obtain the following program, having

$$s \;=\; \uparrow_{\#}/p \triangleleft segs\, x \;\;\wedge\;\; t \;=\; \uparrow_{\#}/p \triangleleft tails\, x$$

as an invariant:

$$
\begin{array}{l}
\quad s := [\,] \\
;\ t := [\,] \\
;\ \textbf{do}\ \textit{not eof} \rightarrow \textit{read(a)} \\
\qquad ;\ \textbf{if}\ p(t +\!\!+ [a]) \qquad\qquad \rightarrow t := t +\!\!+ [a] \\
\qquad\quad [\!]\ \neg p(t +\!\!+ [a]) \wedge p[a] \ \rightarrow t := [a] \\
\qquad\quad [\!]\ \neg p(t +\!\!+ [a]) \wedge \neg p[a] \rightarrow t := [\,] \\
\qquad\quad \textbf{fi} \\
\qquad ;\ s := s \uparrow_{\#} t \\
\quad \textbf{od}
\end{array}
$$

If we want to consider this program as a real-time program, then $p(t +\!\!+ [a])$ has to be computable in constant time. Since p is prefix-closed, we have for all strings t:

$$p(t +\!\!+ [a]) \;=\; p(t) \wedge \text{'something'};$$

to compute 'something' it is often necessary to keep track of some help information $\phi(t)$. The result is the following proposition, which is easily checked.

Proposition 3 *Let p be a predicate which is both prefix-closed and overlap-closed. Assume there is a function ϕ, and a pair of constant computable functions π_1 and π_2, in such a way that*

$$p(t +\!\!+ [a]) \;=\; p(t) \wedge \pi_1(\phi(t), a)$$

and

$$\phi(t +\!\!+ [a]) \;=\; \pi_2(\phi(t), a)$$

for all strings t and all elements a. Let $\phi\,[\,] = A$. Then the following program is a real-time program computing $\uparrow_{\#}/p \triangleleft segs$, having

$$s \;=\; \uparrow_{\#}/p \triangleleft segs\, x \;\wedge\; t \;=\; \uparrow_{\#}/p \triangleleft tails\, x \;\wedge\; f \;=\; \phi(t)$$

as an invariant:

$$
\begin{array}{l}
\quad s := [\,] \\
;\ t := [\,] \\
;\ f := A \\
;\ \textbf{do}\ \textit{not eof} \rightarrow \textit{read(a)} \\
\qquad ;\ \textbf{if}\ \pi_1(f, a) \qquad\qquad\qquad \rightarrow t := t +\!\!+ [a] \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ f := \pi_2(f, a) \\
\qquad\quad [\!]\ \neg\pi_1(f, a) \wedge \pi_1(A, a) \quad \rightarrow t := [a] \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ f := \pi_2(A, a) \\
\qquad\quad [\!]\ \neg\pi_1(f, a) \wedge \neg\pi_1(A, a) \rightarrow t := [\,] \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ f := A \\
\qquad\quad \textbf{fi} \\
\qquad ;\ s := s \uparrow_{\#} t \\
\quad \textbf{od}
\end{array}
$$

Example: Let R be some constant time computable relation. Let p be defined as follows.

> $p(t)$ holds if and only if aRb holds for every two consecutive elements a and b in t.

In particular, p holds for strings of length ≤ 1. Let ω be any element not occurring in the string. Choose

$$\begin{aligned} \phi(t) &= \begin{cases} \gg/t & \text{if } t \neq [\,] \\ \omega & \text{if } t = [\,] \end{cases} \\ \pi_1(a,b) &= (a = \omega) \vee (aRb), \\ \pi_2(a,b) &= b. \end{aligned}$$

Then all requirements are fullfilled, so the proposition yields a real-time algorithm for computing the longest p segment.

If the relation R is the equality relation, this is a real-time solution of the *longest plateau problem*: given a string, find the longest segment of which all of the elements are equal.

Other examples of predicates obtained in the same way by a constant time computable relation R are

- ascending;
- descending;
- all elements are equal modulo some given number;
- any two consecutive elements differ at most some constant C;
- any two consecutive elements differ at least some constant C;

and all conjunctions between them.

. . . . . .

Example: Choose p by

$$p(x) \equiv \downarrow/x = \ll/x$$

for non-empty x, and $p\,[\,] = true$. Then choosing

$$\phi(t) = \ll/t$$

for non-empty strings t yields a real-time algorithm for computing the longest p segment.

Also for conjunctions of this predicate and cases of the former example the requirements of the proposition can be fulfilled easily if we choose

$$\phi(t) = (\ll/t, \gg/t).$$

For example, we obtain a real-time algorithm for computing the longest p segment, where p is defined by

$$p(x) \;\equiv\; (\downarrow/x \;=\; \ll/x) \wedge (\forall [a,b] \in \mathit{segs}\, x : b - a < C)$$

for some given constant C.

. . . . . .

Note that if p is any conjunction of predicates from these examples, we have a real-time algorithm for computing the longest p segment too. As remarked in section 4.1, the same holds for any disjunction.

5.2 Only prefix-closed

What to do if the predicate p is not overlap-closed? As before, let

$$t \;=\; \uparrow_{\#}/p \triangleleft \mathit{tails}\, x.$$

Then $\uparrow_{\#}/p \triangleleft \mathit{tails}\,(x +\!\!+ [a])$ is not any more either $t +\!\!+ [a]$ or $[a]$ or $[\,]$. But if the predicate p is prefix-closed, we still have that $\uparrow_{\#}/p \triangleleft \mathit{tails}\,(x +\!\!+ [a])$ is contained in $\mathit{tails}\,(t +\!\!+ [a])$. The number of candidates for $\uparrow_{\#}/p \triangleleft \mathit{tails}\,(x +\!\!+ [a])$ is linear in the length of t, so again choosing

$$s \;=\; \uparrow_{\#}/p \triangleleft \mathit{segs}\, x \;\wedge\; t \;=\; \uparrow_{\#}/p \triangleleft \mathit{tails}\, x$$

as the invariant will lead to an algorithm that is on-line or worse, and not real-time.

The following choice for the invariant is more fruitful:

$$s \;=\; \uparrow_{\#}/p \triangleleft \mathit{segs}\, x \;\wedge\; u \in \mathit{tails}\, x \;\wedge\; \#u = \#s.$$

Assume this invariant holds. Since both t and u are in $\mathit{tails}\, x$ and $\#t \leq \#s = \#u$ we obtain $t \in \mathit{tails}\, u$. So

$$\mathit{tails}\,(t +\!\!+ [a]) \subseteq \mathit{tails}\,(u +\!\!+ [a]) \subseteq \mathit{tails}\,(x +\!\!+ [a]).$$

Since p is prefix-closed we may apply proposition 1, and obtain

$$\uparrow_{\#}/p \triangleleft \mathit{tails}\,(x +\!\!+ [a]) \;=\; \uparrow_{\#}/p \triangleleft \mathit{tails}\,(u +\!\!+ [a]).$$

From the invariant and equation 3 now follows

$$\uparrow_{\#}/p \triangleleft \mathit{segs}\,(x +\!\!+ [a]) \;=\; s \uparrow_{\#} (\uparrow_{\#}/p \triangleleft \mathit{tails}\,(u +\!\!+ [a])).$$

We distinguish two cases: $p(u +\!\!+ [a])$ and $\neg p(u +\!\!+ [a])$. In the case of $p(u +\!\!+ [a])$ we clearly have

$$\uparrow_{\#}/p \triangleleft \mathit{tails}\,(u +\!\!+ [a]) \;=\; u +\!\!+ [a],$$

of which the length is $\#u + 1 > \#u = \#s$. So in that case

$$\uparrow_{\#}/p \triangleleft segs\,(x +\!\!+ [a]) \;=\; u +\!\!+ [a].$$

We conclude that in the case of $p(u +\!\!+ [a])$ the statements

$$u := u +\!\!+ [a]; \quad s := u$$

keep the invariant valid.

It remains to consider the other case: $\neg p(u +\!\!+ [a])$. Assume the invariant holds, then we have

$$\#\uparrow_{\#}/p \triangleleft tails\,(u +\!\!+ [a]) \;\leq\; \#(u +\!\!+ [a]) - 1 \;=\; \#u \;=\; \#s,$$

so

$$\uparrow_{\#}/p \triangleleft segs\,(x +\!\!+ [a]) \;=\; s \uparrow_{\#} (\uparrow_{\#}/p \triangleleft tails\,(u +\!\!+ [a]))$$

is either equal to s or equal to $s \uparrow_{\#} tail\,(u +\!\!+ [a])$, depending whether $p(tail\,(u +\!\!+ [a]))$ holds. So in the case of $\neg p(u +\!\!+ [a])$ the statements

```
  u := tail (u ++ [a])
; if p(u)   → s := s ↑# u
  [] ¬p(u) → skip
  fi
```

keep the invariant valid. If we assume that $z \uparrow_{\#} y = z$ for all z, y with $\#z = \#y$, then the if-statement may be left away. However, if all maximal p segments have to be computed, then this assumption does not hold, and the if-statement is essential.

Combining both cases, for p prefix-closed we obtain the following program for computing $\uparrow_{\#}/p \triangleleft segs$:

```
  s := [ ]
; u := [ ]
; do not eof → read(a)
             ; u := u ++ [a]
             ; if p(u)    → s := u
               [] ¬p(u)   → u := tail (u)
                          ; if p(u)   → s := s ↑# u
                            [] ¬p(u) → skip
                            fi
               fi
  od
```

If we want to consider this program as a real-time program, then p has to be cheap computable. One way to achieve this is by choosing some π_1 in such a way that

$$p(t +\!\!+ [a]) \;=\; p(t) \wedge \pi_1(\cdots)$$

for all t and a, as required in proposition 3. However, in contrary to proposition 3 here the invariant does not imply $p(t)$, so then $p(t +\!\!+ [a])$ cannot be computed from $\pi_1(\cdots)$ only.

Instead we require that p is in constant time computable from some help function ϕ, which can be built up real-time, and for which $\phi(tail\ t)$ can be computed in constant time from $\phi(t)$. The result is formulated in the following proposition.

Proposition 4 *Let p be a prefix-closed predicate. Assume there is a function ϕ, and three constant computable functions π_1, π_2 and π_3, in such a way that*

$$p(t) \;=\; \pi_1(\phi(t)) \quad and \quad \phi(t +\!\!+ [a]) \;=\; \pi_2(\phi(t), a)$$

for all strings t and all elements a, and

$$\phi(tail\ t) \;=\; \pi_3(\phi(t))$$

for all non-empty strings t for which $\neg p(t)$. Let $\phi\,[\,] = A$. Then the following program is a real-time program computing $\uparrow_{\#}/p \triangleleft segs$, having

$$s \;=\; \uparrow_{\#}/p \triangleleft segs\, x \;\wedge\; u \in tails\, x \;\wedge\; \#u = \#s \;\wedge\; f \;=\; \phi(u)$$

as an invariant:

```
    s := [ ]
;   u := [ ]
;   f := A
;   do not eof → read(a)
                 ;  u := u +++ [a]
                 ;  f := π2(f, a)
                 ;  if π1(f)     → s := u
                    [] ¬π1(f)    → u := tail u
                                 ;  f := π3(f)
                                 ;  if π1(f)     → s := s ↑# u
                                    [] ¬π1(f)    → skip
                                    fi
                    fi
    od
```

As already noted, if $\uparrow_\#$ is defined in such a way that $z \uparrow_\# y = z$ for all z, y with $\#z = \#y$, then the innermost if-statement may be left away.

Operationally the segment u shifts over the string from left to right, sometimes increasing, but never decreasing in length. Due to this operational idea this technique is sometimes called *windowing*.

Example: Given a string consisting of non-negative integers and a positive number C, find the longest segment of which the sum does not exceed C. Define

$$\phi(x) = (x, +/x)$$

for all strings x. Then we can define

$$\begin{aligned} \pi_1(x, n) &= (n \leq C), \\ \pi_2((x, n), a) &= (x +\!\!+ [a], n + a), \\ \pi_3(x, n) &= (tail\, x, n - \ll /x), \end{aligned}$$

and all requirements are fulfilled, so proposition 4 gives a real-time algorithm solving this problem.

If also negative numbers are allowed to occur in the string, the predicate is not prefix-closed any more, and proposition 4 cannot be applied. In section 7 we shall give a linear on-line algorithm solving that case.

. . . . . .

Example: The longest *ribbon* in an ascending string: given an ascending string of integers and a positive number C, find the longest segment of which the difference between the maximum and the minimum does not exceed C. Note that for every ascending string the minimum corresponds to the leftmost element and the maximum corresponds to the rightmost element. Define

$$\phi(x) = x$$

for all strings x. Then we can define

$$\begin{aligned} \pi_1(x) &= (\gg /x - \ll /x \leq C), \\ \pi_2(x, a) &= x +\!\!+ [a], \\ \pi_3(x) &= tail\, x, \end{aligned}$$

and all requirements are fulfilled, so proposition 4 gives a real-time algorithm solving this problem.

If the string is not required to be ascending, the predicate is still prefix-closed. In that case $\downarrow /x$ is expected to be an essential ingredient of $\phi(x)$, while $\downarrow /tail\, x$ cannot be computed in a straightforward way only using $\downarrow /x$. In section 6.1 we shall give a solution; the resulting program is linear on-line.

The longest ribbon problem on an ascending string can easily be transformed into the former example and vice versa. In the one direction a string $[a_1, a_2, \ldots, a_n]$ is transformed into

$$[a_2 - a_1, a_3 - a_2, \ldots, a_n - a_{n-1}],$$

in the other direction a string $x = [a_1, a_2, \ldots, a_n]$ is transformed into

$$[0, a_1, a_1 + a_2, \ldots, +/x].$$

. . . . . .

If the requirements of proposition 3 are fulfilled, namely:

- p is a predicate which is both prefix-closed and overlap-closed,
- there is a function $\tilde{\phi}$, and there are constant computable functions $\tilde{\pi_1}$ and $\tilde{\pi_2}$, in such a way that

$$p(t +\!\!+ [a]) \;=\; p(t) \wedge \tilde{\pi}_1(\tilde{\phi}(t), a)$$

and

$$\tilde{\phi}(t +\!\!+ [a]) \;=\; \tilde{\pi}_2(\tilde{\phi}(t), a)$$

for all strings t and all elements a,

then we obtain another real-time algorithm for the longest p segment as follows. Let $\tilde{\phi}\,[\,] = \tilde{A}$. Define

$$\begin{aligned}
\phi(t) &= (\#t, \# \uparrow_{\#} / p \triangleleft \mathit{tails}\ t, \tilde{\phi}(t)) \\
\pi_1(m, n, f) &= (m = n), \\
\pi_2((m, n, f), a) &= \begin{cases} (m+1, n+1, \tilde{\pi}_2(f, a)) & \text{if } \tilde{\pi}_1(f, a)) \\ (m+1, 1, \tilde{\pi}_2(\tilde{A}, a)) & \text{if } \neg\tilde{\pi}_1(f, a)) \wedge \tilde{\pi}_1(\tilde{A}, a) \\ (m+1, 0, \tilde{A}) & \text{if } \neg\tilde{\pi}_1(f, a)) \wedge \neg\tilde{\pi}_1(\tilde{A}, a) \end{cases} \\
\pi_3(m, n, f) &= (m - 1, n, f).
\end{aligned}$$

Now the requirements of proposition 4 are fulfilled and we obtain another real-time algorithm than in proposition 3. One can wonder why it is interesting to derive real-time algorithms for a class of problems for which real-time algorithms already are available. Apart from methodological arguments, it also gives the possibility to take the conjunction between a predicate for which proposition 3 is applicable and one for which proposition 4 is applicable.

6 Applying partitions

There is a close relationship between segment problems and partitions. In this section we shall see how a reasonable investigation of remarks that can be made about the combination of a segment predicate and partitions will lead in a natural way to a class of non-trivial solutions of segment problems.

What is a partition? Intuitively it is a way of splitting up a given string into a couple of segments. In our notation it can easily made precise: a *partition of a string* x is a string of strings xs in such a way that

$$+\!\!+/xs \;=\; x.$$

Choosing this notation does not mean that partitions have to be implemented as strings of strings. We only assume that some basic operations are computable in constant time, like glueing two consecutive partition segments together, extending the partition by a new segment on the right hand side, and inspecting the leftmost or rightmost segment of the partition. This holds for several representations of partitions.

We say that a partition satisfies a segment predicate p if each of the segments of the partition satisfies p. To be sure that for each x a partition satisfying p exists, we require that p holds for one-element strings. A very simple algorithm giving a partition xs satisfying p is:

$xs := [\,]$
; **do** *not eof* $\rightarrow$ *read(a)*
; $y := [a]$
; $xs := xs +\!\!+ [y]$
od

This gives a very trivial partition: it consists purely of the one-element segments, and does not contain any information about p. We are more interested in *maximal* partitions for p: we call a partition xs of x maximal for p if it satisfies p and for each two consecutive segments u, v of xs the concatenation $u +\!\!+ v$ does not satisfy p. In general maximal partitions are not unique. Note that the empty string does not occur as a segment in a maximal partition.

The most straightforward algorithm to produce a maximal partition is obtained by extending the former algorithm as follows:

$$
\begin{array}{l}
\quad xs := [\,] \\
;\ \textbf{do}\ \mathit{not}\ \mathit{eof} \rightarrow\ \mathit{read}(a) \\
\qquad\qquad ;\ \ y := [a] \\
\qquad\qquad ;\ \ \textbf{do}\ xs \neq [\,] \wedge p((\gg/xs) +\!\!+\, y) \rightarrow y := (\gg/xs) +\!\!+\, y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ;\ \ xs := \mathit{init}\,(xs) \\
\qquad\qquad\ \ \textbf{od} \\
\qquad\qquad ;\ \ xs := xs +\!\!+\, [y] \\
\quad \textbf{od}
\end{array}
$$

Let x be the part of the string already read, then the outer loop has as an invariant:

xs is a maximal partition for p of x,

and the inner loop has as invariants:

$xs +\!\!+\, [y]$ is a partition of x satisfying p, and
xs is a maximal partition for p of $+\!\!+/xs$.

This algorithm is called the *greedy algorithm.* At a first glance it looks like at least quadratic since it contains nested loops. However, in the inner loop the length of xs always decreases, and we can choose as a variant function

$$2 * \#x \ - \ \#xs,$$

showing that the greedy algorithm is linear in the length of the string, provided that $p((\gg/xs) +\!\!+\, y)$ can be computed in constant time. This algorithm is on-line, but in general not real-time.

Back to segment problems. How can a maximal partition for p be helpful for computing $\uparrow_{\#}/p \triangleleft \mathit{segs}$? The following proposition gives one possibility; we shall see that it is not the only one.

Proposition 5 *Let p be a postfix-closed and overlap-closed predicate. Let x be any string and let xs be a maximal partition for p of x. Then*

$$\uparrow_{\#}/p \triangleleft \mathit{tails}\, x \ = \ \gg/xs.$$

Proof: If $\#xs = 1$ then we have $x = \gg/xs$ and px holds, so

$$\uparrow_{\#}/p \triangleleft \mathit{tails}\, x \ = \ x \ = \ \gg/xs.$$

So we may assume that $\#xs \geq 2$. Let u and v be the last two elements of xs, i.e., $v = \gg/xs$ and $u = \gg/\mathit{init}\, xs$. Since xs is maximal for p we have pu and pv and $\neg p(u +\!\!+\, v)$. Let $w = \uparrow_{\#}/p \triangleleft \mathit{tails}\, x$; since pv and $v \in \mathit{tails}\, x$ we obtain $\#w \geq \#v$. Since $\neg p(u +\!\!+\, v)$ and p is postfix-closed we obtain $\#w < \#(u +\!\!+\, v)$. Since pu and pw and $\neg p(u +\!\!+\, v)$ and p is overlap-closed we obtain $w = v$, which we had to prove. □

Both requirements overlap-closed and postfix-closed are necessary in this proposition. For example, the predicate p defined by

$$px \equiv \#x \leq 2$$

is postfix-closed but not overlap-closed, while $\uparrow_{\#}/p \triangleleft tails\, x \neq\!\!\gg /xs$ if $\#x$ is odd and at least 3, and xs is the maximal partition for p of x obtained by the greedy algorithm. The predicate p defined by

$$px \equiv \#x \neq 2$$

is overlap-closed but not postfix-closed, while again $\uparrow_{\#}/p \triangleleft tails\, x \neq\!\!\gg /xs$ if $\#x$ is at least 3, and xs is the maximal partition for p of x obtained by the greedy algorithm.

We want to combine this proposition and the greedy algorithm. We also want the result to be applicable for predicates that do not necessarily hold for all one element strings. In order to reach that goal we define a predicate $\tilde{p}$ for each predicate p as follows

$$\tilde{p}x \equiv px \vee \#x \leq 1.$$

Clearly $\tilde{p}$ holds for all one element strings. Note that if p is overlap-closed and postfix-closed, then the same holds for $\tilde{p}$. Further one easily sees that $\uparrow_{\#}/p \triangleleft tails\, x$ is equal to $\uparrow_{\#}/\tilde{p} \triangleleft tails\, x$ if p holds for $\uparrow_{\#}/\tilde{p} \triangleleft tails\, x$, and otherwise it is equal to $[\,]$. As a consequence we obtain the following proposition.

Proposition 6 *Let p be a postfix-closed and overlap-closed predicate. Assume there is a function ϕ, and three constant computable functions π_1, π_2 and π_3, in such a way that*

$$p(t +\!\!+ u) = \pi_1(\phi(t), \phi(u)) \text{ if } \tilde{p}(t) \wedge \tilde{p}(u), \text{and}$$
$$\phi(t +\!\!+ u) = \pi_2(\phi(t), \phi(u))$$

for all non-empty strings t and u, and

$$\phi([a]) = \pi_3(a)$$

for all elements a. Then the following program is a linear on-line program computing $\uparrow_{\#}/p \triangleleft segs$:

```
   xs := [ ]
;  z := [ ]
;  s := [ ]
;  do not eof → read(a)
              ;  y := [a]
              ;  f := π3(a)
              ;  do xs ≠ [ ] ∧ π1(≫/z, f) → y := (≫/xs) ++ y
                                           ;  f := π2(≫/z, f)
                                           ;  xs := init(xs)
                                           ;  z := init(z)
                 od
              ;  if π1(f)  → s := s ↑# y
                 [] ¬π1(f) → skip
                 fi
              ;  xs := xs ++ [y]
              ;  z := z ++ [f]
   od
```

In this program the outer loop has as invariants:

> *xs is a maximal partition for $\tilde{p}$ of x, and*
> *$z = \phi^*(xs)$, and $s = \uparrow_\# / p \triangleleft segs\, x$.*

and the inner loop has as invariants:

> *$xs +\!\!+ [y]$ is a partition of x satisfying $\tilde{p}$, and*
> *xs is a maximal partition for $\tilde{p}$ of $+\!\!+ /xs$, and*
> *$z = \phi^*(xs)$, and $f = \phi(y)$.*

Example: Let p be defined by

$$p(t) \equiv \downarrow /t = \gg /t$$

for non-empty x, and $p[\,] = true$. Then by choosing

$$\phi(t) = \gg /t$$

all conditions are easily fulfilled, yielding a linear on-line algorithm for computing $\uparrow_\# / p \triangleleft segs$. If the implementation of partitions is in such a way that the last elements of its elements are available in constant time, the additional string z in the algorithm is easily eliminated.

An easier linear algorithm for the same problem is obtained by reading the elements from right to left instead of from left to right, and applying proposition 3. However, then the resulting algorithm is not on-line.

Depending on a relation R we can define a useful predicate p_R on non-empty segments:

$$p_R u \equiv (\forall a \in init\, u : aR \gg /u).$$

For example, the p in the last example is equal to $p_{\geq}$. By definition, p_R is postfix-closed and holds for one-element strings for all relations R. It is easily seen that p_R is overlap-closed if and only if R is transitive. The following proposition states that in the context of partitions the predicate p_R is easy to compute if R is transitive.

Proposition 7 *Let R be a transitive relation and let u and v be non-empty segments satisfying p_R. Then*

$$p_R(u +\!\!+ v) \equiv \gg /uR \gg /v.$$

Proof: If not $\gg /uR \gg /v$ then we have not $p_R(u +\!\!+ v)$ since $\gg /u$ is an element of $init\,(u +\!\!+ v)$ and $\gg /v = \gg /(u +\!\!+ v)$.

On the other hand assume that $\gg /uR \gg /v$ holds. Let a be an arbitray element of $init\,(u +\!\!+ v)$ and let $b = \gg /v = \gg /(u +\!\!+ v)$. We distinguish three cases:

- $a \in init\, v$. Since v satisfies p_R we have aRb.
- $a = \gg /u$. Since $\gg /uR \gg /v$ we have aRb.
- $a \in init\, u$. Since u satisfies p_R we have $aR \gg /u$. Since $\gg /uR \gg /v$ and R is transitive we have aRb.

In all cases we have aRb, so $u +\!\!+ v$ satisfies p_R, which we had to prove. □

Applying this proposition to the greeady algorithm yields a linear on-line algorithm for computing $\uparrow_{\#}/p_R \triangleleft segs$, which can also be found by choosing $\phi(t) = \gg /t$ proposition 6. This is a generalization of the above example.

However, partitions satisfying p_R have more interesting properties, which turn out to be useful for finding solutions of segment problems of other predicates than only p_R itself. The next proposition states that the minimum or the maximum of a segment are computable in constant time if a maximal partition for some p_R of that segment is available. For any relation R we define its complement R^C by

$$aR^C b \equiv \neg(aRb).$$

Proposition 8 *Let R be a relation for which both R and R^C are transitive. Let xs be a maximal partition for p_R of some non-empty string x. Let $b = \gg / \ll /xs$. Then*

- *aRb for all $a \in init \ll /xs$, i.e., all elements a left from b, and*
- *$bR^C a$ for all elements a of elements of $tail\, xs$, i.e., all elements a right from b.*

So for R being $<, \leq, \geq, >$, the element b is respectively the leftmost maximum of x, the rightmost maximum of x, the rightmost minimum of x and the leftmost minimum of x.

Proof: The first assertion holds since xs is a partition satisfying p_R. Since xs is maximal for p_R we have

$$\gg/uR^C \gg/v$$

for all consecutive segments u, v in xs. Since $b = \gg/ \ll/xs$ and R^C is transitive one proves by induction to the length of xs that

$$bR^C \gg/v$$

for all $v \in tail\, xs$. So for the rightmost elements of elements of $tail\, xs$ we are done. Let a be any element of $init\, v$ with $v \in tail\, xs$. Since xs satisfies p we have $aR \gg/v$. Assume bRa, by transitivity of R we then have $bR \gg/v$, contradiction. So $bR^C a$, which we had to prove. □

6.1 The longest ribbon

Given a positive constant C a segment of integers is called a ribbon if the greatest difference between the elements of that segment does not exceed C. Since the greatest difference is equal to the maximum minus the minimum, we can write this definition in our notation as follows:

$$ribbon\, t \equiv \uparrow/t - \downarrow/t \leq C.$$

A derivation of an $n \log n$ algorithm finding the longest ribbon can be found in [7]; in this section we shall apply partitions resulting in a linear on-line solution of the longest ribbon problem.

As a basis for our program we choose as invariants

- $s = \uparrow_\# / ribbon \triangleleft segs\, x$,
- $t = \uparrow_\# / ribbon \triangleleft tails\, x$;

as before, x is the part of the string that has been read already. Applying proposition 1 gives:

```
   s := []
;  t := []
;  do not eof → read(a)
                ;  t := t ++ [a]
                ;  do ¬ribbon t → t := tail t
                   od
                ;  s := s ↑# t
   od
```

as a correct program.

The problem now is how to compute *ribbon t*. Since *ribbon* holds for one-element strings, we see that $t \neq [\,]$ is an invariant of the inner loop. Since *ribbon t* is an invariant of the outer loop, we see that *ribbon init t* holds as a precondition for the inner loop. Moreover, *ribbon* is postfix-closed, so even *ribbon init t* is an invariant of the inner loop. Finally, it is easy to verify that for non-empty strings t with $\gg/t = a$ we have

$$ribbon\, t \;\equiv\; ribbon\, init\, t \;\wedge\; a \leq\downarrow /t + C \;\wedge\; a \geq\uparrow /t - C.$$

Hence our program may be modified to

$s := [\,]$
; $t := [\,]$
; **do** *not eof* $\rightarrow$ *read(a)*
; $t := t +\!\!+ [a]$
; **do** $a >\downarrow /t + C \;\vee\; a <\uparrow /t - C \rightarrow t := tail\, t$
od
; $s := s \uparrow_{\#} t$
od

How do we compute $\downarrow /t$ and $\uparrow /t$? Proposition 8 suggests to choose maximal partitions for $p_>$ and $p_<$ of t as help information to make $\downarrow /t$ and $\uparrow /t$ available in constant time. Now the problem remains how to compute maximal partitions for $p_>$ and $p_<$ of *tail t* from similar partitions of t. The next proposition states that this can be done in constant time, in a more general setting than we need for the longest ribbon problem.

Proposition 9 *Let p be an overlap-closed and postfix-closed predicate and let xs be a maximal partition for p of some non-empty string x. Let xs' be defined as follows:*

$$xs' \;=\; \begin{cases} tail\, xs & \text{if } \#(\ll/xs) = 1 \\ [tail \ll/xs] +\!\!+ tail\, xs & \text{otherwise.} \end{cases}$$

Then xs' is a maximal partition for p of tail x.

Proof: Since p is postfix-closed all segments of the partition xs' satisfy p. It remains to show that xs' is maximal: the concatenation of any two consecutive segments of xs' has to satisfy $\neg p$. The only possible concatenation of this kind which is not a similar concatenation in xs, is $(tail\, u) +\!\!+ v$, where u and v are the two leftmost segments of xs and $\#u \neq 1$. So *tail u* is not empty; from pv and $\neg p(u +\!\!+ v)$ and p is overlap-closed we obtain $\neg p((tail\, u) +\!\!+ v)$, which we had to prove. □

Now we have collected all ingredients for the solution of the longest ribbon problem. As invariants we choose

- $s = \uparrow_{\#}/ribbon \triangleleft segs\, x$,
- $t = \uparrow_{\#}/ribbon \triangleleft tails\, x$,
- ys is a maximal partition for $p_{>}$ of t, and
- zs is a maximal partition for $p_{<}$ of t.

As a consequence from proposition 8 we obtain

$$\downarrow /t = \gg / \ll /ys \quad \text{and} \quad \uparrow /t = \gg / \ll /zs.$$

The resulting program reads:

```
    s := []
;   t := []
;   ys := []
;   zs := []
;   do not eof → read(a)
                ;  t := t ++ [a]
                ;  y := [a]
                ;  do ys ≠ [] ∧ (≫/ ≫/ys) > a → y := (≫/ys) ++ y
                                                ;  ys := init(ys)
                   od
                ;  ys := ys ++ [y]
                ;  y := [a]
                ;  do zs ≠ [] ∧ (≫/ ≫/zs) < a → y := (≫/zs) ++ y
                                                ;  zs := init(zs)
                   od
                ;  zs := zs ++ [y]
                ;  do a > (≫/ ≪/ys) + C ∨ a < (≫/ ≪/zs) − C → ys := ys'
                                                            ;  zs := zs'
                                                            ;  t := tail t
                   od
                ;  s := s ↑# t
   od
```

The first and second inner loop are copied from the greedy algorithm; the guards can be chosen in this way according to proposition 7 and $a = \gg /y$. The partitions ys' and zs' in the third inner loop are defined as in proposition 9. The linearity of the program follows from the variant function

$$4\#x - \#t - \#ys - \#zs.$$

The program is on-line; it is not real-time.

6.2 A more general segment decomposition

Until now the solutions of segment problems were based upon the structural property equation (3) from section 2:

$$\uparrow_\#/p \triangleleft segs\,(x +\!\!+ [a]) \;=\; (\uparrow_\#/p \triangleleft segs\,x)\uparrow_\#(\uparrow_\#/p \triangleleft tails\,(x +\!\!+ [a])).$$

This property forces the computation of $\uparrow_\#/p \triangleleft segs$ to be done strictly from left to right. As we saw this often leads to a linear algorithm perfectly well, but for some predicates it does not. Of course by reversing it can also be done strictly from right to left, but then again a fixed direction is chosen. Since the notion of segments is perfectly symmetrical, we should like to have a defining property of segments which is symmetrical too. A useful property satisfying this requirement is

$$segs\,(x +\!\!+ [a] +\!\!+ y) \;=\; segs\,x \cup segs\,y \cup \{u +\!\!+ [a] +\!\!+ v \mid u \in tails\,x \wedge v \in inits\,y\}.$$

Intuitively this property is clear, and it can be proven from our definition of segments in a straightforward way. Applying equations (1) and (2) from section 2 to this property, we obtain

$$\uparrow_\#/p \triangleleft segs\,(x +\!\!+ [a] +\!\!+ y) \;= \tag{4}$$
$$(\uparrow_\#/p \triangleleft segs\,x)\uparrow_\#(\uparrow_\#/p \triangleleft segs\,y)\uparrow_\#(\uparrow_\#/p \triangleleft \{u +\!\!+ [a] +\!\!+ v \mid u \in tails\,x \wedge v \in inits\,y\}).$$

Note that equation (4) is a generalization of equation (3): if we choose $y = [\,]$ in equation (4) then the result is exactly equation (3).

In this section we shall combine equation (4) and maximal partitions to find solutions of segment problems. If for a particular choice of a the value

$$\uparrow_\#/p \triangleleft \{u +\!\!+ [a] +\!\!+ v \mid u \in tails\,x \wedge v \in inits\,y\}$$

is easy to compute, we may hope that we have

$$\phi(x +\!\!+ [a] +\!\!+ y) \;=\; \pi(\phi(x), a, \phi(y)) \tag{5}$$

for some cheap computable function π, where $\phi(x)$ is defined to be $\uparrow_\#/p \triangleleft segs\,x$.

The idea now is to compute $\phi(x)$ for a given string x by building up a partition of x, and applying equation (5) each time when two consecutive segments of the partition are glued together. If at the end the partition consists of only one segment, we are done. How can we apply equation (5) when two consecutive segments u and v are glued together? If $\phi(u)$ and $\phi(v)$ are available, we cannot apply equation (5) to compute $\phi(u +\!\!+ v)$. However, if $\phi(init\,u)$ and $\phi(init\,v)$ are available, we can apply equation (5) to compute

$$\phi(init\,(u +\!\!+ v)) \;=\; \pi(\phi(init\,u), \gg/u, \phi(init\,v)),$$

so that is what we are going to do. Let R be any transitive relation, and let x again be the part of the string already read. Let

$$f \;=\; \phi([\,]).$$

Choosing as invariants for the outer loop:

xs is a maximal partition for p_R of x, and
$z = (\phi \circ init)^* xs$

and for the inner loop:

$xs +\!\!+ [y]$ is a partition of x satisfying p_R, and
xs is a maximal partition for p_R of $+\!\!+ /xs$, and
$z = (\phi \circ init)^* xs$ and $c = \phi(init\, y)$

we can extend the greedy algorithm to

$xs := [\,]$
; **do** *not eof* $\rightarrow$ *read(a)*
; $y := [a]$
; $c := f$
; **do** $xs \neq [\,] \wedge (\gg / \gg /xs)Ra) \rightarrow y := (\gg /xs) +\!\!+ y$
; $c := \pi(\gg /z, \gg / \gg /xs, c)$
; $xs := init\,(xs)$
; $z := init\,(z)$
od
; $xs := xs +\!\!+ [y]$
; $z := z +\!\!+ [c]$
od

We shall refer to this algorithm by $(*)$. The guard of the inner loop has its shape according to proposition 7.

Under what conditions will the computation

$$\phi(u +\!\!+ [a] +\!\!+ v) \;=\; \pi(\phi(u), a, \phi(v))$$

be executed during this algorithm? From the guard of the inner loop and proposition 7 we see that it is only done if $u +\!\!+ [a]$ satisfies p_R, or, equivalently,

$$\forall b \in u : bRa.$$

If also R^C is transitive the inner loop has an extra invariant

$$xs \neq [\,] \;\Rightarrow\; (\forall b \in init\, y : (\gg / \gg /xs)R^C b). \tag{6}$$

Initially $y = [a]$ so $init\, y = [\,]$, so then this property holds. Next we prove that this property remains invariant after each step of the inner loop. If xs consists of one segment then after one step xs is empty and the property trivially holds. If xs consists of at least two segments let $u =\gg /init\, xs$ and $v =\gg /xs$ be the last two segments of xs. We have to prove that

$$\forall b \in init\,(v +\!\!+ y) : (\gg /u)R^C b.$$

assuming that equation (6) holds. We distiguish two cases: $b \in v$ and $b \in init\, y$. First let $b \in v$. Since xs satisfies p_R, in particular p_R holds for v, so $bR \gg /v$. Suppose that $(\gg /u)Rb$, then by transitivity of R we obtain $(\gg /u)R \gg /v$, contradicting the maximality of xs for p_R. Hence $(\gg /u)R^C b$. Next let $b \in init\, y$. From equation (6) we know $(\gg /v)R^C b$. Since xs is maximal for p_R we have $(\gg /u)R^C \gg /v$. Since R^C is transitive we conclude that also in this case $(\gg /u)R^C b$.

Now we have proved that equation (6) is an invariant of the inner loop; as a consequence the computation

$$\phi(u +\!\!+ [a] +\!\!+ v) \;=\; \pi(\phi(u), a, \phi(v))$$

is only executed during the algorithm if

$$\forall b \in v : aR^C b.$$

This algorithm was intended to compute $\phi(x)$ for a given string x. If we apply it to x it only computes $\phi(init\, u)$ for segments u of a partition of x, and not $\phi(x)$. We can bridge this gap by not applying the algorithm only to x, but to $x +\!\!+ [\omega]$ for some particular element ω in such a way that the corresponding partition of $x +\!\!+ [\omega]$ is forced to consist of only one segment. Then that segment is $x +\!\!+ [\omega]$, while $init\,(x +\!\!+ [\omega]) = x$, exactly what we need. The next proposition states how to choose ω.

Proposition 10 *Let R be any relation; let ω be an element such that $aR\omega$ for all elements a. Let x be an arbitrary string. Then $[x +\!\!+ [\omega]]$ is the only maximal partition for p_R of $x +\!\!+ [\omega]$.*

Proof: Assume there is a maximal partition xs for p_R of $x +\!\!+ [\omega]$ consisting of more than one segment. Let $u =\gg /init\, xs$ and $v =\gg /xs$ be the two leftmost segments of xs. Since $\gg /(u +\!\!+ v) = \omega$ and $aR\omega$ for all elements a we see that $u +\!\!+ v$ satisfies p_R, contradicting the maximality of xs. □

If the set of elements does not contain such an element ω, an abstract element ω can be added. Extending the relation R to this extended set by defining

> $aR\omega$ for all elements a, including ω, and $\omega R^C a$ for all elements a, excluding ω

does not affect the transitivity of R and R^C.

Combining the above observations we have proved the following, which is the main proposition of this section. On the one hand it provides the key idea for all examples in this section, on the other hand its applicability is not restricted to segment problems.

Proposition 11 *Let R be a relation for which both R and R^C are transitive and which is computable in constant time. Let ϕ be a function on strings for which there exists a constant computable π for which*

$$\phi(u +\!\!+ [a] +\!\!+ v) \quad = \quad \pi(\phi(u), a, \phi(v))$$

for all strings u and v and elements a for which

$$\forall b \in u : bRa \quad \textit{and} \quad \forall b \in v : aR^C b.$$

Then for any string x the above algorithm $(*)$ applied to $x +\!\!+ [\omega]$ is a linear algorithm computing $\phi(x)$.

Since the result is not available before adding the element ω to the input, the resulting algorithm is in general not on-line.

Often the ordinary number order "$<$" is chosen for R. In that case the condition on the computation of $\phi(u +\!\!+ [a] +\!\!+ v)$ is equivalent to: a is the leftmost maximum of $u +\!\!+ [a] +\!\!+ v$. For R being $\leq, \geq, >$, it is respectively the rightmost maximum, the rightmost minimum and the leftmost minimum. The condition $\forall b \in u : b < a$ we shall often abbreviate to the equivalent condition $\uparrow / u < a$, and similar for $\leq, \geq, >$.

As noted by S.D. Swierstra, this proposition is closely related to precedence parsing.

Example: The longest low segment: we are looking for

$$\uparrow_{\#} / low \triangleleft segs\,,$$

where *low* is defined by

$$low\ x \quad \equiv \quad \uparrow / x < \# x.$$

Define

$$\phi(x) = (\# x, \uparrow_{\#} / low \triangleleft segs\ x)$$

for each segment x, and let R be either "$<$" or "$\leq$". If $\uparrow / x \leq a$ and $\uparrow / y \leq a$ then

$$\phi(x +\!\!+ [a] +\!\!+ y) = \begin{cases} (len, x +\!\!+ [a] +\!\!+ y) & \text{if } a < len \\ (len, x_2 \uparrow_{\#} y_2) & \text{otherwise} \end{cases}$$

where

$$\phi(x) = (x_1, x_2),$$
$$\phi(y) = (y_1, y_2),$$
$$len = x_1 + y_1 + 1.$$

So proposition 11 can be applied and the longest low segment can be computed in linear time. The resulting algorithm was first found by R.S. Bird and L.G.T.M.

Meertens before it was discovered to be a particular case of this far more general proposition.

A very nice and totally different solution of this problem is treated in [3]: after two scans of preprocessing the longest low segment is found in one linear scan.

. . . . . .

Example: The longest box segment: we are looking for

$$\uparrow_{\#}/box \triangleleft segs\,,$$

where *box* is defined by

$$box\ x \quad \equiv \quad \ll/x = \downarrow/x \wedge \gg/x = \uparrow/x.$$

Let p be defined by

$$p\ x \quad \equiv \quad \ll/x = \downarrow/x$$

and define

$$\phi(x) \quad = \quad (\uparrow_{\#}/p \triangleleft tails\ x, \downarrow/x, \uparrow_{\#}/box \triangleleft segs\ x)$$

for each segment x, and choose R to be "$\leq$" (here "$<$" will not suffice). If $\uparrow/x \leq a$ and $\uparrow/y < a$ then

$$\phi(x +\!\!+ [a] +\!\!+ y) = \begin{cases} (x_1 +\!\!+ [a] +\!\!+ y, & x_2, & x_3 \uparrow_{\#} y_3 \uparrow_{\#} (x_1 +\!\!+ [a])) & \text{if } x_2 \leq y_2 \\ (y_1, & y_2, & x_3 \uparrow_{\#} y_3 \uparrow_{\#} (x_1 +\!\!+ [a])) & \text{if } x_2 > y_2 \end{cases}$$

where

$$\phi(x) = (x_1, x_2, x_3),$$
$$\phi(y) = (y_1, y_2, y_3).$$

So proposition 11 holds and the longest *box* segment can be computed in linear time.

This problem is also treated in [3]. Surprisingly, there it is called being *really difficult*, at least more difficult than the *low* segment problem, while in our approach it is of the same degree of difficulty.

. . . . . .

Example: A variation on the longest low segment: we are looking for

$$\uparrow_{\#}/p \triangleleft segs\,,$$

where p is defined by

$$p\ x \quad \equiv \quad \uparrow/x + \downarrow/x < \#x.$$

Define

$$\phi(x) = (\#x,\ \downarrow/x,\ \uparrow_{\#}/p \triangleleft segs\ x)$$

for each segment x, and let R again be either "$<$" or "$\leq$". If $\uparrow/x \leq a$ and $\uparrow/y \leq a$ then

$$\phi(x +\!\!+ [a] +\!\!+ y) = \begin{cases} (len,\ min,\ x +\!\!+ [a] +\!\!+ y) & \text{if } a + min < len \\ (len,\ min,\ x_3 \uparrow_{\#} y_3) & \text{otherwise} \end{cases}$$

where

$$\phi(x) = (x_1, x_2, x_3),$$
$$\phi(y) = (y_1, y_2, y_3),$$
$$len = x_1 + y_1 + 1,$$
$$min = x_2 \downarrow y_2.$$

Again the longest p segment can be computed in linear time.

. . . . . .

Example: The converse of the former example: we are looking for

$$\uparrow_{\#}/p \triangleleft segs\,,$$

where p is defined by

$$p\,x \quad \equiv \quad \uparrow/x + \downarrow/x > \#x.$$

Define

$$\phi(x) \;=\; (\#x,\ \uparrow/x,\ \uparrow_{\#}/p \triangleleft segs\,x)$$

for each segment x, and let R be either "$>$" or "$\geq$". If $\downarrow/x \geq a$ and $\downarrow/y \geq a$ then we have to compute $\phi(x +\!\!+ [a] +\!\!+ y)$ using a, $\phi(x)$ and $\phi(y)$. Before we can do so we need some observations. Write

$$\phi(x) = (x_1, x_2, x_3) \quad \text{and} \quad \phi(y) = (y_1, y_2, y_3).$$

Note that

$$p([a] +\!\!+ y) \;\equiv\; (a + y_2 > y_1 + 1).$$

First assume $x_2 \leq y_2$ and $a + y_2 > y_1 + 1$. Then $p([a] +\!\!+ y)$ equals true. May be this segment $[a] +\!\!+ y$ can be extended to the left while p remains to hold. For any tail segment $\tilde{x}$ of x we have

$$\downarrow/(\tilde{x} +\!\!+ [a] +\!\!+ y) = a \quad \text{and} \quad \uparrow/(\tilde{x} +\!\!+ [a] +\!\!+ y) = y_2.$$

So among all of these segments $\tilde{x} +\!\!+ [a] +\!\!+ y$ is the longest one for which p holds is obtained by choosing $\tilde{x}$ to be the tail segment of x of length

$$(a + y_2 - y_1 - 2) \downarrow x_1.$$

Since $\uparrow/x +\!\!+ [a] +\!\!+ y = y_2$ we conclude that $\tilde{x} +\!\!+ [a] +\!\!+ y$ is the longest segment of $x +\!\!+ [a] +\!\!+ y$ containing a that satisfies p.

Next assume $x_2 \leq y_2$ and $a + y_2 \leq y_1 + 1$. Let

$$w = \tilde{x} +\!\!+ [a] +\!\!+ \tilde{y}$$

where $\tilde{x}$ is any tail segment of x and $\tilde{y}$ is any initial segment of y. Suppose $p(w)$ holds, then

$$\#w \; < \uparrow /w + \downarrow /w \; \leq \; y_2 + a \; \leq \; y_1 + 1.$$

So $\#w \leq y_1$, and there exists a segment $\tilde{w}$ of y for which

$$\#\tilde{w} \; = \; \#w \quad \text{and} \quad \uparrow /\tilde{w} \; = \uparrow /y \; = \; y_2.$$

Then we have

$$\uparrow /\tilde{w} + \downarrow /\tilde{w} \; \geq \; y_2 + a \; > \; \#w \; = \; \#\tilde{w}$$

so also $p(\tilde{w})$ holds. We conclude that

$$\uparrow_{\#} /p \triangleleft segs\,(x +\!\!+ [a] +\!\!+ y) \;\; = \;\; x_3 \uparrow_{\#} y_3.$$

By interchanging x and y in the above two cases all cases are covered. Combining all four cases we obtain

$$\phi(x +\!\!+ [a] +\!\!+ y) \;\; = \;\; (x_1 + y_1 + 1,\; x_2 \uparrow y_2,\; z),$$

where

$$z = \begin{cases} x_3 \uparrow_{\#} y_3 & \text{if } x_2 \leq y_2 \wedge a + y_2 \leq y_1 + 1 \\ & \text{or if } y_2 \leq x_2 \wedge a + x_2 \leq x_1 + 1 \\ x_3 \uparrow_{\#} \tilde{x} +\!\!+ [a] +\!\!+ y & \text{if } x_2 \leq y_2 \wedge a + y_2 > y_1 + 1 \\ y_3 \uparrow_{\#} x +\!\!+ [a] +\!\!+ \tilde{y} & \text{if } y_2 \leq x_2 \wedge a + x_2 > x_1 + 1 \end{cases}$$

where $\tilde{x}$ is the tail segment of x of length

$$(a + y_2 - y_1 - 2) \downarrow x_1$$

and $\tilde{y}$ is the initial segment of y of length

$$(a + x_2 - x_1 - 2) \downarrow y_1.$$

Using proposition 11 the longest p segment now can be computed in linear time. If only the length of the longest p segment has to be computed, the algorithm can slightly be simplified.

This algorithm is rather complicated. The problem seems to be indeed rather difficult; linear solutions of this very simply formulated problem not applying proposition 11 are not known by the author, and are left as a challenge to the reader.

7 Leftmost at most rightmost

In this section we shall present a linear on-line algorithm finding the longest segment of which the leftmost element is less or equal to the rightmost element. This predicate can be described more formal and general as follows. Let R be an antisymmetric relation of which the complement R^C is transitive, for example, R equals $\leq$. Let the predicate p be defined by

$$px \equiv x = [\,] \vee (\ll/x)R(\gg/x).$$

Note that in general p does not satisfy any of the properties prefix-closed, postfix-closed and overlap-closed.

For the algorithm we need an additional string. In the case of R equals '$\leq$' this additional string can be interpreted as follows: for each element of the original string the additional string contains the minimum of all elements that have been read before that element. In order to achieve this, it is convenient to define $\gg/$ on the empty string. We introduce an abstract element ω and define

$$\gg/[\,] = \omega.$$

To allow ω to be the left argument of R we define $\omega R^C a$ for all elements a; clearly this definition does not affect the transitivity of R^C. The additional string y can now be built up as follows:

$y := [\,]$
; **do** *not eof* $\rightarrow$ *read(a)*
; **if** $(\gg/y)Ra) \rightarrow y := y +\!\!+ [\gg/y]$
[] $(\gg/y)R^C a) \rightarrow y := y +\!\!+ [a]$
fi
od

As usual, let x be the part of the string already read. Clearly $\#x = \#y$ is an invariant of this program. An interesting invariant which is easily verified using the antisymmetry of R and the transitivity of R^C, is the following:

for every x_1, x_2, y_1, y_2 satisfying

$$x = x_1 +\!\!+ x_2 \wedge y = y_1 +\!\!+ y_2 \wedge \#x_1 = \#y_1 \wedge \#x_2 = \#y_2$$

we have

$$\forall b \in x_1 : (bR^C \gg/y_1) \vee b = \gg/y_1$$

and

$$(y_2 \neq [\,] \wedge \gg/y_1 \neq \ll/y_2) \Rightarrow \ll/x_2 = \ll/y_2.$$

In particular by choosing $y_1 = [\,]$ we have

$$(x \neq [\,]) \Rightarrow (\ll/x = \ll/y).$$

In this invariant the way how to split up x into x_1 and x_2 is still free. Instead of choosing $x_2 = \uparrow_{\#}/p \triangleleft tails\, x$ as an extra invariant we apply the windowing technique from section 5.2 and add the invariant:

$$\#x_2 \;=\; \#s \;\wedge\; s \;=\; \uparrow_{\#}/p \triangleleft segs\, x.$$

The resulting program reads as follows:

$y_1 := [\,]$
; $y_2 := [\,]$
; $x_1 := [\,]$
; $x_2 := [\,]$
; **do** *not eof* $\rightarrow$ *read(a)*
; $x_2 := x_2 +\!\!+ [a]$
; **if** $(\gg/y_2)Ra \rightarrow y_2 := y_2 +\!\!+ [\gg/y_2]$
[] $(\gg/y_2)R^C a \rightarrow y_2 := y_2 +\!\!+ [a]$
fi
; $y_1 := y_1 +\!\!+ [\ll/y_2]$
; $y_2 := tail\, y_2$
; $x_1 := x_1 +\!\!+ [\ll/x_2]$
; $x_2 := tail\, x_2$
; **do** $y_1 \neq [\,] \wedge (\gg/y_1)Ra \rightarrow y_2 := [\gg/y_1] +\!\!+ y_2$
; $y_1 := init\, y_1$
; $x_2 := [\gg/x_1] +\!\!+ x_2$
; $x_1 := init\, x_1$
od
; $s := s \uparrow_{\#} x_2$
od

The only non-trivial part in the correctness proof of this algorithm is the invariance of

$$\#x_2 \;=\; \#s \;\wedge\; s \;=\; \uparrow_{\#}/p \triangleleft segs\, x.$$

To prove this invariance, consider the postcondition of the inner loop. The negation of the guard is

$$y_1 = [\,] \;\vee\; \gg/y_1 R^C a;$$

since $\forall b \in x_1 : (bR^C \gg/y_1) \vee b = \gg/y_1$ and R^C is transitive we obtain

$$\forall b \in x_1 : bR^C a.$$