Science of Computer Programming 18 (1992) 39-66
Elsevier

Longest segment problems

H. Zantema

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

Communicated by R. Bird
Received October 1990
Revised May 1991

Abstract

Zantema, H., Longest segment problems, Science of Computer Programming 18 (1992) 39-66.

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 terms of 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 (i.e., 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 *longest segment problem*, or, for short, a *segment problem*. If nothing is known about p the only way to solve this problem is to compute all segments of the string, determine for each segment whether p holds, and keep track of the longest segment satisfying p. 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 a strong optimization can often be made, resulting in a linear algorithm. This paper is an investigation of possible optimizations of this type. Many solutions of segments problems have appeared in the literature, e.g., [6], and exercises 1, 19, 27, 31 and 50 in [9]. We add several new solutions, including one of the following problem: 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.

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 Section 4 some basic properties of algorithms like 'on-line' and 'real-time' are discussed, as are some basic properties of predicates, like 'prefix-closed'. In Sections 5 and 6 the first algorithms appear. Although they are simple, they are the starting point for the classification presented in this paper.

The most interesting part of this paper is Section 7: the application of partitions. The main result is Proposition 10, which provides the key for many solutions, and is also applicable to 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 can also be found in [6] and [12]. One difference is that we tend to on-line algorithms and avoid preprocessing.

In Section 8 the problem of finding two elements in a string with the left one $\leqslant$ the right one, and as far apart as possible, is treated as a segment problem. The last section contains an overview of the results.

2. Notation

In many presentations, segment problems are expressed using indexing. We prefer to avoid introducing indexing in favour of basic operations on strings. Although we do not follow the way of program development as suggested in [1], we borrow some notation.

When no confusion is possible, parentheses of function application and function composition symbols are sometimes omitted, so gfx means $g(f(x)) = (g \circ f)(x)$.

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 $+\!\!+$, the size function by $\#$. Further we write

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

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

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

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

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

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. The operator $\oplus$ needs to be associative; if it is also commutative and idempotent then $\oplus/$ is also defined on finite sets as well as strings.

Without loss of generality we always assume that $p[\,] = \mathit{true}$. The set X will always contain the empty string $[\,]$, so $[\,] \in p \lhd X$.

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. Hence $\uparrow_{\#}/$ denotes the longest of a string of segments or a set of segments. 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. A similar property for $p\lhd$, immediate from its definition, is

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

for each predicate p and all sets of strings X and Y.

Finding the longest segment satisfying p now can be described as computing the function $\uparrow_{\#}/p\lhd segs$, where $segs\,x$ denotes the set of segments of x. We still have to define *segs*; we do it in terms of initial and tail segments. Let

$$tail[a_1, a_2, \ldots, a_n] = [a_2, a_3, \ldots, a_n],$$

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

As a consequence, 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

$$tails[a_1, a_2, \ldots, a_n] = \{[a_1, a_2, \ldots, a_n], [a_2, \ldots, a_n], \ldots, [a_n], []\},$$

$$inits[a_1, a_2, \ldots, a_n] = \{[], [a_1], [a_1, a_2], \ldots, [a_1, a_2, \ldots, a_n]\}.$$

Now we define *segs* inductively:

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

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

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

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

For many predicates p this is the property we need for the derivation of an algorithm for computing $\uparrow_{\#}/p\lhd segs$. In Section 7.3 however, we shall need a generalization.

All of our algorithms consist of the initialization of some variables and a main loop. In this main loop the given string is read from left to right; each element is inspected only once. Correctness proofs and complexity bounds are based on the notions of *invariants* and *variant functions* as in [7]. Let z be the total string to be considered, x the part of z already read, and y the part of z still to be read. As an invariant we then have

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

The general program reads as follows:

initialization
; $x := []$
; $y := z$
; **do** $y \neq [] \rightarrow a := \ll / y$
; $x := x +\!\!+ [a]$
; $y :=$ *tail* y
; *body*
od

All algorithms in this paper consist of filling in the *initialization* and the *body* in this program scheme. In giving invariants, the general invariant $z = x +\!\!+ y$ is left implicit. In the other invariants and in the algorithms z and y will not occur. The variable a will appear in the algorithms as the last read element.

3. Superlinear segment problems

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

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

So if there is a linear algorithm to compute $\uparrow_{\#} / p \lhd \mathit{segs}$, 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.)

Note that array indexing is *not* considered as a basic step in our model. If it is a basic step for arrays of which the index type equals the type of the string elements, then a linear element uniqueness algorithm can be given.

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

example, if

$$p(x) \equiv (\ll/x + \gg/x = C)$$

for some constant C, or if

$$p(x) \equiv (+/x = C)$$

for some constant C, then one can prove that there is no linear algorithm computing $\uparrow_{\neq}/p \lhd segs$, 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 our algorithms elements of a string z are inspected from left to right. An algorithm computing $f(z)$ 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. If the time between any two consecutive inspections in an on-line algorithm is independent of the length of the string then the algorithm is called *real-time*. For example, an algorithm containing some preprocessing of the given string 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. Conversely, it can happen that the time between any two consecutive inspections is not bounded while the average time is bounded (the cost is *amortized constant* as it is sometimes called in the literature). In that case the algorithm is both linear and on-line, but not real-time.

In [1] an on-line algorithm is written as $\pi(\oplus \not\leftarrow s)$, and is called a *directed reduction*.

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

$$(p \vee q) \lhd X = (p \lhd X) \cup (q \lhd X)$$

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

$$\uparrow_{\neq}/(p \vee q) \lhd segs\, x = (\uparrow_{\neq}/p \lhd segs\, x) \uparrow_{\neq} (\uparrow_{\neq}/q \lhd segs\, x).$$

So, given two linear algorithms computing $\uparrow_{\neq}/p \lhd segs$ and $\uparrow_{\neq}/q \lhd segs$, we can construct a linear algorithm computing $\uparrow_{\neq}/(p \vee q) \lhd segs$ by joining them together and taking the $\uparrow_{\neq}$ 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 $\uparrow_{\neq}/p \lhd segs$ and $\uparrow_{\neq}/q \lhd segs$, is there a similar construction giving an algorithm computing $\uparrow_{\neq}/(p \wedge q) \lhd segs$? The answer is *no*. For example, define the predicates p and q by

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

$$q(x) \equiv (\gg/x = \uparrow/x).$$

In Section 5 we construct a real-time algorithm for the longest p segment; in Section 7 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 7.3 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

$$p(x) \equiv (\ll/x \leqslant \gg/x),$$

$$q(x) \equiv (\ll/x \geqslant \gg/x).$$

In Section 8 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 $\uparrow_{\neq}/p \lhd segs$ is there a construction giving a algorithm computing $\uparrow_{\neq}/\neg p \lhd segs$? The answer is again *no*. For example, let p be defined by

$$p(x) \equiv (\ll/x \neq \gg/x).$$

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 8. For the non-existence of a linear algorithm for the longest $\neg p$ segment we again refer to Section 3.

4.2. *Closure properties*

- 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, postfixed-closed and segment-closed predicates are closed under disjunction. 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.

5. The first solutions

In the program scheme of Section 2 choose

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

as an invariant. By the assumption $p[\,] = true$ we may choose $s := [\,]$ as the initialization, by equation (3) we may choose

$$s := s \uparrow_{\#} (\uparrow_{\#}/p \lhd tails(x +\!\!+ [a]))$$

as the body. The following proposition states that for prefix-closed predicates $\uparrow_{\#}/p \lhd tails(x +\!\!+ [a])$ only depends on $\uparrow_{\#}/p \lhd tails\, x$ and a. It can also be found as Lemma 3.1 in [2].

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

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

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

$$p \lhd tails(t +\!\!+ [a]) \subseteq p \lhd tails(x +\!\!+ [a]).$$

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

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

so $\uparrow_{\#}/p \lhd tails(x +\!\!+ [a]) = \uparrow_{\#}/p \lhd tails(t +\!\!+ [a])$, which we had to prove. □

A consequence is the following. Let p be a prefix-closed predicate. Then we can choose

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

as the invariant in the program scheme, $s := [\,]$; $t := [\,]$ as the initialization and

$$t := \uparrow_{\#}/p \lhd tails(t +\!\!+ [a])$$
$$; s := s \uparrow_{\#} t$$

as the body. In order to derive a complete program, we still have to be able to compute $\uparrow_{\#}/p \lhd tails(t +\!\!+ [a])$. 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 $t = \uparrow_{\#}/p \lhd tails\, x$. Then $\uparrow_{\#}/p \lhd tails(x +\!\!+ [a])$ is either $t +\!\!+ [a]$, or $[a]$, or $[\,]$.*

Proof. Let $z = \uparrow_{\neq} / p \lhd tails(x +\!\!+ [a])$. According to Proposition 1 we have

$$z = \uparrow_{\neq} / p \lhd tails(t +\!\!+ [a]),$$

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 t and z satisfy p, while $t = w +\!\!+ y$ 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 $z = \uparrow_{\neq} / p \lhd tails(t +\!\!+ [a]) = t +\!\!+ [a]$. □

As a consequence, if p is both prefix-closed and overlap-closed, we can choose

$$s = \uparrow_{\neq} / p \lhd segs\ x \wedge t = \uparrow_{\neq} / p \lhd tails\ x$$

es the invariant, $s := []$; $t := []$ as the initialization, and

$$\begin{array}{ll} \textbf{if } p(t +\!\!+ [a]) & \rightarrow t := t +\!\!+ [a] \\ [] \neg p(t +\!\!+ [a]) \wedge p[a] & \rightarrow t := [a] \\ [] \neg p(t +\!\!+ [a]) \wedge \neg p[a] & \rightarrow t := [] \\ \textbf{fi} & \\ ; s := s \uparrow_{\neq} t & \end{array}$$

as the body of the program scheme. For the resulting algorithm to be real-time, $p(t +\!\!+ [a])$ 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.

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, such 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. Then there is a real-time algorithm computing $\uparrow_{\neq} / p \lhd segs$.

Proof. Let $\phi[] = A$. Choose

$$s = \uparrow_{\neq} / p \lhd segs\ x \wedge t = \uparrow_{\neq} / p \lhd tails\ x \wedge f = \phi(t)$$

as the invariant, $s := [\,]$; $t := [\,]$; $f := A$ as the initialization, and

$$\begin{array}{ll}
\textbf{if } \pi_1(f, a) & \rightarrow t := t +\!\!+ [a] \\
 & \;; f := \pi_2(f, a) \\
[\!] \neg\pi_1(f, a) \wedge \pi_1(A, a) & \rightarrow t := [a] \\
 & \;; f := \pi_2(A, a) \\
[\!] \neg\pi_1(f, a) \wedge \neg\pi_1(A, a) & \rightarrow t := [\,] \\
 & \;; f := A \\
\textbf{fi} & \\
; s := s\uparrow_{\neq} t &
\end{array}$$

as the body of the program scheme. □

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 $\leqslant 1$. Let ω be any element not occurring in the string. Choose

$$\phi(t) = \begin{cases} \gg/t & \text{if } t \neq [\,], \\ \omega & \text{if } t = [\,], \end{cases}$$

$$\pi_1(a, b) = (a = \omega) \vee (aRb),$$

$$\pi_2(a, b) = b.$$

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* [7, Section 16.3]: given a string, find the longest segment of which all of the elements are equal. □

Example. Choose p by

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

for non-empty x, and $p[\,] = \mathit{true}$. Then choosing $\phi(t) = \ll/t$ for non-empty strings t yields a real-time algorithm for computing the longest p segment. □

If p is any conjunction of predicates from these examples, a real-time algorithm for computing the longest p segment is obtained by choosing $\phi(t) = (\ll/t, \gg/t)$.

For example, this holds for

$$p(x) \equiv (\downarrow/x = \ll/x) \wedge (\forall [a, b] \in segs\, x\colon b - a < C)$$

for some given constant C. As remarked in Section 4.1, real-time algorithms can also be obtained for disjunctions.

6. The windowing technique

What to do if the predicate p is not overlap-closed? As before, let $t = \uparrow_{\#}/p \lhd tails\, x$. Then $\uparrow_{\#}/p \lhd tails(x +\!\!+ [a])$ is not any more either $t +\!\!+ [a]$ or $[a]$ or $[]$. But if the predicate p is prefix-closed, we still have that $\uparrow_{\#}/p \lhd tails(x +\!\!+ [a])$ is contained in $tails(t +\!\!+ [a])$. The number of candidates for $\uparrow_{\#}/p \lhd tails(x +\!\!+ [a])$ is linear in the length of t. So keeping track of $t = \uparrow_{\#}/p \lhd tails\, x$ in the program and choosing

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

as the invariant as we did before will not give a real-time algorithm.

The following choice for the invariant is more fruitful:

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

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

$$tails(t +\!\!+ [a]) \subseteq tails(u +\!\!+ [a]) \subseteq tails(x +\!\!+ [a]).$$

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

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

From the invariant and equation (3) now follows

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

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

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

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

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

We conclude that in the case of $p(u +\!\!+ [a])$ the statements $u := u +\!\!+ [a]$; $s := u$ keep the invariant valid.

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

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

so

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

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

$$\begin{array}{l} u := tail(u +\!\!+ [a]) \\ ; \textbf{if } p(u) \quad \rightarrow s := s\uparrow_{\#} u \\ \quad [] \neg p(u) \rightarrow \textbf{skip} \\ \quad \textbf{fi} \end{array}$$

keep the invariant valid.

Combining both cases, for p prefix-closed we may choose $s := []$; $u := []$ as the initialization and

$$\begin{array}{l} u := u +\!\!+ [a] \\ ; \textbf{if } p(u) \quad \rightarrow s := u \\ \quad [] \neg p(u) \rightarrow u := tail(u) \\ \qquad\qquad ; \textbf{if } p(u) \quad \rightarrow s := s\uparrow_{\#} u \\ \qquad\qquad\quad [] \neg p(u) \rightarrow \textbf{skip} \\ \qquad\qquad\quad \textbf{fi} \\ \quad \textbf{fi} \end{array}$$

as the body of the program scheme.

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*.

To obtain a real-time algorithm from the above result, we need an efficient computation of p as is described 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, such that*

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

for all strings t and all elements a, and

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

for all non-empty strings t for which $\neg p(t)$. Then there is a real-time algorithm computing $\uparrow_{\#}/p \lhd segs$.

Proof. Let $\phi[] = A$. Choose

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

as the invariant, $s := []$; $u := []$; $f := A$ as the initialization, and

$$
\begin{array}{l}
\quad u := u +\!\!+ [a] \\
; f := \pi_2(f, a) \\
; \textbf{if } \pi_1(f) \rightarrow s := u \\
\quad [] \neg \pi_1(f) \rightarrow u := \mathit{tail}\, u \\
\qquad\qquad ; f := \pi_3(f) \\
\qquad\qquad ; \textbf{if } \pi_1(f) \rightarrow s := s \uparrow_{\neq} u \\
\qquad\qquad\quad [] \neg \pi_1(f) \rightarrow \textbf{skip} \\
\qquad\qquad\quad \textbf{fi} \\
\textbf{fi}
\end{array}
$$

as the body of the program scheme. □

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

$$\pi_1(x, n) = (n \leqslant C),$$

$$\pi_2((x, n), a) = (x +\!\!+ [a], n + a),$$

$$\pi_3(x, n) = (\mathit{tail}\, x, n - \ll / x),$$

and all requirements of the proposition are fulfilled, giving a real-time algorithm.

If negative numbers are also allowed to occur in the string, the predicate is no longer prefix-closed, and Proposition 4 cannot be applied. In Section 8 we shall show how a linear on-line algorithm for that case can be given. □

7. Applying partitions

There is a close relationship between segment problems and partitions. A *partition of a string* x is defined to be a string of strings xs such that

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

We say that a partition satisfies a segment predicate p if each of the segments of the partition satisfies p. To ensure that for each x a partition satisfying p exists, we require that p holds for one-element strings. A partition xs of x is called *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 of a non-empty string.

Choose in the program scheme $xs := [\,]$ as the initialization and as the body:

$$\begin{aligned} & y := [a] \\ & ; \textbf{do } xs \neq [\,] \wedge p((\gg/xs) +\!\!+ y) \to y := (\gg/xs) +\!\!+ y \\ & \qquad\qquad\qquad\qquad\qquad\qquad ; \; xs := init(xs) \\ & \textbf{od} \\ & ; xs := xs +\!\!+ [y] \end{aligned}$$

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 $+\!\!+/xs$.

As a consequence, the resulting algorithm computes a maximal partition. It is called the *greedy algorithm.* 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. The algorithm is on-line, but in general not real-time. It is the basis of all algorithms of this section.

The next proposition states that for a particular class of predicates $p((\gg/xs) +\!\!+ y)$ can indeed be computed in constant time. Given a relation R on elements, we define the predicate p_R on non-empty segments by

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

For example, we have

$$p_{\geq}(t) \equiv (\downarrow/t = \gg/t).$$

By definition, p_R is postfix-closed and holds for singletons. Further p_R is overlap-closed if and only if R is transitive.

Proposition 5. *Let R be a transitive relation and let u and v be non-empty segments satisfying p_R. Then*

$$p_R(u +\!\!+ v) = (\gg/u)R(\gg/v).$$

Proof. If not $(\gg/u)R(\gg/v)$ 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 $(\gg/u)R(\gg/v)$ holds. Let a be an arbitrary 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 $(\gg/u)R(\gg/v)$ we have aRb.
- $a \in init\, u$. Since u satisfies p_R we have $aR(\gg/u)$. Since $(\gg/u)R(\gg/v)$ and R is transitive we have aRb.

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

Combining this proposition and the greedy algorithm yields a linear on-line algorithm for computing a maximal partition for p_R. Three independent ways of applying maximal partitions to segment problems are given in the next three subsections, respectively.

7.1. *The longest p_R segment*

Proposition 6. *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*

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

Proof. If $\#xs = 1$ then we have $x = \gg/xs$ and $p(x)$ holds, so

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

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 $u = \gg/init\, xs$. Since xs is maximal for p we have $p(u)$ and $p(v)$ and $\neg p(u + v)$. Let $w = \uparrow_{\#}/p \lhd tails\, x$; since $p(v)$ and $v \in tails\, x$ we obtain $\#w \geq \#v$. Since $\neg p(u + v)$ and p is postfix-closed we obtain $\#w < \#(u + v)$. Since $p(u)$ and $p(w)$ and $\neg p(u + v)$ and p is overlap-closed we obtain $w = v$, which we had to prove. □

Both overlap-closed and postfix-closed are necessary requirements in this proposition. For example, the predicate $(\#x \leq 2)$ is postfix-closed but not overlap-closed, and the predicate $(\#x \neq 2)$ is overlap-closed but not postfix-closed. For both predicates a counterexample to the proposition is easily found.

Combining Propositions 5 and 6 we can modify the body of the greedy algorithm to

$$\begin{aligned} &y := [a] \\ &; \textbf{do } xs \neq [] \wedge (\gg/\gg/xs)\, R\, (\gg/y) \rightarrow y := (\gg/xs) + y \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ;\ xs := init(xs) \\ &\ \textbf{od} \\ &; s := s\uparrow_{\#} y \\ &; xs := xs + [y] \end{aligned}$$

having

$$s = \uparrow_{\#}/p_R \lhd segs\, x$$

as an extra invariant. So if R is transitive and computable in constant time, then we have a linear on-line algorithm computing the longest p_R segment.

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.

7.2. 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:

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

The problem is to find the longest ribbon in a given string in linear time. Although *ribbon* is both prefix-closed and postfix-closed, it is rather difficult. We solve it by using partitions. Another solution is given in [4]. Although it is claimed to be $n \log n$, it can be proven to be linear.

Choose as an invariant

$$s = \uparrow_{\neq}/ribbon \lhd segs\ x \wedge t = \uparrow_{\neq}/ribbon \lhd tails\ x.$$

In the general program scheme we can choose $s := []$; $t := []$ as the initialization and

$$\begin{aligned} & t := t +\!\!+ [a] \\ & ; \textbf{do } \neg ribbon\ t \rightarrow t := tail\ t \\ & \quad \textbf{od} \\ & ; s := s\uparrow_{\neq} t \end{aligned}$$

as the body. The problem is how to compute $ribbon(t)$. Since *ribbon* holds for singletons, 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. Since *ribbon* is postfix-closed $ribbon(init\ t)$ 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 the body may be modified to

$$\begin{aligned} & t := t +\!\!+ [a] \\ & ; \textbf{do } a > \downarrow/t + C \vee a < \uparrow/t - C \rightarrow t := tail\ t \\ & \quad \textbf{od} \\ & ; s := s\uparrow_{\neq} t \end{aligned}$$

Now the problem is to compute $\downarrow/t$ and $\uparrow/t$ efficiently. The next more general proposition gives a solution, applying maximal partitions. For any relation R we define its complement R^C by

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

Proposition 7. *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., for all elements a to the left of b, and*
- *$bR^C a$ for all elements a of elements of $tail\ xs$, i.e., for all elements a to the right of b.*

So for R being $<$, $\leqslant$, $\geqslant$, $>$, 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

$$(\gg/u)R^C(\gg/v)$$

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. □

The next step in the longest ribbon problem is 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 8. *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 $p(v)$ 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 the invariant we choose

$$\begin{aligned} s = {} & \uparrow_{\#}/ ribbon \lhd segs\, x \wedge t = \uparrow_{\#}/ ribbon \lhd tails\, x \\ & \wedge\ y \text{ is a maximal partition for } p_> \text{ of } t \\ & \wedge\ zs \text{ is a maximal partition for } p_< \text{ of } t. \end{aligned}$$

As a consequence from Proposition 7 we obtain

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

The initialization is $s := []$; $t := []$; $ys := []$; $zs := []$, the resulting body reads:

$$
\begin{array}{l}
\quad t := t +\!\!+ [a] \\
; y := [a] \\
; \textbf{do } ys \neq [] \wedge (\gg/\gg/ys) > a \rightarrow y := (\gg/ys) +\!\!+ y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ; \; ys := init(ys) \\
\quad \textbf{od} \\
; ys := ys +\!\!+ [y] \\
; y := [a] \\
; \textbf{do } zs \neq [] \wedge (\gg/\gg/zs) < a \rightarrow y := (\gg/zs) +\!\!+ y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ; \; zs := init(zs) \\
\quad \textbf{od} \\
; zs := zs +\!\!+ [y] \\
; \textbf{do } a > (\gg/\ll/ys) + C \vee a < (\gg/\ll/zs) - C \rightarrow ys := ys' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ; \; zs := zs' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ; \; t := tail\ t \\
\quad \textbf{od} \\
; s := s \uparrow_{\#} t
\end{array}
$$

The first and second inner loop are simply copied from the greedy algorithm; the guards can be chosen in this way according to Proposition 5 and $a = \gg/y$. The partitions ys' and zs' in the third inner loop are defined as in Proposition 8. The linearity of the algorithm follows from the invariant function

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

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

A similar (but easier) algorithm can be found for the problem of the largest square under a histogram: find the longest p segment where

$$p(x) \equiv \downarrow/x \geqslant \#x.$$

Here we need only one partition: a maximal partition for $p_{>}$ of $\uparrow_{\#}/p \lhd tails\ x$. Another solution of this problem is given in [10].

7.3. A more general segment decomposition

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

$$\uparrow_{\#}/p \lhd segs(x +\!\!+ [a]) = (\uparrow_{\#}/p \lhd segs\ x) \uparrow_{\#} (\uparrow_{\#}/p \lhd tails(x +\!\!+ [a])).$$

This property forces the computation of $\uparrow_{\#}/p \lhd segs$ to be done strictly from left to right. Since the notion of segments is perfectly symmetrical, we should like to have a defining property of segments which is symmetrical too, like

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

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

$$\begin{aligned}&\uparrow_{\#}/p \lhd segs(x +\!\!+ [a] +\!\!+ y)\\ &\quad = (\uparrow_{\#}/p \lhd segs\, x) \uparrow_{\#} (\uparrow_{\#}/p \lhd segs\, y) \uparrow_{\#}\\ &\qquad (\uparrow_{\#}/p \lhd \{u +\!\!+ [a] +\!\!+ v \mid u \in tails\, x \wedge v \in inits\, y\}).\end{aligned} \tag{4}$$

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. The idea is to compute some function ϕ on strings closely related to $\uparrow_{\#}/p \lhd segs$. This will be done by the greedy algorithm for some p_R, while keeping track of ϕ-values of inits of the partition segments. If at the end the partition consists of only one segment, then the ϕ-value of the init of the whole string has been computed, and so has $\uparrow_{\#}/p \lhd segs$. We assume that ϕ has a property similar to equation (4):

$$\phi(x +\!\!+ [a] +\!\!+ y) = \pi(\phi(x), a, \phi(y)),$$

where π is some efficiently computable function. As a consequence,

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

Let R be any transitive relation, and let $f = \phi([])$. We extend the body of the greedy algorithm to

$$\begin{aligned}&y := [a]\\ &; c := f\\ &; \textbf{do } xs \neq [] \wedge (\gg/\gg/xs) R a \to y := (\gg/xs) +\!\!+ y\\ &\qquad\qquad\qquad\qquad\qquad\qquad ; \; c := \pi(\gg/z, \gg/\gg/xs, c)\\ &\qquad\qquad\qquad\qquad\qquad\qquad ; \; xs := init(xs)\\ &\qquad\qquad\qquad\qquad\qquad\qquad ; \; z := init(z)\\ &\quad \textbf{od}\\ &; xs := xs +\!\!+ [y]\\ &; z := z +\!\!+ [c]\end{aligned}$$

The invariant of the outer loop is:

$$xs \text{ is a maximal partition for } p_R \text{ of } x \wedge z = (\phi \circ init)^* xs.$$

The invariant of the inner loop is:

$$\begin{aligned}&xs +\!\!+ [y] \text{ is a partition of } x \text{ satisfying } p_R\\ &\wedge\ xs \text{ is a maximal partition for } p_R \text{ of } +\!\!+/xs\\ &\wedge\ z = (\phi \circ init)^* xs \wedge c = \phi(init\, y).\end{aligned}$$

The guard of the inner loop has its shape according to Proposition 5. We shall refer to the resulting algorithm by $(*)$.

From the guard of the inner loop and Proposition 5 we see that the computation

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

is only executed if $u +\!\!+ [a]$ satisfies p_R, or, equivalently, $(\forall b \in u: bRa)$.

Next we show that if the complement R^C of R is transitive, the inner loop has an extra invariant

$$xs \neq [] \Rightarrow (\forall b \in \mathit{init}\, y: (\gg/\gg/xs) R^C b). \tag{5}$$

Initially $y = [a]$ so $\mathit{init}\, y = []$, so then (5) holds. If xs consists of one segment then after one step xs is empty and (5) trivially holds. If xs consists of at least two segments, let $u = \gg/\mathit{init}\, xs$ and $v = \gg/xs$ be the last two segments of xs. We have to prove that

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

assuming that equation (5) holds. We distinguish two cases: $b \in v$ and $b \in \mathit{init}\, y$. 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 $b \in \mathit{init}\, y$. From equation (5) 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 (5) 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 both $(\forall b \in u: bRa)$ and $(\forall b \in v: aR^C b)$.

This algorithm was intended to compute $\phi(x)$ for a given string x. If we apply it to x, then it only computes $\phi(\mathit{init}\, u)$ 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 $\mathit{init}(x +\!\!+ [\omega]) = x$, exactly what we need. The next proposition states how to choose ω.

Proposition 9. *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 $u = \gg/\mathit{init}\, xs$ and $v = \gg/xs$ be the two rightmost 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,
$\omega R^C a$ 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 10. *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 exiists a constant computable π for which*

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

for all strings u and v and elements a for which

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

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 $\leqslant$, $\geqslant$, $>$, it is respectively the rightmost maximum, the rightmost minimum and the leftmost minimum. The condition $(\forall b \in u\colon b < a)$ can be abbreviated to the equivalent condition $\uparrow / u < a$, and similarly for $\leqslant$, $\geqslant$, $>$.

If in Proposition 10 the function ϕ is replaced by the function ϕ' defined by

$$\phi'(x) = (x, \phi(x)),$$

we can weaken the condition on ϕ: it is also allowed that the constant time computation of $\phi(u +\!\!+ [a] +\!\!+ v)$ not only depends on $\phi(u)$, a and $\phi(v)$, but also on u and v. In the examples we shall indeed assume that the corresponding segments u and v are available for this computation.

As noted by S.D. Swierstra, Proposition 10 is closely related to precedence parsing.

Example. The longest low segment: we are looking for

$$\uparrow_{\#} / \mathit{low} \lhd \mathit{segs},$$

where *low* is defined by

$$\mathit{low}(x) \equiv \uparrow / x < \# x.$$

Define

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

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

$$\phi(x) = (x_1, x_2), \qquad \phi(y) = (y_1, y_2),$$

$$len = x_1 + y_1 + 1.$$

So Proposition 10 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.L.T. 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 [6]: after two scans of preprocessing the longest low segment is found in one linear scan.

Example. The longest box segment; *box* is defined by

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

Let p be defined by

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

and define

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

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

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

where

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

So Proposition 10 holds and the longest *box* segment can be computed in linear time.

This problem is also treated in [6]. 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: define p by

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

Let

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

for each segment x, and let R again be either "$<$" or "$\leqslant$". Then Proposition 10 can be applied, so the longest p segment can be computed in linear time. □

Example. The converse of the former example: define p by

$$p(x) \equiv (\uparrow/x + \downarrow/x > \#x).$$

Define

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

for each segment x, and let R be either "$>$" or "$\geqslant$". If $\downarrow/x \geqslant a$ and $\downarrow/y \geqslant a$ 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 $x_2 \leqslant y_2$ 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 the segments $\tilde{x} +\!\!+ [a] +\!\!+ y$, 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 $x_2 \leqslant y_2$ and $a + y_2 \leqslant y_1 + 1$. 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

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

So $\#w \leqslant y_1$, 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

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

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

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

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

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

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 10 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 be simplified slightly.

This algorithm is rather complicated. The problem seems to be indeed rather difficult; linear solutions of this very simply formulated problem not applying Proposition 10 are not known by the author, and are left as a challenge to the reader. □

The applicability of Proposition 10 is not restricted to longest segment problems. It is also useful for partition problems and other problems concerning segments. For example, by choosing $\phi(x) = \#x$ and choosing "$>$" or "$\geqslant$" for R, we easily obtain a linear algorithm finding the rectangle of largest area under a histogram. This algorithm is the essential ingredient of an algorithm finding the maximal area of any constant zero submatrix of a given matrix, linear in the number of matrix elements. This problem was first presented by R.S. Bird, and is also discussed in [12].

8. Leftmost at most rightmost

In this last section we 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 is generalized as follows. Let R be an antisymmetric relation of which the complement R^C is transitive, for example, R equals $\leqslant$. Define the predicate p by

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

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 y. In the case of R equals "$\leqslant$" 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. Choose $y := []$ as the initialization and

if $(\gg/y)Ra) \rightarrow y := y +\!\!+ [\gg/y]$
$[]\ (\gg/y)R^C a) \rightarrow y := y +\!\!+ [a]$
fi

as the body of the program scheme. As usual let x be the part of the string already read; clearly $\#x = \#y$ is an invariant of the 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 to split up x into x_1 and x_2 is not determined. Instead of choosing $x_2 = \uparrow_{\#}/p \lhd \mathit{tails}\, x$ as an extra invariant we apply the windowing technique from Section 6 and add the invariant:

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

The initialization reads $y_1 := []$; $y_2 := []$; $x_1 := []$; $x_2 := []$, the total body in the program scheme is

$$
\begin{aligned}
& x_2 := x_2 +\!\!+ [a] \\
& ; \textbf{if } (\gg/y_2)Ra \rightarrow y_2 := y_2 +\!\!+ [\gg/y_2] \\
& \ [] (\gg/y_2)R^C a \rightarrow y_2 := y_2 +\!\!+ [a] \\
& \ \textbf{fi} \\
& ; y_1 := y_1 +\!\!+ [\ll/y_2] \\
& ; y_2 := \mathit{tail}\, y_2 \\
& ; x_1 := x_1 +\!\!+ [\ll/x_2] \\
& ; x_2 := \mathit{tail}\, x_2 \\
& ; \textbf{do } y_1 \neq [] \wedge (\gg/y_1)Ra \rightarrow y_2 := [\gg/y_1] +\!\!+ y_2 \\
& \qquad\qquad\qquad\qquad\quad ; y_1 := \mathit{init}\, y_1 \\
& \qquad\qquad\qquad\qquad\quad ; x_2 := [\gg/x_1] +\!\!+ x_2 \\
& \qquad\qquad\qquad\qquad\quad ; x_1 := \mathit{init}\, x_1 \\
& \ \textbf{od} \\
& ; s := s{\uparrow_{\#}} x_2
\end{aligned}
$$

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

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

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 $(\forall b \in x_1 : bR^C a)$. Expressed in words: no tail of x longer than x_2 satisfies p. If the body of the inner loop is not executed the inner loop has $\#x_2 = \#s$ as a postcondition, and we are done. In the other case the body is executed at least once and the inner loop has $(\ll/y_2)Ra$ as a postcondition. Combining this with $(\gg/y_1)R^C a$ we obtain $\gg/y_1 \neq \ll/y_2$; from the invariant we then conclude that $\ll/x_2 = \ll/y_2$, so $(\ll/x_2)Ra$, so x_2 satisfies p. Combining this with the above remarks we conclude that x_2 is the longest tail satisfying p, which we had to prove.

The linearity of the algorithm follows from the variant function

$$\#x + \#x_2,$$

which increases by one in every step of the algorithm. From the structure of the algorithm we see that it is on-line.

If only the length of the longest p segment is to be computed, all x_1 and x_2 can simply be removed in the algorithm, since the length of x_2 is equal to the length of y_2. If both x and y are implemented as arrays, then the assignments for x_1, x_2, y_1 and y_2 can be written shorter as one shift of an array index.

Choosing R to be $\leqslant$ we have solved the 'leftmost at most rightmost' segment problem. A small modification in the algorithm leads to a linear on-line algorithm for the longest p_C segment, where

$$p_C(x) \equiv x = [] \vee (\ll/x) \leqslant (\gg/x) + C$$

for any constant C. The composition of this algorithm and the transformation mapping $[a_1, a_2, \ldots, a_n]$ to $[a_2 - a_1, a_3 - a_2, \ldots, a_n - a_{n-1}]$ yields a linear on-line algorithm finding the longest segment of which the sum is at most C. In this solution both positive and negative numbers are allowed in the string. In Section 6 we already found a real-time solution for the same problem in the case that the string consists only of non-negative numbers.

9. Conclusions and final remarks

In this paper we have derived and presented a number of algorithms determining the longest segment of a given string satisfying some fixed predicate p. All of the algorithms are linear in the length of the string. In most cases the predicate p consists of restrictions on linear combinations of simple functions on segments x, like the leftmost element $\ll/x$, the rightmost element $\gg/x$, the maximum $\uparrow/x$, the minimum $\downarrow/x$, the sum $+/x$ or the length $\#x$. We found real-time algorithms finding the longest p segment for $p(x)$ defined by

$$\downarrow/x = \ll/x,$$

or

$$+/x \leqslant C \quad \text{for strings of non-negative numbers.}$$

We found linear on-line algorithms finding the longest p segment for $p(x)$ defined by

$$\downarrow/x = \gg/x,$$

or

$$\uparrow/x - \downarrow/x \leqslant C,$$

or

$$\downarrow/x \geqslant \#x,$$

or

$$\ll/x \leqslant \gg/x,$$

or

$$+/x \leqslant C.$$

Further we found linear algorithms finding the longest p segment for $p(x)$ defined by

$$\uparrow/x < \#x,$$

or

$$\ll/x = \downarrow/x \wedge \gg/x = \uparrow/x,$$

or

$$\uparrow/x + \downarrow/x < \#x,$$

or

$$\uparrow/x + \downarrow/x > \#x.$$

Applying the same techniques, solutions can be found for slightly modified predicates, for example for

$$2*\uparrow/x + 3*\downarrow/x \geqslant 4*\#x + 5.$$

However, not all similar predicates lead to linear algorithms. For example, for $p(x)$ defined to be

$$\ll/x = \gg/x,$$

or

$$+/x = C$$

the complexity of any algorithm finding the longest p segment in a string of length n can be proven to be at least $O(n \log n)$.

For the predicates

$$\uparrow/x - \downarrow/x \leqslant \#x,$$

and

$$\uparrow/x - \downarrow/x \geqslant \#x$$

our methods failed until now; the corresponding segment problems are still open.

Not all known linear solutions of segment problems are treated in this paper. One example is the palindrome problem: find the longest segment that is its own reverse. In [8] a linear on-line algorithm solving this problem is given. Rather difficult is the problem of the longest square segment, a segment is called square if it is of the shape $u + u$. Related results can be found in [3, 11].

Our methods can also be applied to partition problems and other problems concerning segments. For example, in a given matrix the maximal constant zero submatrix can be found linear in the number of matrix elements.

References

[1] R.S. Bird, An introduction to the theory of lists, in: M. Broy, ed., *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series **F36** (Springer, Berlin, 1987).

[2] R.S. Bird, J. Gibbons and G. Jones, Formal derivation of a pattern matching algorithm, *Sci. Comput. Programming* **12** (1989) 93–104.

[3] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45** (1986) 63–86.

[4] E.W. Dijkstra, The problem of the longest ribbon, EWD-note 943 (1985).

[5] D.P. Dobkin and R.J. Lipton, On the complexity of computations under varying sets of primitives, *J. Comput. System Sci.* **18** (1979) 86–91.

[6] J.P.H.W. van den Eijnde, Left-bottom and right-top segments, *Sci. Comput. Programming* **15** (1990) 79–94.
[7] D. Gries, *The Science of Programming* (Springer, Berlin, 1981).
[8] J.T. Jeuring, The derivation of an algorithm for finding palindromes, *Notes of the International Summer School on Constructive Algorithmics*, Hollum, Ameland, the Netherlands, 1989.
[9] M. Rem, Small program exercises, *Sci. Comput. Programming* **3** (1983) 313–319, **5** (1985) 309–316, **7** (1986) 87–97 and 243–248, **11** (1988) 167–173.
[10] M. Rem, Small program exercises 17, *Sci. Comput. Programming* **8** (1987) 307–313.
[11] M. Rem, Small program exercises 20, *Sci. Comput. Programming* **10** (1988) 99–105.
[12] J.C.S.P van der Woude, Rabbitcount := Rabbitcount − 1, in: J.L.A. van de Snepscheut, ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science **375** (Springer, Berlin, 1989).