Backing up in Singly Linked Lists

Amir M. Ben-Amram*
The Academic College of Tel-Aviv
4 Antokolski Street
64044 Tel-Aviv, Israel
amirben@mta.ac.il

Holger Petersen
University of Stuttgart
Universitätsstr. 38
D-70569 Stuttgart
petersen@informatik.uni-stuttgart.de

Abstract

We show how to reduce the time overhead for implementing two-way movement on a singly linked list to $O(n^{\epsilon})$ per operation without modifying the list and without making use of storage other than a finite number of pointers into the list. We also prove a matching lower bound.

These results add precision to the intuitive feeling that doubly linked lists are more efficient than singly linked lists, and quantify the efficiency gap in a read-only situation. We further analyze the number of points of access into the list (pointers) necessary for obtaining a desired value of ϵ. We obtain tight tradeoffs which also separate the amortized and worst-case settings.

Our upper bound implies that read-only programs with singly-linked input can do string matching much faster than previously expected.

1 Introduction

In this work we consider read-only programs that process a singly-linked list. A read-only program may neither modify its input nor allocate additional

*Part of this author's work was done during a visit to DIKU, at the University of Copenhagen. This visit was supported by DART, a project under the Danish Research Councils.

storage; it makes use of a fixed set of pointer variables for accessing the input list. When no other type of variable is used, we have a *pure pointer* program. Read-only programs of this kind are discussed for instance by Jones [5] who writes:

> ... one has a suspicion that such programs will take extra time [compared to programs with two-way access to their input] due to the complexity of "backing up" to inspect an already-seen input.

This intuition on the inefficiency of backing up in a read-only list is part of programming folklore, and sometimes used to motivate the introduction of doubly-linked lists (e.g., [3]). In the rest of this paper we will employ the attributes "one-way" and "two-way" rather than "singly linked" or "doubly linked," because they refer more precisely to the difference which we are studying. Results for the restricted class of pure pointer programs without recursion could be phrased in terms of one-way multi-head finite automata. The variables correspond to heads of automata and assignments could be interpreted as head-to-head jumps. The latter is however a rather unusual model and we prefer the more general framework from [5].

The obvious method for reaching the previous element in a one-way list is to restart scanning from the head of the list, and takes $O(n)$ time. We show that for every $k > 0$, a program with pointers that move both ways can be simulated by a program with one-way pointers at a multiplicative overhead of $O(n^{1/k})$ in running time. The strategy of nesting intervals employed here has been introduced by Chandra in the context of compiling linear recursive programs [2]. It is rephrased in such a way as to enable mixing access to the input and other operations of the program being simulated. This is the key to efficient string-matching with pure pointer programs, answering a question raised in [6]. Further the $O(n^{1/k})$ time bound is not only obtained in an amortized sense, but also in the worst-case per simulated step, using a coroutine construction.

The bounds can be achieved by pure pointer programs; however we obtain a finer understanding of the results by considering programs that have two types of variables, pointers for accessing the list and bounded counters for calculations. For such programs, we show a tight tradeoff between the time overhead and the number of pointer variables in the simulator. Specifically, to achieve $O(n^{1/k})$ amortized overhead, $(k-1)d+2$ one-way pointers suffice to simulate d two-way pointers. For the worst-case bound, we use $(k-1)(d+1)+2$ pointers in the simulator. We prove that these results are optimal (give or take one pointer). Thus, we settle the question of the complexity of simulating two-way access for programs with a fixed number of variables. We also quantify the resource penalty for "de-amortizing" a time bound, a topic of independent interest. Swamy and Savage [13] present a similar trade-off for linear recursion, but their proof of optimality is based on the assumption that a given graph (representing a linear list) has to be pebbled. Our results can be seen as strengthening these earlier investigations, since we consider only the output-behaviour of the program being simulated and not its internal details.

String matching by read-only programs. The string matching problem (over a finite alphabet) asks for deciding whether a given pattern string p occurs in a text t over a fixed alphabet — without loss of generality, $\{0,1\}$; formally, to recognize the set of strings (or lists of symbols)

$$M = \{p\#t \mid p, t \in \{0,1\}^*, \quad t = t_1 p t_2\}.$$

The obvious method for locating p in t tries to match p against successive portions of t. More precisely, each position in t is considered as a possible start-point of p. Clearly the last $|p| - 1$ positions are excluded, since p cannot match a suffix of t shorter than itself. At all other positions p and t are checked symbol-by-symbol. For input of length $n = |p| + |t| + 1$ this approach is of time complexity $O(n^2)$, and can be implemented by a read-only program. Assuming that the pattern is scanned in a left-to-right fashion, on input $0^k 1 \# 0^{2k}$ the algorithm requires $\Omega(k^2)$ steps showing its worst-case running time to be $\Theta(n^2)$.

Most of the faster well-known string matching algorithms (e.g., [7]) use storage proportional to the size of the pattern for saving the results of a preprocessing phase. Thus they cannot be implemented as a read-only program. An exception is the constant-space, linear-time string-matching algorithm due to Galil and Seiferas [4]. However, this algorithm makes use of pointers that can be moved both ways. Considering this background, Jones [6] suggested the string matching problem as a candidate application for which the relative inefficiency of read-only programs over one-way lists might be demonstrated.

Our contribution to answering this question is in fact an upper bound, which follows directly from [4], using our simulation:

Corollary 1 *For every $k > 0$, string matching can be done by a read-only program in time $O(n^{1+1/k})$.*

Other applications. The obvious application of our results is to processing a set of data that can only be traversed in the forward direction, possibly due to a limited programming interface to the data set. Another application is navigating in an implicit list, such as a pseudo-random sequence. A typical pseudo-random generator allows for fast production of the next number from the current one, but it may be infeasible to directly compute a predecessor.

Structure of this paper. The algorithm that achieves the amortized-time upper bound is described in Section 3, followed by a section on lower bounds. Section 5 gives the worst-case time upper bound, also followed by a lower-bound section. The paper concludes with a discussion of extensions and open problems.

A preliminary exposition of this work was presented at the Thirty-First Annual ACM Symposium on the Theory of Computation (STOC), 1999.

2 Our programming model

The programs we investigate are written in a simple imperative language without procedure recursion, a "flow chart language." A program has a finite number of variables, some of which act as pointers into a linear input list. The list cannot be modified. The pointer variables can only be modified by assignment operations (placing in one variable a copy of the contents of another) or by advancing a pointer to the next element in the list. Initially all pointer variables point to the head of the list. A pointer moved off the list obtains a special value `nil` that can be recognized by the program. The set of instructions could be enhanced by a "rewind"-operation that moves a pointer to the beginning of the list. Such an instruction can be simulated with the help of a pointer kept to the beginning of the list. Since this causes no time overhead and requires only one extra head we will not use a "rewind"-operation.

The only other type of variable we allow at this stage, other than a pointer, is the bounded counter. Such a variable assumes integer values in the range $[0, n]$ where n is the length of the input list. The operations applicable to counters are assignment, increment, decrement and equality test. Its initial value is 0. In Section 3.3, we discuss the effect of restricting the counter operations to a smaller set, or avoiding the use of counters altogether, a restriction that yields pure pointer programs.

Our goal is to compare the efficiency of programs in the above language to that of programs in a richer language, that includes an operation for moving a pointer to the preceding position in the list.

Naturally, both versions of the programming language would also include some facilities for computing with the list contents; reading the datum under a pointer and possibly manipulating it. A precise specification of these facilities is not important at this stage, however we assume that the program can retain the value last read by each of its pointers. This means that there is no need to dereference a pointer repeatedly if it's not being moved.

In our discussions we associate the directions "left" and "right" with, respectively, the backward and forward directions in the input list.

3 The amortized-time upper bound

The purpose of this section is to prove the following:

Theorem 1 *Let $k > 0$. Every read-only program with pointers that admit two-way movement can be simulated by a program with one-way pointers, such that if the original program's run time over a given input of length n is $t(n)$, the simulator runs in $O(n + n^{1/k}t(n))$ time.*

If the simulated program has d pointer variables, the simulator uses $(k-1)d + 2$ pointer variables and $kd + O(1)$ counters to simulate them.

The Simulation Algorithm. We initially compute an approximation to $\sqrt[k]{n}$ in a counter variable r (the "radius"); the details of this computation are deferred to the next subsection. We assume that n is big enough so that $r \geq 2$; a small n can be treated as a special case. Now, let v be any pointer variable of the simulated program. We associate with v a set of pointer variables v_j, $0 \leq j \leq k-1$. In addition, we associate with v a set of counters, described below. Two additional variables used by the simulator are h, storing a pointer to the head of the list (that never moves), and t for temporary usage.

We now describe how the pointers associated with v are used to simulate its movement. As it turns out, the variable v_0 does not need to be explicitly maintained, although it is convenient for the explanation to assume that such a variable exists. Therefore, the actual number of pointer variables in the simulator is $(k-1)d + 2$.

The role of v_0 is to simulate v exactly, pointing always at the same position in the list as v does. The role of v_j, for $j > 0$, is to mark the beginning of an interval of $5r^j$ positions, called the *level j interval.* This interval contains the position of v_{j-1} (in the sequel we will say, metonymically, "the

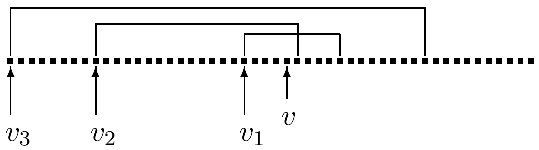

Figure 1: Pointers maintained by the simulator and their associated intervals. Every interval contains the location of the next (inner) pointer, but the intervals are not necessarily nested.

interval contains v_{j-1}"). For uniformity of notation we use v_k as an alias for h; so there is also a level k interval, beginning always at the list's head. Figure 1 shows a possible situation involving three levels of intervals.

For each $0 \leq j < k$, we dedicate a counter L_j^v to maintaining the distance of v_j from v_{j+1}. This distance will always be a multiple of r^j, and L_j^v stores the actual distance divided by r^j. Thus, L_j^v never exceeds $5r-1$. Its initial value is 0, as all pointers initially point at the head of the list.

Simulating pointer operations. Recall that v_0 is not actually maintained. Therefore, to simulate access to the list contents using v, we set the temporary variable t to v_1 and move it L_0^v positions forward, at a cost of $O(r)$. It is natural to assume that once the program learned the value stored in the list cell, there is no need to read it again, so this cost has to be incurred only once after a step in which v moved.

Simulating a move of v, either forward or backward, amounts in the most common case to just incrementing or decrementing L_0^v. Special cases arise at the ends of the interval; namely backing up when $L_0^v = 0$ or moving ahead when $L_0^v = 5r-1$. In such cases we shift one or more of the enclosing intervals in the appropriate direction so that v_0 will be in the center of the interval again.

We now describe procedures for shifting the interval of level j (for any $0 < j < k$) by $2r^j$ positions precisely, either forward or backward. To shift the interval forward, we simply advance v_j while counting from $2r^j$ down to 0 (see the next subsection for more details on the arithmetics). We also add 2 to L_j^v. If it exceeds $5r-1$, we trigger a right-shift of v_{j+1} as well. The latter will be moved $2r^{j+1}$ positions, so L_j^v will be decreased by $2r$. This brings it again into the permitted range. Note that under these operations, the value of L_j^v stays always even. We will maintain this invariant in the following operations as well.

A shift of the level j interval backward is done as follows. If $L_j^v > 1$, we reset v_j to v_{j+1} and shift it forward $L_j^v - 2$ times r^j positions. If $L_j^v \leq 1$ we have to shift the level $j+1$ interval first. This causes L_j^v to increase by $2r$, and allows the operation to be completed.

The time to shift a level j interval (shifts triggered at other levels not included) is obviously $O(r^j)$ for the forward direction and $O(r^{j+1})$ for shifting backward. We now demonstrate that this time amortizes to $O(r)$ per original program step.

Lemma 1 *Starting at either the beginning of the simulation, or a point in time when a shift at level j is completed, at least $2r^j$ steps that move the originally stored in v have to be performed before another shift at level j is necessary.*

Proof. This is proved by induction on j. For $j = 1$ the proposition is easy to verify. For $j > 1$, a shift at level j is triggered by v_{j-1} crossing one of the ends of the level j interval. Consider the situation when such a shift is completed; v_{j-1} then lies in the middle fifth of the level j interval. It is $2r^j$ positions away from either end of the interval, so v_{j-1} will have to cover this distance before another shift at level j is necessary. As a shift at level $j-1$ moves v_{j-1} a distance of $2r^{j-1}$, we see that r such shifts will have been performed before the next shift at level j. By induction, every shift at level $j-1$ comes after at least $2r^{j-1}$ steps moving v. Thus the next level j shift will come after at least $r \cdot 2r^{j-1} = 2r^j$ such steps.

The argument regarding the period before the very first shift is similar and we omit the details. □

The lemma shows that the *amortized* time for shifts at every level is $O(r)$.

Apart from accessing v and moving it forward or backward, the simulated program may also assign the value of v to another pointer variable u (assuming $d > 1$). In the simulator we handle this as follows:

1. For $j = 1, \ldots, k-1$ check whether $L_{j-1}^v < r$ or $L_{j-1}^v \geq 4r$; this indicates that v_{j-1} is at the first (respectively last) fifth of the level j interval. If this is the case, shift v_j backward (respectively forward) a distance of $2r^j$ and replace L_j^v with $L_j^v - 2$ (respectively $L_j^v + 2$). The updated value of L_j^v will be used to decide if a shift is needed at the next level.

2. Now copy the variables v_j and L_j^v into u_j and L_j^u, respectively, for all $j < k$.

The purpose of the first step is to ensure that the complexity analysis is not hampered by assignments. If a pointer variable v is copied into one or more other variables u_i, we want to apply the amortization argument to every such u_i separately. Indeed, the procedure guarantees that counting from the assignment operation, at least r^j steps *which move* u must occur before we need to shift u^j again. In a similar manner we can verify that the additional shift introduced for the assignment does not affect the amortized cost associated with v too much.

3.1 Approximating roots

The setup phase of the simulation we described requires a time-efficient procedure for computing r, an approximation to the kth root of n.

We assume now that the value of n is known or has been found by traversing the list once.

It is well-known that $\lfloor \log_2 n \rfloor$ can be computed by successively dividing the input length by 2 and counting the number of iterations until it is reduced to 0. For reasons to be elaborated later, we are interested in implementing the computation using a minimal set of arithmetic operations, namely decrement and comparison to zero. Using these operations, division by 2 takes time linear in the value divided. Hence, computation of the logarithm takes

$$O(n + \frac{n}{2} + \frac{n}{4} + \cdots) = O(n)$$

time.

Next, we compute, in $O(\log n)$ time (using decrements), the value $y = \lfloor (1 - 1/k) \lfloor \log_2 n \rfloor \rfloor$. We set $r = \lceil n/2^y \rceil$; it can be computed in linear time by repeatedly dividing n by 2 while counting from y down to zero. We have

$$\begin{aligned} \lceil n/2^y \rceil &\geq 2^{\log_2 n - y} \\ &\geq 2^{\log_2 n - (k-1)/k \cdot \lfloor \log_2 n \rfloor} \\ &\geq 2^{(\log_2 n)/k} \\ &= \sqrt[k]{n} \end{aligned}$$

and

$$\begin{aligned} \lceil n/2^y \rceil &\leq 2^{\log_2 n - y} + 1 \\ &\leq 2^{\log_2 n - (k-1)/k \cdot \log_2 n + 2} + 1 \\ &= 4\sqrt[k]{n} + 1. \end{aligned}$$

Therefore $r = \Theta(\sqrt[k]{n})$. Note that the inequality $r \geq \sqrt[k]{n}$ guarantees that the interval system described above covers the list without any "leftovers." In the algorithm, we also use constant multiples of r, such as $2r$ and $5r$; these too can be computed in $O(n)$ time using decrements only (first compute $x = n - 2r$ and then $n - x$, etc.).

When shifting intervals, we have to count from r^j down to 0, for some $j < k$. However, direct computation of r^j is unnecessary, as we can count in base-r notation by means of j counters that count down from $r - 1$ to 0. The counting process takes $O(r^j)$ time, so it has no effect on the time bounds of the simulation.

3.2 Eliminating the setup phase

Using the above method for computing r, the setup phase takes $\Theta(n)$ time. This can be disregarded if the total time to simulate a program is considered, and the original program at least scans the input. The setup phase can be reduced to constant time if the programming language includes sufficient unit-cost arithmetic operations and the value of n is provided (so that the program does not have to scan the input for determining its length).

Using a well-known technique for adjusting bounds dynamically, it is also possible to reduce the setup phase to constant time when n is unknown in the beginning of the simulation. This change may be useful if we intend to apply the technique to lists whose length cannot be determined in advance. Such an example could be the pseudo-random sequence application, when the number of elements to be generated is not known before the program runs.

Let us briefly describe the modifications required for achieving this result: initially, r is set to a constant (say 2). The simulation proceeds without change as long as there is no need to shift the outermost interval. If such a shift is triggered, this indicates that we underestimated the length of the list, so at this point we replace r by $2r$. As a result, each auxiliary counter L_j^v has to be divided by 2^j. The division takes $O(r)$ time using decrements. If the division is not even, v_j will be shifted forward to round up the fraction. This restructuring preserves the interval structure, and its cost is $O(r^k)$. We notice that the modifications affect the amortization argument; in fact, the time until the next shift at every level may be reduced. Therefore, we bound the cost incurred by a restructuring by the cost of shifting all intervals backward once. This cost is $O(r^k)$, with r denoting the value before the restructuring. We amortize this cost over forward steps; after each restructuring, at least r^k forward steps have to be taken until the next restructuring is triggered. Its amortized cost is therefore $O(1)$.

Remark: we have referred to a single simulated variable v. If $d > 1$ pointers are being simulated, the restructuring should be performed for all of them at once, since the value of r is shared.

3.3 Pointer arithmetic

In this subsection we show that the upper bound given by Theorem 1 also holds in the most restricted model we consider, that of *pure pointer programs*. These programs have pointers as the only type of variables. Therefore, to implement the algorithms of the previous section, we have to simulate counter variables by means of pointers. We recall some known techniques performing such simulations (e.g., [5]). We will show that the simulations are efficient enough for our purposes.

Number the nodes of the input list in descending order, from n to 1. We simulate a counter of value i by a variable pointing at node i; a variable holding `nil` encodes the value 0. Clearly, testing for a zero value takes constant time, and so does decrementing an encoded counter value by one. These operations suffice to implement the computation of r described in Section 3.1 in time $O(n)$ (we do not attempt to exclude the setup phase in this model).

Once r has been computed, we make use of pointer variables to represent r-bounded counters, encoding values between 0 and r. Such a variable can be reset to r at any time by copying from a variable dedicated to maintaining r. It can be decremented and compared to zero in the usual way. Such a counter can also be *incremented* in $\Theta(r)$ steps. First the value i is replaced with $r - i$ by decrementing a copy of r in parallel with a copy of i, until the latter hits zero. Then the value is decremented giving $r - i - 1$. Finally $r - i - 1$ is replaced with $i + 1$ in the same way as before. Two r-bounded counters, i and j, can be compared in $O(r)$ steps by decrementing copies of i and j in parallel until they reach zero.

The main use of counters in the algorithm of Theorem 1 is to maintain the distances between pointers, that is, the values L_j^v. Each such value lies between 0 and $5r$, and can therefore be conveniently represented in base-r notation by a pair of counters, one bounded by 5, the other by r. The 5-bounded counter can also be maintained in the finite control, thus saving a pointer variable. It is easy to see that decrement, increment, and comparison, as well as adding or subtracting r, can be done in $O(r)$ time; these are all the operations needed for implementing the algorithm, and the overhead caused by implementing the counters this way amounts to $O(r)$

per simulated step, maintaining the stated time bound. We thus have

Theorem 2 *Let $k > 0$. Consider a read-only pure pointer program with pointers that admit two-way movement over a list of length n, running in time $t(n)$. The program can be simulated by a pure pointer program with one-way pointers in time $O(n + n^{1/k}t(n))$.*

4 The amortized-time lower bound

Our goal in this section is to show that for simulators with a finite number of pointer variables and counters, every simulation leads to a polynomial overhead. Moreover, the tradeoff between the overhead and the number of pointer variables in the simulator, exhibited by Theorem 1, is optimal up to a difference of a single variable.

Our proof pertains to a simulation which is *semi-on-line*. This term means that if the source program produces a sequence of output data, the simulator has to produce the same sequence at a speed which corresponds to the source program's speed, up to a given *simulation overhead*. A fully on-line simulation is one where output operations may be interspersed with input operations. This stronger requirement was necessary for some previous results of a nature somewhat similar to ours, for example separating Turing machines with different number of heads [11], or demonstrating the efficiency of "mutable" linked structures [12].

Obviously, we assume that the programming language includes an output operation. This operation appends an element (of an appropriate type) to the output sequence. In fact, we shall assume that the type of output data is the same as the type of data stored in the nodes of the input list. Note that these data do not play a role in our simulation algorithm, but they are important for defining the extrinsic behaviour of the simulated program; our notion of a simulation only requires that this behaviour be duplicated by the simulator.

Definition 1 *Consider a program* `p` *that, given an input list I, produces the first m output values within a time span of $t_I(m)$ steps (not necessarily starting at the beginning of the run). A program* `q` simulates `p` semi-on-line with overhead bounded by *$b(n)$ if, for every list I,* `q` *produces the same output as does* `p`*; moreover,* `q` *produces the first m output data within $b(n) \cdot t_I(m)$ steps, where n is the length of the list.*

Note that the time-spans compared have to contain all the output operations, but may exclude a "setup phase" before the start of the output sequence.

To obtain a lower bound on $b(n)$, we choose a program that performs a task using 2-way pointers, and prove a lower bound on the speed of any 1-way pointer program performing the same task. Our choice of the "hard to simulate" program is quite natural: it is a program that outputs the contents of the list in reverse order. Obviously, accessing a list in reverse order is difficult with 1-way pointers, but the above definition takes a "black box" approach in that the simulator only has to produce the correct output sequence — no requirement is made on its internals. The tool which allows us to draw conclusions on the internal behaviour of the program from the output it produces is *incompressibility theory* [8]. Classically, this theory is applied to strings over a finite alphabet, so we too will prove our lower bound for programs whose list data are symbols from a finite alphabet (we remark that assuming an infinite alphabet, as for example in [12], would lead to a weaker result with a much easier proof).

4.1 Programs using a finite alphabet

We now assume here that the elements stored in the list as well as the output symbols are from a fixed alphabet, say $\{0, 1\}$. We allow the simulator to use both pointer variables and counter variables, where counter values lie between 0 and n, the length of the input (in fact a bound polynomial in n can also be allowed). We do not restrict the kind of operations that may be applied to counter values. This makes the model of the lower bound stronger than that of the upper bound.

In the proof of the lower bound the following lemma is the key result. Its proof uses the fact that there are *incompressible strings* of every length [8]. Briefly, these are strings that cannot be reconstructed from a shorter string using a fixed decoding algorithm.

A *self-delimiting* encoding $\hat{x}$ of a string x allows to recover x from any string $\hat{x}y$. One way to obtain such an encoding is to form $1^{|x|}0x$. This string has length $2|x| + 1$.

By $\log n$ we mean, in this proof, the base-two logarithm of n rounded up to an integer.

Lemma 2 *Let the read-only program P with k variables (either pointers or counters) operate on input in $\{0, 1\}^*$. Let $s = |\hat{P}|$, the length of a self-delimiting encoding of the program text. Let w be an arbitrary input word* *and let x be a portion of w of length p where $p > s + \log s + (k+4)(\log |w| + 1)$. If P outputs the reversal of x during any part of its execution on w, and while doing so never has a pointer pointing into x, then w can be compressed.*

Proof. The following information will be concatenated in order to give a compressed encoding of w.

- The encoding of P's program text in s symbols,
- the length of w in self-delimiting binary encoding ($\leq 2\log |w| + 1$ symbols),
- the position of x in w ($\log |w|$ symbols — note that a fixed-length encoding can be used now since $|w|$ has been specified),
- the length of x ($\log |w|$ symbols),
- the number of the instruction to be executed at the moment when P starts to output the reversal of x ($\log s$ symbols),
- the values of P's variables at this moment, where pointer values are specified as positions in the list ($k \log |w|$ symbols),
- w with x deleted ($|w| - p$ symbols).

The following *decoding algorithm* recovers w from the above data.

- Allocate a list u of length $|w|$. Store the $|w| - p$ symbols of w that are provided into their proper positions, skipping the p positions representing x (those can be filled arbitrarily).
- Prepare to simulate P by setting its variables to the provided values; pointer variables will be set to the appropriate positions in u.
- Simulate P starting from the provided instruction number, until p output symbols have been produced.
- Insert the output symbols, in reverse order, into u at the portion representing x.

Note that the simulation of P over the list u is guaranteed to proceed identically to running it on w, since there will never be a pointer into the portion where u and w may differ. Thus, the algorithm restores w correctly.

According to the assumption of the lemma, the length of the description used to reconstruct w is less than $|w|$. Hence w is compressible. □

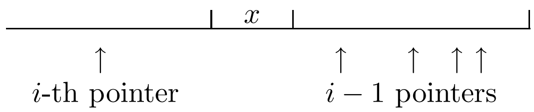

Figure 2: The induction step in the proof of Theorem 3.

Theorem 3 *There is a read-only program R over two-way lists such that any read-only program over one-way lists that simulates R with overhead $O(n^{1/k})$ must use at least k pointer variables.*

Proof. Program R receives a linear list w of n binary digits, and outputs its reversal. This obviously takes linear time, and in fact once output is initiated the reversal is produced in constant time per symbol using a single pointer. We now consider a read-only program P which using one-way pointers reverses a list with overhead bounded by $O(n^{1/k})$. By Definition 1, this means that for some integer constant c, program P takes at most $cn^{1/k} \cdot m$ time to produce m output symbols counting from the first output operation. The part of the run that starts at the first output operation will be called the *reversal*.

Let $s = |\hat{P}|$ as in Lemma 2. Let k_P be the number of variables in P (pointers as well as bounded counters). Our purpose is to show $k_P \geq k$. Choose natural numbers n, p such that $p > (2c+2)^{k-1}$, $n = p^{2k}$ and $p > s + \log s + (k_P + 4)(\log n + 1)$. Let w to be an incompressible string of length n. Note that these choices satisfy the assumptions of Lemma 2.

We will prove inductively that in certain regions of the input at least one pointer has to be located at the start of the reversal. More precisely, our inductive claim is the following: when reversal starts, at least i pointers must be located in the suffix of the input of length $(2c+2)^i p^{2i+1}$.

For $i = 0$ this statement holds trivially. To prove the claim for $i \geq 1$ we focus on w's segment x of length p preceding the suffix of length $L_i = (2c+2)^{i-1}p^{2i-1}$. By the inductive assumption, the latter suffix contains at least $i - 1$ pointers when reversal starts. The segment x satisfies the conditions of Lemma 2; therefore its reversal cannot be output without some variable pointing into it. Reversal of the suffix including x has to be completed in at most $cn^{1/k}(L_i + p) = cp^2(L_i + p)$ steps. Since pointers to the right of x cannot move left, a variable that initially points inside x or to the left of x has to supply the pointer into x for its reversal (Figure 2). The initial distance of this pointer from x cannot be larger than the above time bound. Thus there are at least i pointers in a suffix of length

$$\begin{aligned} & cp^2(L_i + p) + L_i + p \\ & \qquad = (cp^2+1)(2c+2)^{i-1}p^{2i-1} + (cp^2+1)p \\ & \qquad \leq (2cp^2+2)(2c+2)^{i-1}p^{2i-1} \\ & \qquad \leq (2c+2)^i p^{2i+1}. \end{aligned}$$

By the choice of p we have $(2c+2)^{k-1}p^{2k-1} \leq (p-1)p^{2k-1} \leq n-p$. Hence, our inductive claim shows that at least $k-1$ pointers have to be positioned, a reversal begins, on a suffix of length $n-p$. Another pointer is necessary for the prefix of length p, completing the proof. □

The next theorem shows how the number of pointers needed in the simulator relates to the number of pointers in the simulated program, in addition to the simulation overhead. It shows that the number of pointers used by the simulation shown in Theorem 1 is optimal for any number of simulated pointers, give or take one pointer variable.

Theorem 4 *For every $d > 0$, there is a read-only program R_d over two-way lists using d pointers, such that any read-only program over one-way lists that simulates R with overhead bounded by $O(n^{1/k})$ requires at least $(k-1)d+1$ pointer variables.*

Proof. Program R_d receives a linear list w of $n = dm$ binary symbols. Consider this input to be a row-major listing of a $d \times m$ matrix. The program outputs the columns of the matrix from the rightmost column backward. This can be achieved as follows: first, a counter is set to $m = n/d$ (at the same time, R_d can verify that d divides n, and halt otherwise). Then d pointers are positioned at distances of $m, 2m, \ldots, dm$ from the head of the list. All of this takes linear time. We remark that the extra counter can be avoided if the end of the list is marked using a third symbol; at any rate, our main interest here is in accounting for pointers. After the preparation, the program repeatedly outputs the symbols under the pointers in a round-robin fashion while moving them backward.

Let P be a program simulating R_d with one-way pointers with overhead bounded by $O(n^{1/k})$. Note that if we look only at the outputs belonging to the ith row of the matrix, for a specific i, we see that P reverses a list of length m with overhead bounded by $O(d \cdot n^{1/k}) = O(m^{1/k})$ (as d is a constant). Thus, the proof of the previous theorem can be used for P, showing that when reversal starts, at least $k-1$ pointers must be located in a suffix of row i whose length is at most $m - p = m - m^{1/2k}$, regardless of the locations on pointers on other rows. In total, this accounts for $(k-1)d$ pointers that have to be positioned in distinct locations at the start of the output by P. An additional pointer is necessary for the prefix of length p of the first row, completing the proof. $\square$

We remark that the lower bound proofs do not assume any bound on the running time of the initial phase before output is begun. Thus, while our simulator used linear-time setup, the simulation could not be accelerated with a longer setup phase.

4.2 The lower bound with richer data types

Computational models intended for the description and analysis of practical algorithms often allow the program to use data types richer than symbols or bounded counters. Typical examples are integers of a given "word length" that may be larger than $\Theta(\log n)$, unbounded integers and real numbers. It is also frequently assumed that the program can apply a given set of primitive operations to these data, such as arithmetic operations, at unit cost.

Data structure algorithms are often described in a way oblivious to the kind of data actually stored within the structure. For example a linked list can include in each cell an "item" of an arbitrary data type. Our simulation algorithm is in fact oblivious in this way. However if one assumes that the program has the ability to use a stronger data type, a lower-bound proof has to take that into account; obviously, a contrived data type with unit-cost operations can invalidate a lower bound. Quite naturally, the lower bound proofs of this section generalize to richer data types if they exhibit an appropriate incompressibility property, corresponding to the incompressibility of strings. The tradeoff between simulation speed and the number of pointer variables used will, in such cases, be the same. Since this topic diverges from the main theme of our paper, the details appear in an appendix.

5 The worst-case time upper bound

In this section we show a simulation of two-way access to the list using one-way pointers, whose overhead *per operation* is bounded by $O(n^{1/k})$. The idea is to modify the algorithm of Section 3 in such a way that the overhead is evenly distributed, thus avoiding operations that are slowed down by a factor exceeding $O(n^{1/k})$. The simulation algorithm in this section is more complicated and uses more pointer variables (in fact part of the complication results from our goal to minimize the number of additional pointers). In the next section, we will show that the increase in the number of pointers cannot be avoided.

Theorem 5 *Let $k > 0$. Every read-only program with pointers that admit two-way movement can be simulated using one-way pointers with a worst-case time of $O(n^{1/k})$ per simulated operation, where n is the length of the input list, after a linear-time setup phase.*

The simulating program uses $(k-1)(d+1)+2$ pointer variables to simulate d original pointers. Apart from pointers, it makes use of $O(kd)$ bounded counters. By implementing counters using pointers, the same running time can be achieved with a pure-pointer program.

As a first stage in proving this theorem, we show how to simulate a program that does not use pointer assignment, so that pointer variables move independently of each other.

5.1 Real and Imaginary Pointers

The presentation of the simulation algorithm will be simplified using a convenient generalization of a pointer variable. A *pointer object* represents a position that can be in the list (a *real* pointer) or out of the list (an *imaginary* pointer). We only use imaginary pointers to represent negative positions, i.e., before the head of the list. We number the positions of the list 0 to $n-1$, so that "negative positions" are represented by negative numbers. The position of an imaginary pointer will be between $-n$ and -1, so it can be represented by a single n-bounded counter. Thus, a pointer object is implemented as either a real pointer or a counter, plus a flag which tells which is the case.

An imaginary pointer automatically turns into a real one if it reaches the head of the list. For this purpose, the program reserves one fixed pointer to the head of the list that can be assigned to the pointer variable of the pointer object.

In the following algorithms all references to "pointers" actually refer to pointer objects unless stated otherwise.

5.2 Simulating programs without assignment

The simulator dedicates k pointers to simulate every original pointer variable v, denoted as usual by $v_1, \ldots, v_k$. Of these pointers, v_k will always be imaginary, so the number of pointers actually used for this purpose is $d(k-$

1). Additional $k-1$ pointers are used by the *coroutine* mechanism which we explain below. We also need one temporary pointer and a fixed real pointer to the head of the list so that in total we need $(d+1)(k-1)+2$ pointer variables.

As in Section 3, the program starts with a setup phase in which the "radius" r is computed. If we have d original pointers to simulate we choose r to be an even integer approximately equal to $n^{1/k}$, so that $r^k \geq n$. We assume n to be large enough so that $r > 30d$ (this bound is convenient in our analysis but can certainly be reduced). The role of v_j is to mark the beginning of an interval of $4r^j$ positions, containing the position of v_{j-1}. This is the *level j interval*. As in Section 3, it is convenient to use the name v_0 for the position of the simulated pointer v, even though we do not dedicate a variable to this purpose.

Our goal is to maintain the invariant $0 < L_j^v < 4r$ for each $0 \leq j < k$, where the L_j^v are like in Section 3. By keeping the last datum seen by v in the program's memory, we avoid having to access the position as long as the pointer doesn't move. We next describe how movement of v is simulated.

Initial state. As the program begins, v is at the head of the list. All v_j are set up to its left (as imaginary pointers) at growing distances so that $L_j^v = 2r$. Setting up the corresponding counters is part of the linear-time setup phase and can be done even with restricted counter operations, in the manner of Section 3.1.

Pointer movement. To simulate a movement of v, either backward or forward, we adjust the counter L_0^v. To maintain the invariant $0 < L_0^v < 4r$, we introduce a *coroutine mechanism* that shifts v_1 in due time. This idea resembles the solution in Section 3, except that here the shifting of v_1 will begin *before* the invariant is about to be broken. The same mechanism will be used to shift v_2 if necessary for keeping the constraints on L_1^v, and so on.

A *coroutine* is a procedure whose thread of control is parallel to that of the simulator's main program. The simulator has $k-1$ coroutines. After every pointer movement simulated, the simulator executes a step (described below) on behalf of each active coroutine, and also checks whether a previously inactive one needs to be activated. Each coroutine step takes $O(r)$ time.

The role of the coroutine for level j, with $0 \leq j \leq k-2$, is to shift intervals at level $j+1$ backward, ensuring the constraints on L_j^v are never broken. Note that there is only one level j coroutine, serving all the simulated pointers.

Thus, v should be understood here as a generic variable name, ranging over the set V of simulated pointer variables. A more precise formulation of the coroutine's role is: the coroutine at level j ensures the constraints on L_j^v for all $v \in V$.

A coroutine at level $k-1$ is not necessary. To see this, note that the rightmost possible location of v_{k-1} is at the end of the list, which is at most $3r^k$ positions away from the initial location of v_k. The leftmost possible location of v_{k-1} can be computed by assuming that $L_0^v, \ldots, L_{k-2}^v$ are all as big as possible, while v is located at the left end of the list. In this situation, the location of v_{k-1} is still above $-2r^k$.

We now fix $0 \le j < k-1$ and describe the behaviour of C_j, the coroutine for level j.

C_j is only resumed if there is a v such that $L_j^v \le r$ or $L_j^v \ge 3r$. We say that in this case v_j deserves attention, because it is close to one of the ends of its enclosing interval. The simulator checks for this condition whenever the coroutine is inactive and some L_j^v is modified (L_0^v is modified by simulated pointer movements, while for $j > 0$, L_j^v is only modified by the level $j-1$ coroutine). Once the coroutine is activated *on behalf of* v, it will spend a certain number of steps servicing this pointer. In the meantime, other level j counters can keep changing, and therefore once the coroutine completes its activity, the simulator may find that there is now more than one simulated v such that v_j deserves attention. The coroutine is then invoked on behalf of the most deserving pointer: namely the one closest to the end of its interval. We later use a potential-function argument to prove that this strategy ensures that no pointer will ever exit its interval.

During its activity, the coroutine makes use of a dedicated auxiliary pointer (hence the total of $k-1$ such pointers for all the coroutines). This variable is used to locate the future (shifted) position for v_{j+1}. For a forward shift, this means moving forward from v_{j+1}; for a back-shift, we move forward from v_{j+2}. Upon invocation, the coroutine initializes this variable and sets a counter (also dedicated to the coroutine) to the distance to be traveled: for a forward shift, it is r^{j+1}. For a backward shift, it is $(L_{j+1}^v - 1)r^{j+1}$. In every coroutine step, the pointer is moved $5r$ positions (less if the count is exhausted), so that the process is completed in at most

$$\frac{(4r-1)r^{j+1}}{5dr} < (4/5)d^{-1}r^{j+1} \tag{1}$$

steps of $O(r)$ time each.

Upon completion of its action, the coroutine sets v_{j+1} to its new location, increments (or decrements) L_{j+1}^v appropriately, and replaces L_j^v with $L_j^v - r$

(respectively, $L_j^v + r$). All this is done in a single $O(r)$-time step after which the coroutine becomes inactive. We remark that the value of L_j^v can differ at this point from its initial value when the coroutine was activated. It is not hard to see that the update (adding or subtracting r) is nonetheless correct.

Recall that V denotes the set of original pointer variables, $|V| = d$. We use a parameter t to denote time, measured by simulated pointer movements (any other operations performed by the simulated program are immaterial to the simulation algorithm). Let $L_j^v(t)$ denote the value of counter L_j^v at time t. We define

$$\begin{aligned} \phi_j(t) &= (d-1)\min_{v\in V} L_j^v(t) \\ \Phi_j(t) &= \phi_j(t) + \sum_{v\in V} \min(L_j^v(t), 2r) \end{aligned}$$

as auxiliary functions. By means of these functions the counters for all variables are combined and can be bounded. From this "compound value" properties of the individual counters can be derived.

Further, let $v_j(t)$ denote the position of v_j at time t.

Lemma 3 *For every $0 \le j < k-1$, the following statements hold:*

(i) For any given times t_0 and $t_1 > t_0$, $\sum_{v\in V} |v_j(t_1) - v_j(t_0)| < t_1 - t_0 + 6r^j$.

(ii) If C_j is started at time t, we have $\Phi_j(t) \ge (2d-1)r$.

(iii) For all $v \in V$, if C_j is started at time t, then $L_j^v(t) \ge d^{-1}r$.

(iv) For all $v \in V$, L_j^v never drops below $d^{-1}r/2$.

(v) For all $v \in V$, L_j^v never surpasses $4r^{j+1} - d^{-1}r/2$.

Proof. The lemma is proved by induction on j, but most of the arguments are the same for the base case $j = 0$ and the general case. Therefore, we give the proof once, noting the difference between the cases when there is one. We prove the four claims one by one, each one relying on the previous. The fifth claim will not be proved explicitly; it is symmetric to (iv), when the distance to the end of the interval is considered instead of the distance from its beginning. Since the algorithm treats forward and backward movement in a similar way, the reader should be able to see that the proof of (v) is symmetric to the proof of (iv) and can be omitted.

We prove Part (i) first for $j = 0$. Here, v_0 is v itself so the sum to be bounded is the total distance traveled by all the pointers from time t_0 to

t_1. Clearly, it is bounded by $t_1 - t_0$. For $j > 0$, we employ an inductive argument, and use part (iv) of the lemma for $j-1$. This part shows that v_{j-1} never leaves the interval of $5r^j$ positions starting at v_j. The greatest possible sum of distances between the positions of level j pointers at t_0 and their positions at t_1 will be obtained if the level $j-1$ pointers move as much as possible (by the inductive assumption, this is bounded by $t_1 - t_0 + 6r^{j-1}$) and in addition, each v_j increases its distance from v_{j-1} as much as possible ($5r^j$). This sums up to

$$t_1 - t_0 + 6r^{j-1} + 5r^j < t_1 - t_0 + 6r^j \ .$$

Next we prove Part (ii). Recall that C_j is invoked whenever the condition $(\exists v)L_j^v \leq r$ becomes true, and will be reinvoked repeatedly until the condition ceases to hold; such a series of invocations constitutes a *run*. We prove (ii) by induction on the invocations of C_j within an arbitrary run.

The induction basis is the first invocation in a run. At this point, $\min_{v\in V} L_j^v$ is precisely r: this is true because prior to that moment, all L_j^v values were bigger than r, and they only drop in steps of 1. Therefore if t is the time of the first invocation,

$$\Phi_j(t) = (d-1)r + \sum_{v\in V} \min(L_j^v(t), 2r) \geq (d-1)r + dr = (2d-1)r \ .$$

For the induction step, we assume that at time t in which C_j is invoked, $\Phi_j(t) \geq (2d-1)r$, and let t' be the time when this invocation is completed; assuming that the run is not completed, i.e., that the coroutine is immediately re-invoked, we should show that a similar inequality holds at time t'. Recall that the coroutine completes in $(4/5)d^{-1}r^{j+1}$ steps. During this period, no v_{j+1} moves, so movement of level j pointers alone can affect the counts L_j^v. By part (i), the possible change in $\sum_{v\in V} L_j^v$ during the activity of the coroutine is bounded by $(4/5)d^{-1}r + 6$; as $r > 30d$, this is bounded by $d^{-1}r$. Since it bounds the decrease in $\sum_{v\in V} L_j^v$, we can also use it to bound the decrease in $\min_{v\in V} L_j^v$. When the coroutine completes, it increases L_j^v by r. We have

$$\Phi_j(t') \geq \Phi_j(t) - (d-1)d^{-1}r - d^{-1}r + r = \Phi_j(t) \geq (2d-1)r$$

concluding the proof for t'.

Part (iii) of the lemma follows from Part (ii). In fact, if $L_j^v(t) < d^{-1}r$, then obviously the minimal L_j^v is smaller than $d^{-1}r$. Note that this term appears in $\phi_j(t)$ as well as being a term in the summation in the definition of $\Phi_j(t)$. All the other terms in the summation are trivially bounded by $2r$. Thus

$$\Phi_j(t) < (d-1)d^{-1}r + d^{-1}r + (d-1)(2r) = (2d-1)r\,,$$

in contradiction to (ii).

For part (iv) we consider step t of the simulation. If C_j is not active (or being activated) at this step, then $L^v_j > r$. If C_j is just being activated, then (iii) shows that $L^v_j \geq d^{-1}r$. Suppose now that C_j is active, having been activated at time $t_0 < t$. Then applying Part (iii) at t_0 yields $L^v_j(t_0) \geq d^{-1}r$. Recall that the coroutine runs for less than $(4/5)d^{-1}r^{j+1}$ steps; hence

$$t - t_0 < (4/5)d^{-1}r^{j+1}\,.$$

From Part (i) we know that the position of v_j at time t is less than $t - t_0 + 6r^j$ apart from its position at t_0. Hence also

$$|L^v_j(t) - L^v_j(t_0)| < (t-t_0)/r^j + 6 < (4/5)d^{-1}r + 6$$

so

$$\begin{aligned} L^v_j(t) &> L^v_j(t_0) - ((4/5)d^{-1}r + 6) \\ &\geq d^{-1}r - ((4/5)d^{-1}r + 6) > 0\,. \end{aligned}$$

□

Part (iv) of the lemma shows that every pointer v_j does always remain within the intended level $j+1$ interval.

5.3 Simulating programs with assignments

If the simulated program copies a pointer x into another pointer y, the simulator should copy all the associated information, namely pointers v_j and counters L^v_j, into u_j and L^u_j respectively. However, without additional care, this could cause the coroutine mechanism to fail. In fact, if L^v_j is low at the time of copying, we create a situation in which too many intervals require shifting at the same time.

The solution used in the amortized-time simulation was to shift the interval just before copying it, but this takes too much time for the worst-case goal. The solution is therefore as follows. When copying the pointers and counters as above, the program also flags the fact that v_j and u_j are now equal (for every j). Whenever coroutine C_j completes, replacing some v_j with a new value (shifted to the left or to the right), the simulator checks whether there are any other pointers u such that u_j is flagged as being equal to v_j. For every such u, the simulator checks L^u_{j-1} to see whether the shift applied to v_j is also in order for u_j (for a left shift, the condition is $L^u_{j-1} \leq r$; for a right shift, $L^u_{j-1} \geq 3r$). If it is, u_j is modified; this only requires copying the new value of v_j and adjusting counters, and can be completed within the same simulation step. Otherwise, the flag for u_j and v_j being equal is turned off.

We omit a formal argument for the correctness of the above solution, that we believe to be straight-forward.

6 The worst-case time lower bound

In this section we prove that simulating 2-way pointers with an $O(n^{1/k})$ time bound on every simulated step necessarily requires more pointers than the amortized-time simulation, matching the upper bound of Theorem 5 up to a difference of one pointer. As in Section 4.1, we prove the result for programs that process lists of symbols drawn from a finite alphabet, while other data types are discussed in the appendix. For preciseness, we first give the worst-case analogue of Definition 1.

Definition 2 *Consider a program* p *that, given an input list I, produces an output sequence of length m, such that the time from output i to output $i+1$ is $t_I(i)$. A program* q simulates p semi-on-line with overhead bounded uniformly by $b(n)$ *if, for every input list,* q *produces the same output as does* p*; moreover, the time from output i to output $i+1$ by* q *is at most $b(n)\cdot t_I(i)$, where n is the length of the list.*

Theorem 6 *There is a read-only program R over two-way lists such that any read-only program over one-way lists that simulates R with overhead bounded uniformly by $O(n^{1/k})$ requires at least $2k-1$ pointer variables.*

Proof. We compare program R from the proof of Theorem 3 with a read-only program P which using one-way pointers reversing a list with overhead bounded uniformly by $O(n^{1/k})$. This means that there is an integer constant c such that consecutive output operations are performed by P at most $cn^{1/k}$ steps apart.

Let s and k_P be as in the proof of Theorem 3. Choose natural numbers n, p such that $n = p^{2k}$, $p > (3c+3)^k$ and also $p > s + \log s + (k_P+4)(\log n + 1)$. Let w to be an incompressible string with $|w| = n$.

We will analyse the computation history of the program over input w, proving that the assumptions imply that the number of pointer variables used must be at least $2k-1$. We count time during the run of the program in program steps (in fact only steps that move pointers matter). For convenience in the analysis, we assume that an output symbol is produced *exactly* every $cn^{1/k} = cp^2$ steps. There is no loss of generality in this assumption, because if the program is speedier, we can insert no-op steps into the computation history to satisfy the assumption. For uniformity, we also place the origin of the time line exactly cp^2 steps before the first output operation.

We number the positions of w in reverse order; thus position 1 is the rightmost, and in general position i is output at time $cp^2 \cdot i$. Note that due to this convention, when we refer to a group of list positions using interval notation $[a, b)$, a is its rightmost point and $b-1$ its leftmost point.

We use a pebble metaphor to describe the program access to the list. A pebble rests on a list position whenever a pointer variable points there. Pebbles move forward as pointers advance, or disappear as a result of pointer re-assignment. When two pointers address the same position, it is covered by a single pebble; thus, a pebble may split (as one of the two pointers moves). When a split occurs, we can choose arbitrarily which of the two is considered to be "the same pebble" as the one before the split; this liberty justifies later claims about a given pebble travelling from point A to point B in the list.

Clearly, the number of pebbles located on the list at any moment during the run is a lower bound on the number of pointer variables in the program.

Let $f(j) = (3c+3)^{j-1}p^{2j-1}$. Note that by the choice of p, we have $3f(k) + p < n$.

Claim 1 *For every j, $1 \le j \le k$, and for every t_0, $0 < t_0 \le cp^2(n - 3f(j) - p)$, there is a time step $t \in [t_0, t_0 + cp^2 f(j)]$ such that at least $2j-1$ distinct positions in the interval $[\lceil t_0/cp^2 \rceil, \lceil t_0/cp^2 \rceil + f(j+1))$ are pebbled at time t.*

We prove this claim by induction on j.

For $j = 1$, and $0 < t_0 \le cp^2(n - 3f(1) - p)$, consider w's subsequence x at positions $[r, r+p)$ where $r = \lceil t_0/cp^2 \rceil$. The reversal of this subsequence takes place between t_0 and $t_0 + cp^3$. Since x satisfies the conditions of the Compression Lemma its reversal cannot be output without some variable pointing into it at some time during the process. Only a pebble that is initially on x or to its left can reach a position within x during its reversal. The position of this pebble at t_0 is thus at least r, and because it must enter x before time $t_0 + cp^3$, its position at t_0 cannot be farther than cp^3

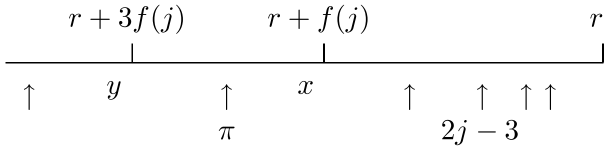

Figure 3: The induction step in the proof of Claim 1 — Case 1

from the left end of x. Hence this pebble's location at t_0 is in the interval $[r, r+p+cp^3) \subset [r, r+f(2))$. We choose $t = t_0$.

For $j > 1$, fix $0 < t_0 \leq cp^2(n - 3f(j) - p)$, and define $r = \lceil t_0/cp^2 \rceil$ as before. By the inductive assumption there is a $t \in [t_0, t_0 + cp^2 f(j-1)]$ such that at time t, $2j-3$ distinct positions are pebbled within the list segment $[r, r+f(j))$. Consider w's subsequence x in positions $[r+f(j), r+f(j)+p)$. As above, we deduce the existence of a pebble π located at time t on x or to its left. Note that we have now accounted for $2j-2$ pebbled positions at time t. To establish that one more pebble is required, we consider the exact location of pebble π at time t.

Case 1: (Figure 3) at time t, π's position is in the interval $[r+f(j), r+3f(j))$. Then we repeat the above argument, considering w's subsequence y at positions $[r+3f(j), r+3f(j)+p)$. We conclude that at time t there is, in addition to $2j-2$ pebbles to the right of y, also one on y or to its left. Reversal of y has to be completed by time $t_0 + cp^2(3f(j)+p)$, so at time t, the pebble cannot be more than $cp^2(3f(j)+p)$ steps away from y. Summing up, we identified $2j-1$ pebbles between position r and position

$$r + 3f(j) + p + cp^2(3f(j)+p) < r + f(j+1)\,.$$

Thus in this case, the claim is satisfied.

Case 2: (Figure 4) at time t, π's position is at least $r+3f(j)$. Let $t_1 \geq t$ be the time at which π first reaches point $r+3f(j)$. From this point, it still has $2f(j)-p$ positions to go before reaching x. Thus, letting t_2 be the time when π reaches x, we have

$$t_2 \geq t_1 + 2f(j) - p. \tag{2}$$

We now concentrate on the time interval $[t_1, t_1 + cp^2 f(j-1)] \subset [t_1, t_2]$. The positions of π during that period are restricted to the segment $S_1 =$

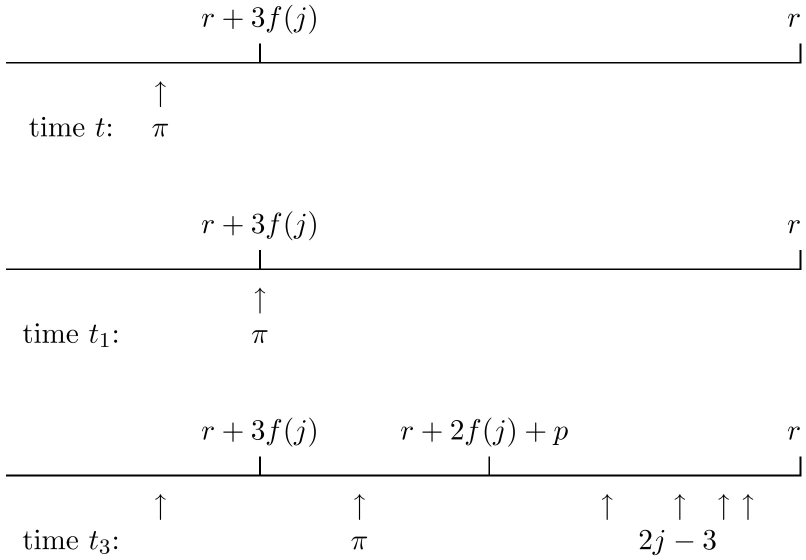

Figure 4: The induction step in the proof of Claim 1 — Case 2

$[r + 3f(j) - cp^2 f(j-1), r + 3f(j)]$. Note also that since the reversal of x must be completed before time $t_0 + cp^2(f(j) + p)$, we have

$$t_2 < t_0 + cp^2(f(j) + p) \tag{3}$$

and therefore

$$t_1 < t_0 + cp^2(f(j) + p). \tag{4}$$

Using the induction hypothesis, we apply Claim 1 to the time interval $[t_1, t_1 + cp^2 f(j-1)]$ and find a time t_3 in this interval where $2j - 3$ pebbles occupy positions within

$$S_2 = [\lceil t_1/cp^2 \rceil, \lceil t_1/cp^2 \rceil + f(j)) \subseteq [r, r + 2f(j) + p].$$

Using the definition of f, it is not hard to show that $2f(j) + p < 3f(j) - cp^2 f(j-1)$, which means that segments S_1 and S_2 are disjoint. We conclude that at time t_3, there are $2j - 3$ distinct pebbles to the right of π. It rests to identify another pebble to the left of π. To this end, we consider once again w's subsequence y at positions $[r + 3f(j), r + 3f(j) + p)$. At time t_3, reversal of this substring has not yet begun, and the above $2j - 2$ pebbles are to its right. Thus we need another one on y or to its left, reaching $2j-1$ pebbles between position r and position

$$r + 3f(j) + p + cp^2(3f(j) + p) < r + f(j+1)$$

concluding the proof of the claim in this case.

Having proved Claim 1, we apply it with $j = k$ and $t_0 = 0$, concluding that there is a time (during the reversal of the suffix of length $f(k)$) where at least $2k-1$ distinct positions are pebbled, completing the proof of Theorem 6. $\square$

Theorem 7 *For every $d > 0$, there is a read-only program R_d over two-way lists using d pointers, such that any semi-on-line simulation of R_d by a read-only program over one-way lists that has a uniform time-overhead of $O(n^{1/k})$ requires at least $(k-1)(d+1)+1$ pointer variables.*

Proof. Program R_d is the same as in the proof of Theorem 4: it receives a list w of $n = dm$ binary symbols, considered as a row-major listing of a $d \times m$ matrix, and outputs the columns of the matrix from the rightmost column backward. The time to produce every output element, once output has been initiated, is $O(1)$.

Now, let P be a program simulating R_d with one-way lists and time-overhead of $O(n^{1/k})$ per operation. Concentrate on one of the rows of length m; in R_d, these elements are output in reverse order, at $O(d)$ time per element. The simulating program has to emit these elements with a delay bounded by $O(dn^{1/k})$, i.e., bounded by $cn^{1/k}$ for some constant $c \geq d$.

The function f used in the last proof satisfies $3f(k) + p < n/d$; thus we can apply Claim 1 to the *first* row, concluding that there is a point in time, $t^* \leq cp^2 f(k)$, where at least $2k-1$ variables point into the first row. Observe that at time t^*, the reversal of all rows should stand at precisely the same point; that is, at most $1 + f(k)$ symbols of the row have been output. There still is a portion larger than $f(k)$ waiting for reversal in each of the rows. We can then apply the arguments in the proof of Theorem 3 to show that at time t^*, there must be $k-1$ variables pointing into that row[1]. We conclude that the program must use at least $2k - 1 + (d-1)(k-1)(k-1)(d+1) + 1$ distinct pointer variables. $\square$

[1]The ending comment in that proof, stating that a kth variable is also necessary, does not apply here, because that kth pointer could be initially located left of the beginning of the row, and therefore accounted for in another row.

7 Simulations with non-constant memory

Our computational model assumed that any given program has a constant number of pointer variables. However, there are natural applications for a model where a non-constant number of pointers can be allocated (though this number would still have to be much smaller than the size of the list, to keep the flavor of the problem).

This change requires a stronger programming language, which can manage this set of pointers, and possibly some associated data; the most natural model may be a program in an unrestricted language exercising restricted access to an external data object (the list). This object enforces the read-only access as well as the set of operations applicable to a "pointer."

There is no essential obstacle to using the algorithms we presented in this setting, however if k is not a constant, one should note that the simulation costs are $O(kn^{1/k})$, where we previously wrote $O(n^{1/k})$. As k grows, our upper bound thus departs from the lower bound (which stated that an $O(n^{1/k})$ simulation requires k pointers); in fact the best upper bound we obtain is $O(\log n)$, using $\log n$ pointers. Our lower bound only shows that an $O(\log n)$ simulation requires $\log n / \log \log n$ pointers, and gives no lower bound on time if $\log n$ pointers are allowed.

A related issue is reducing the cost of forward steps. It seems reasonable to ask for a simulation that does not slow down the forward steps too much in order to allow for simulating the (possibly less frequent) back-steps. Our $O(n^{1/k})$ amortized-time simulation algorithm (Section 3) actually uses only $O(k)$ time per operation when amortizing over forward steps only. This is because forward shifts are easier than backward shifts. In contrast, the worst-case solution uses $O(kn^{1/k})$ time in both kinds of steps, because the coroutines may be working on a backward shift while forward steps are being simulated. We now describe a modification to the algorithms that improves the efficiency of simulating forward steps to $O(k)$ in the worst-case.

The essence of the modification is as follows. Whenever a simulated pointer variable v moves forward, the simulator moves all the associated pointer variables v_j one step forward as well. This keeps all the distances between these pointers unchanged, and therefore no more work needs to be done for preserving the invariants of the simulation algorithms. Thus, a forward step is completed in $O(k)$ time.

In combining this idea with the *amortized-cost* algorithm we run into one difficulty, namely that the algorithm assumed v_k to be forever fixed at the head of the list. This can be easily corrected by maintaining v_k as a "pointer object" (see Section 5), located far enough before the beginning of the list.

We obtain an algorithm performing forward steps in $O(k)$ worst-case time, and back-steps as well as pointer assignments in $O(kn^{1/k})$ amortized time.

In combining the above idea with the *worst-case* algorithm, we have to take care of the coroutine mechanism; if a coroutine is servicing pointer v at the time the forward step is taken, the coroutine's temporary pointer has to be advanced together with v. Reviewing Section 5 shows that the simulation works smoothly except for the handling of pointer assignments. In fact, our modification fails if pointer assignments are to be handled as in Section 5. However, efficient ($O(k)$ worst-case time) pointer assignments *can* be combined with the above $O(k)$ worst-case forward step if we increase the number of pointers in the simulator to $2d(k-1)+2$, so that every simulated pointer can have its own coroutine at every level.

The above results have recently been improved by Matias and Porat [10]. Their algorithms simulate back-steps in $O(kn^{1/k})$ worst-case time, using k pointers per simulated pointer, while keeping the cost of forward steps at $O(1)$. They can also reverse the tradeoff, simulating a back-step in $O(k)$ time using $O(kn^{1/k})$ storage.

8 Open problems

1. As noted in the last section, our upper and lower bounds do not match when the number of pointers k is variable. In particular, for the interesting case $k = \log n$, the best known upper bound is $O(\log n)$ while our lower bound theorems do not provide any lower bound.

2. Our research was initially motivated by comments of Neil Jones regarding the weakness of programs over one-way lists (Section 1). However we have not allowed our programs to include procedure recursion, a feature considered by Jones (see [5, Sec 24.2] for details). Clearly, our lower bound proofs do not hold for recursive programs; an input list can be easily reversed in linear time using recursion. Thus, determining the gap between programs with one-way lists and those with two-way lists, in the recursive model, is yet to be done.

3. As discussed in the introduction, our simulations show that string matching can be performed by a read-only pure-pointer program (without auxiliary space) over a one-way list in time $O(n^{1+\epsilon})$, for $\epsilon > 0$. However, our lower bounds do not apply to this problem. Can it be solved in linear time by this model?

4. Another shortcoming of our lower bounds is that they rely on the requirement for semi-on-line simulation. We conjecture that a similar lower bound holds for off-line problems. A problem suggested by our proof is that of recognizing palindromes.

A Lower-bound proofs with richer data types

In this appendix we show that our lower-bound results hold for programs that process lists not of symbols, but of certain *richer* data type. A richer data type is one whose elements can be manipulated by some "arithmetic" primitives, for example the integers with the usual arithmetic operations. Of course, we assume that our programming language has the capability of maintaining a (finite) set of variables for data elements and of manipulating them. This change requires the Compression Lemma to be revised. In fact, we have to replace Kolmogorov-style incompressibility (of strings) by a more general notion which applies to some richer data types.

A.1 General incompressibility

Generalized incompressibility was defined in [1] with the intention of generalizing proofs based on incompressibility to richer data types, which is just our goal in this section. A data type is characterized in [1] as a triple $\mathcal{T} = (\mathcal{D}, \mathcal{F}, <)$ where $\mathcal{D}$ is the *domain*, $\mathcal{F}$ the set of *computation primitives* and $<$ the order relation used in comparison instructions. A computational process that operates on an n-tuple of values from $\mathcal{D}$ can often be described as an $\mathcal{F}$ *decision tree*. In such a tree, every internal node represents a decision. The node is labelled with two expressions, combining input variables with operations from $\mathcal{F}$. Comparison of the values of these expressions determines the branch taken by the computation. Every leaf node is labelled with a finite sequence of expressions, that yield the output of a computation that reaches that leaf. There is no limit to the size of the expressions or the tree as long as everything is finite.

We observe, that such trees are a (non-uniform) abstraction of programs that operate on inputs from $\mathcal{D}$ and are *genuinely time-bounded*, an expression introduced by Meyer auf der Heide [9] to indicate that the program has a time bound that only depends on the number of input elements (not on their values). To demonstrate that a function can be computed by a decision tree, it suffices to give an informal description of an algorithm that is clearly time-bounded and uses the provided computational primitives.

A data type $\mathcal{T}$ is *incompressible* if for all $n > 0$ there is no pair f_n, g_n of functions, computable by $\mathcal{F}$ decision trees, such that f_n maps $\mathcal{D}^n$ into $\mathcal{D}^{n-1}$ and g_n is its inverse.

This property was proved in [1] for the following data types:

1. Binary words of a prescribed length, with any set of operations.

2. Real numbers, with any set of operations that compute piecewise continuous functions; for example, algebraic operations, the floor function, and many other common functions.

3. Unbounded integers, with operations for addition, subtraction, multiplication, bitwise Boolean operations and restricted shifts.

Let P be a program, that operates on input x that is a sequence of elements from $\mathcal{D}$. We denote the output of P by $[\![\mathrm{P}]\!]x$.

A.2 Extending the lower bound

We now extend our class of read-only programs by allowing a program to maintain a finite number of variables of a given data type $\mathcal{T}$ to which it can apply the primitive operations of the data type, at unit cost. We also assume that the input list nodes contain elements of $\mathcal{D}$ (the domain of data type $\mathcal{T}$). The problem used to prove the lower bound is, once again, reversing the input list. In fact, the proofs of our four lower-bound theorems need not be modified except for replacing the use of the Compression Lemma (Lemma 2) by an appropriate new lemma. In this lemma we will replace the concept of an *incompressible string* by that of an *incompressible finite sequence of elements of* $\mathcal{T}$, defined next.

Definition 3 *Let E, D be two genuinely time-bounded programs (informally: encoder and decoder), using data type $\mathcal{T}$. Let $x \in \mathcal{D}^n$. We say that x is* compressed *by (E, D) if there is a sequence $y \in \mathcal{D}^m$, for some $m < n$, such that $[\![E]\!]x = y$, and $[\![D]\!]y = x$.*

If x is not compressed by (E, D) we call it incompressible *for this pair of programs.*

Note that this type of incompressibility is dependent on the choice of the encoder and decoder programs. However the following is an immediate consequence of the definitions:

Observation 1 *If $\mathcal{T}$ is an incompressible data type, then for every pair of time-bounded programs and every $n > 0$ there are sequences of length n incompressible for the given programs.*

Lemma 4 *Let $\mathcal{T}$ be an incompressible data type. Let P be a read-only, time-bounded program with k variables (either pointer variables or $\mathcal{T}$-variables) that operates on $\mathcal{D}$-lists. Let $s = |\hat{P}|$, the length of a self-delimiting encoding of the program text. Let w be a sequence of elements of $\mathcal{D}$ and let x be a portion of w of length p where $p > \log s + (k+4)(\log |w| + 1)$. If P outputs the reversal of x during any part of its execution on w, and while doing so never has a pointer pointing into x, then w can be compressed by a pair of time-bounded programs (E, D) which only depend on P.*

Proof. The decoding algorithm D is the same as in the proof of Lemma 2, except that the program P is not given as part of the description, but is fixed. This makes the decoder D time-bounded, given that P is time-bounded. Recall that the output of the decoding algorithm is the reconstructed sequence w.

The encoding algorithm runs as follows. Its input is the sequence w. First, it completely simulates P on input w while recording a complete history of this computation, including variable contents and pointer positions at each step. Then, it computes $p_0 = 1 + \log s + (k+4)(\log |w| + 1)$, and looks for the situation described in the lemma (reversal of a portion x of length p_0 while no pointers point there). Note that if such a situation occurs for $p > p_0$ than it also occurs for p_0. Once such a situation is found, the encoder produces the output described in the proof of Lemma 2, excluding the program text. As in the proof of Lemma 2, this output is shorter than w and allows for reconstruction of w by the decoder. Note that the encoder is time-bounded because P is. □

References

[1] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39:617–648, 1992.

[2] Ashok K. Chandra. Efficient compilation of linear recursive programs. In Ronald V. Book, editor, *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 16–25. IEEE Computer Society Press, 1973.

[3] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press, Rockville, Maryland, 1983.

[4] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26:280–294, 1983.

[5] Neil D. Jones. *Computability and Complexity — From a Programming Perspective*. MIT Press, Cambridge, Mass., London, England, 1997.

[6] Neil D. Jones. Must fast pattern matchers be impure? Note distributed during Dagstuhl Seminar 9823 “Programs: Improvements, Complexity, and Meanings”, June 8–12, 1998.

[7] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[8] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*, 2nd edition. Springer, Berlin-Heidelberg-New York, 1997.

[9] Friedhelm Meyer auf der Heide. On genuinely time bounded computations. Proc. 6th Symp. on Theoretical Aspects of Computer Science (STACS). Lecture Notes in Computer Science 349, 1–16. Springer-Verlag, 1989.

[10] Yossi Matias and Eli Porat. Efficient pebbling for list traversal synopses. Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP), 2003. Lecture Notes in Computer Science 2719, 918–928, Springer-Verlag, 2003.

[11] Wolfgang J. Paul, Joel I. Seiferas and Janos Simon. An information-theoretic approach to time bounds for on-line computation. *Journal of Computer and System Sciences*, 23:2 (1981), 108–126.

[12] Nicholas Pippenger. Pure versus impure Lisp. *ACM Transactions on Programming Languages and Systems*, 19:223–238, 1997.

[13] Sowmitri Swamy and John E. Savage. Space-time tradeoffs for linear recursion. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages (POPL’79)*, pages 135–142. ACM Press, 1979.