Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams

Paweł Gawrychowski[1], Oleg Merkurev[2], Arseny M. Shur[3], and Przemysław Uznański[4]

1 Institute of Informatics, University of Warsaw, Poland
2 Institute of Mathematics and Computer Science, Ural Federal University, Ekaterinburg, Russia
3 Institute of Mathematics and Computer Science, Ural Federal University, Ekaterinburg, Russia
4 Department of Computer Science, ETH Zürich, Switzerland

Abstract

We consider computing a longest palindrome in the streaming model, where the symbols arrive one-by-one and we do not have random access to the input. While computing the answer exactly using sublinear space is not possible in such a setting, one can still hope for a good approximation guarantee. Our contribution is twofold. First, we provide lower bounds on the space requirements for randomized approximation algorithms processing inputs of length n. We rule out Las Vegas algorithms, as they cannot achieve sublinear space complexity. For Monte Carlo algorithms, we prove a lower bounds of $\Omega(M \log \min\{|\Sigma|, M\})$ bits of memory; here $M = n/E$ for approximating the answer with additive error E, and $M = \frac{\log n}{\log(1+\varepsilon)}$ for approximating the answer with multiplicative error $(1 + \varepsilon)$. Second, we design three real-time algorithms for this problem. Our Monte Carlo approximation algorithms for both additive and multiplicative versions of the problem use $\mathcal{O}(M)$ words of memory. Thus the obtained lower bounds are asymptotically tight up to a logarithmic factor. The third algorithm is deterministic and finds a longest palindrome exactly if it is short. This algorithm can be run in parallel with a Monte Carlo algorithm to obtain better results in practice. Overall, both the time and space complexity of finding a longest palindrome in a stream are essentially settled.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases streaming algorithms, space lower bounds, real-time algorithms, palindromes, Monte Carlo algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.18

1 Introduction

In the streaming model of computation, a very long input arrives sequentially in small portions and cannot be stored in full due to space limitation. While well-studied in general, this is a rather recent trend in algorithms on strings. The main goals are minimizing the space complexity, i.e., avoiding storing the already seen prefix of the string explicitly, and designing real-time algorithm, i.e., processing each symbol in worst-case constant time. However, the algorithms are usually randomized and return the correct answer with high probability. The prime example of a problem on string considered in the streaming model is pattern matching, where we want to detect an occurrence of a pattern in a given text. It is somewhat surprising that one can actually solve it using polylogarithmic space in the streaming model, as proved by Porat and Porat [13]. A simpler solution was later given by Ergün et al. [6],

27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016).
Editors: Roberto Grossi and Moshe Lewenstein; Article No. 18; pp. 18:1–18:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

while Breslauer and Galil designed a real-time algorithm [3]. Similar questions studied in such setting include multiple-pattern matching [4], approximate pattern matching [5], and parametrized pattern matching [9].

We consider computing a longest palindrome in the streaming model, where a palindrome is a fragment which reads the same in both directions. This is one of the basic questions concerning regularities in texts and it has been extensively studied in the classical non-streaming setting, see [1, 7, 11, 12] and the references therein. The notion of palindromes, but with a slightly different meaning, is very important in computational biology, where one considers strings over $\{A, T, C, G\}$ and a palindrome is a sequence equal to its reverse complement (a reverse complement reverses the sequences and interchanges A with T and C with G); see [8] and the references therein for a discussion of their algorithmic aspects. Our results generalize to biological palindromes in a straightforward manner.

We denote by $\mathsf{LPS}(S)$ the problem of finding the maximum length of a palindrome in a string S (and a starting position of a palindrome of such length in S). Solving $\mathsf{LPS}(S)$ in the streaming model was recently considered by Berenbrink et al. [2], who developed tradeoffs between the bound on the error and the space complexity for additive and multiplicative variants of the problem, that is, for approximating the length of the longest palindrome with either additive or multiplicative error. Their algorithms were Monte Carlo, i.e., returned the correct answer with high probability. They also proved that any Las Vegas algorithm achieving additive error E must necessarily use $\Omega(\frac{n}{E}\log|\Sigma|)$ bits of memory, which matches the space complexity of their solution up to a logarithmic factor in the $E \in [1, \sqrt{n}]$ range, but leaves a few questions. Firstly, does the lower bound still hold for Monte Carlo algorithms? Secondly, what is the best possible space complexity when $E \in (\sqrt{n}, n]$ in the additive variant, and what about the multiplicative version? Finally, are there real-time algorithms achieving these optimal space bounds? We answer all these questions.

Our main goal is to settle the space complexity of LPS. We start with the lower bounds in Sect. 2. First, we show that Las Vegas algorithms cannot achieve sublinear space complexity at all. Second, we prove a lower bound of $\Omega(M \log \min\{|\Sigma|, M\})$ bits of memory for Monte Carlo algorithms; here $M = n/E$ for approximating the answer with additive error E, and $M = \frac{\log n}{\log(1+\varepsilon)}$ for approximating the answer with multiplicative error $(1+\varepsilon)$. Then, in Sect. 3 we design real-time Monte Carlo algorithms matching these lower bounds up to a logarithmic factor. Our real-time Monte Carlo algorithm for LPS with additive error E uses $\mathcal{O}(n/E)$ words of space, and our real time Monte Carlo algorithm for LPS with multiplicative error $\varepsilon \le 1$ uses $\mathcal{O}\big(\frac{\log n}{\varepsilon}\big)$ words of space. Finally we present, for any m, a deterministic $\mathcal{O}(m)$-space real-time algorithm solving LPS exactly if the answer is less than m and detecting a palindrome of length $\ge m$ otherwise. The last result implies that if the input stream is fully random, then with high probability its longest palindrome can be found exactly by a real-time algorithm within logarithmic space.

Notation and Definitions. Let S denote a string of length n over an alphabet $\Sigma = \{1, \ldots, N\}$, where N is polynomial in n. We write $S[i]$ for the ith symbol of S and $S[i..j]$ for its *substring* (or *factor*) $S[i]S[i+1]\cdots S[j]$; thus, $S[1..n] = S$. A prefix (resp. suffix) of S is a substring of the form $S[1..j]$ (resp., $S[j..n]$). A string S is a *palindrome* if it equals its *reversal* $S[n]S[n-1]\cdots S[1]$. By $L(S)$ we denote the length of a longest palindrome which is a factor of S. log stands for the binary logarithm.

We consider the *streaming model* of computation: the input string $S[1..n]$ (called the *stream*) is read left to right, one symbol at a time, and cannot be stored, because the available space is sublinear in n. The space is counted as the number of $\mathcal{O}(\log n)$-bit machine words. An algorithm is *real-time* if the number of operations between two reads is bounded by a constant. An approximation algorithm for a maximization problem has *additive error* E (resp., *multiplicative error* ε) if it finds a solution with the cost at least $OPT - E$ (resp., $\frac{OPT}{1+\varepsilon}$), where OPT is the cost of optimal solution; here both E and ε can be functions of the size of the input. In the $\mathsf{LPS}(S)$ problem, $OPT = L(S)$.

A *Las Vegas algorithm* always returns a correct answer, but its memory usage on the inputs of length n is a random variable. A *Monte Carlo algorithm* gives a correct answer with high probability and has deterministic working time. Here we call "high" the probability greater than $1 - 1/n$.

2 Lower Bounds

In this section we use Yao's minimax principle [15] to prove lower bounds on the space complexity of the LPS problem in the streaming model, where the length n and the alphabet Σ of the input stream are specified. We denote this problem by $\mathsf{LPS}_\Sigma[n]$.

▶ **Theorem 1** (Yao's minimax principle for randomized algorithms)**.** *Let $\mathcal{X}$ be the set of inputs for a problem and $\mathcal{A}$ be the set of all deterministic algorithms solving it. Then, for any $x \in \mathcal{X}$ and $A \in \mathcal{A}$, the cost of running A on x is denoted by $c(a, x) \geq 0$.*

Let p be the probability distribution over $\mathcal{A}$, and let A be an algorithm chosen at random according to p. Let q be the probability distribution over $\mathcal{X}$, and let X be an input chosen at random according to q. Then $\max_{x\in\mathcal{X}} \mathbf{E}[c(A, x)] \geq \min_{a\in\mathcal{A}} \mathbf{E}[c(a, X)]$.

We use the above theorem for both Las Vegas and Monte Carlo algorithms. For Las Vegas algorithms, we consider only correct algorithms, and $c(x, a)$ is the memory usage. For Monte Carlo algorithms, we consider all algorithms (not necessarily correct) with memory usage not exceeding a certain threshold, and $c(x, a)$ is the correctness indicator function, i.e., $c(x, a) = 0$ if the algorithm is correct and $c(x, a) = 1$ otherwise.

Our proofs will be based on appropriately chosen padding. In some cases the padding requires a larger (but constant) alphabet, which can be always reduced to binary while increasing the size of the input by a constant factor. For the padding we will often use an infinite string $\nu = 0^1 1^1 0^2 1^2 0^3 1^3 \ldots$, or more precisely its prefixes of length d, denoted $\nu(d)$. Here 0 and 1 should be understood as two characters not belonging to the original alphabet. The longest palindrome in $\nu(d)$ has length $\mathcal{O}(\sqrt{d})$.

▶ **Theorem 2** (Las Vegas approximation)**.** *Let $\mathcal{A}$ be a Las Vegas streaming algorithms solving $\mathsf{LPS}_\Sigma[n]$ with additive error $E \leq 0.99n$ or multiplicative error $(1 + \varepsilon) \leq 100$ using $s(n)$ bits of memory. Then $\mathbb{E}[s(n)] = \Omega(n \log |\Sigma|)$.*

Proof. By Theorem 1, it is enough to construct a probability distribution $\mathcal{P}$ over Σ^n such that for any deterministic algorithm $\mathcal{D}$, its expected memory usage on a string chosen according to $\mathcal{P}$ is $\Omega(n \log |\Sigma|)$ in bits.

Consider solving $\mathsf{LPS}_\Sigma[n]$ with additive error E. We define $\mathcal{P}$ as the uniform distribution over $\nu(\frac{E}{2})x\$\$y\nu(\frac{E}{2})^R$, where $x, y \in \Sigma^{n'}$, $n' = \frac{n}{2} - \frac{E}{2} - 1$, and \$ is a special character not in Σ. Let us look at the memory usage of $\mathcal{D}$ after having read $\nu(\frac{E}{2})x$. We say that x is "good" when the memory usage is at most $\frac{n'}{2} \log |\Sigma|$ and "bad" otherwise. Assume that $\frac{1}{2}|\Sigma|^{n'}$ of all x's are good, then there are two strings $x \neq x'$ such that the state of $\mathcal{D}$ after having read both $\nu(\frac{E}{2})x$ and $\nu(\frac{E}{2})x'$ is exactly the same. Hence the behavior of $\mathcal{D}$ on $\nu(\frac{E}{2})x\$\$x^R\nu(\frac{E}{2})^R$ and $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ is exactly the same. The former is a palindrome of length $n = 2n' + E + 2$, so $\mathcal{D}$ must answer at least $2n' + 2$, and consequently the latter also must contain a palindrome of length at least $2n' + 2$. A palindrome inside $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ is either fully contained within $\nu(\frac{E}{2})$, x', x^R or it is a middle palindrome. But the longest palindrome inside $\nu(\frac{E}{2})$ is of length $\mathcal{O}(\sqrt{E}) < 2n'+2$ (for n large enough) and the longest palindrome inside x or x^R is of length $n' < 2n'+2$, so since we have exluced other possibilities, $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ contains a middle palindrome of length $2n'+2$. This implies that $x = x'$, which is a contradiction. Therefore, at least $\frac{1}{2}|\Sigma|^{n'}$ of all x's are bad. But then the expected memory usage of $\mathcal{D}$ is at least $\frac{n'}{4}\log|\Sigma|$, which for $E \le 0.99n$ is $\Omega(n\log|\Sigma|)$ as claimed.

Now consider solving $\mathsf{LPS}_\Sigma[n]$ with multiplicative error $(1+\varepsilon)$. An algorithm with multiplicative error $(1+\varepsilon)$ can also be considered as having additive error $E = n\cdot\frac{\varepsilon}{1+\varepsilon}$, so if the expected memory usage of such an algorithm is $o(n\log|\Sigma|)$ and $(1+\varepsilon)\le 100$ then we obtain an algorithm with additive error $E \le 0.99n$ and expected memory usage $o(n\log|\Sigma|)$, which we already know to be impossible. ◀

Now we move to Monte Carlo algorithms. We first consider exact algorithms solving $\mathsf{LPS}_\Sigma[n]$; lower bounds on approximation algorithms will be then obtained by padding the input appropriately. We introduce an auxiliary problem $\mathsf{midLPS}_\Sigma[n]$, which is to compute the length of the middle palindrome in a string of even length n over an alphabet Σ.

▶ **Lemma 3.** *There exists a constant γ such that any randomized Monte Carlo streaming algorithm $\mathcal{A}$ solving $\mathsf{midLPS}_\Sigma[n]$ or $\mathsf{LPS}_\Sigma[n]$ exactly with probability $1-\frac{1}{n}$ uses at least $\gamma\cdot n\log\min\{|\Sigma|,n\}$ bits of memory.*

Proof. First we prove that if $\mathcal{A}$ is a Monte Carlo streaming algorithm solving $\mathsf{midLPS}_\Sigma[n]$ exactly using less than $\lfloor\frac{n}{2}\log|\Sigma|\rfloor$ bits of memory, then its error probability is at least $\frac{1}{n|\Sigma|}$.

By Theorem 1, it is enough to construct probability distribution $\mathcal{P}$ over Σ^n such that for any deterministic algorithm $\mathcal{D}$ using less than $\lfloor\frac{n}{2}\log|\Sigma|\rfloor$ bits of memory, the expected probability of error on a string chosen according to $\mathcal{P}$ is at least $\frac{1}{n|\Sigma|}$.

Let $n' = \frac{n}{2}$. For any $x \in \Sigma^{n'}$, $k \in \{1,2,\ldots,n'\}$ and $c \in \Sigma$ we define

$$w(x,k,c) = x[1]x[2]x[3]\ldots x[n']x[n']x[n'-1]x[n'-2]\ldots x[k+1]cx[k-1]\ldots x[2]x[1].$$

Now $\mathcal{P}$ is the uniform distribution over all such $w(x,k,c)$.

Choose an arbitrary maximal matching of strings from $\Sigma^{n'}$ into pairs (x,x') such that $\mathcal{D}$ is in the same state after reading either x or x'. At most one string per state of $\mathcal{D}$ is left unpaired, that is at most $2^{\lfloor\frac{n}{2}\log|\Sigma|\rfloor-1}$ strings in total. Since there are $|\Sigma|^{n'} = 2^{n'\log|\Sigma|} \ge 2\cdot 2^{\lfloor\frac{n}{2}\log|\Sigma|\rfloor-1}$ possible strings of length n', at least half of the strings are paired. Let s be longest common suffix of x and x', so $x = vcs$ and $x' = v'c's$, where $c \ne c'$ are single characters. Then $\mathcal{D}$ returns the same answer on $w(x,n'-|s|,c)$ and $w(x',n'-|s|,c)$, even though the length of the middle palindrome is exactly $2|s|$ in one of them, and at least $2|s|+2$ in the other one. Therefore, $\mathcal{D}$ errs on at least one of these two inputs. Similarly, it errs on either $w(x,n'-|s|,c')$ or $w(x,n'-|s|,c')$. Thus the error probability is at least $\frac{1}{2n'|\Sigma|} = \frac{1}{n|\Sigma|}$.

Now we can prove the lemma for $\mathsf{midLPS}_\Sigma[n]$ with a standard amplification trick. Say that we have a Monte Carlo streaming algorithm, which solves $\mathsf{midLPS}_\Sigma[n]$ exactly with error probability ε using $s(n)$ bits of memory. Then we can run its k instances simultaneously and return the most frequently reported answer. The new algorithm needs $\mathcal{O}(k\cdot s(n))$ bits of memory and its error probability ε_k satisfies:

$$\varepsilon_k \le \sum_{2i<k}\binom{k}{i}(1-\varepsilon)^i\varepsilon^{k-i} \le 2^k\cdot\varepsilon^{k/2} = (4\varepsilon)^{k/2}.$$

Let us choose $\kappa = \frac{1}{6}\frac{\log(4/n)}{\log(1/(n|\Sigma|))} = \frac{1}{6}\frac{1-o(1)}{1+\log|\Sigma|/\log n} = \Theta(\frac{\log n}{\log n+\log|\Sigma|}) = \gamma\cdot\frac{1}{\log|\Sigma|}\log\min\{|\Sigma|,n\}$, for some constant γ. Now we can prove the theorem. Assume that $\mathcal{A}$ uses less than

$\kappa \cdot n \log|\Sigma| = \gamma \cdot n \log \min\{|\Sigma|, n\}$ bits of memory. Then running $\lfloor \frac{1}{2\kappa} \rfloor \geq \frac{3}{4}\frac{1}{2\kappa}$ (which holds since $\kappa < \frac{1}{6}$) instances of $\mathcal{A}$ in parallel requires less than $\lfloor \frac{n}{2} \log|\Sigma| \rfloor$ bits of memory. But then the error probability of the new algorithm is bounded from above by

$$\left(\frac{4}{n}\right)^{\frac{3}{16\kappa}} = \left(\frac{1}{n|\Sigma|}\right)^{\frac{18}{16}} \leq \frac{1}{n|\Sigma|}$$

which we have already shown to be impossible.

The lower bound for $\mathsf{midLPS}_\Sigma[n]$ can be translated into a lower bound for solving $\mathsf{LPS}_\Sigma[n]$ exactly by padding the input so that the longest palindrome is centered in the middle. Let $x = x[1]x[2]\ldots x[n]$ be the input for $\mathsf{midLPS}_\Sigma[n]$. We define

$$w(x) = x[1]x[2]x[3]\ldots x[n/2]1\underbrace{000\ldots0}_{n}1x[n/2+1]\ldots x[n].$$

Now if the length of the middle palindrome in x is k, then $w(x)$ contains a palindrome of length at least $n+k+2$. In the other direction, any palindrome inside $w(x)$ of length $\geq n$ must be centered somewhere in the middle block consisting of only zeroes and both ones are mapped to each other, so it must be the middle palindrome. Thus, the length of the longest palindrome inside $w(x)$ is exactly $n+k+2$, so we have reduced solving $\mathsf{midLPS}_\Sigma[n]$ to solving $\mathsf{LPS}_\Sigma[2n+2]$. We already know that solving $\mathsf{midLPS}_\Sigma[n]$ with probability $1-\frac{1}{n}$ requires $\gamma \cdot n \log \min\{|\Sigma|, n\}$ bits of memory, so solving $\mathsf{LPS}_\Sigma[2n+2]$ with probability $1-\frac{1}{2n+2} \geq 1-\frac{1}{n}$ requires $\gamma \cdot n \log\{|\Sigma|, n\} \geq \gamma' \cdot (2n+2) \log \min\{|\Sigma|, 2n+2\}$ bits of memory. Notice that the reduction needs $\mathcal{O}(\log n)$ additional bits of memory to count up to n, but for large n this is much smaller than the lower bound if we choose $\gamma' < \frac{\gamma}{4}$. ◀

To obtain a lower bound for Monte Carlo additive approximation, we observe that any algorithm solving $\mathsf{LPS}_\Sigma[n]$ with additive error E can be used to solve $\mathsf{LPS}_\Sigma[\frac{n-E}{E+1}]$ exactly by inserting $\frac{E}{2}$ zeroes between every two characters, in the very beginning, and in the very end. However, this reduction requires $\log(\frac{E}{2}) \leq \log n$ additional bits of memory for counting up to $\frac{E}{2}$ and cannot be used when the desired lower bound on the required number of bits $\Omega(\frac{n}{E} \log \min(|\Sigma|, \frac{n}{E})$ is significantly smaller than $\log n$. Therefore, we need a separate technical lemma which implies that both additive and multiplicative approximation with error probability $\frac{1}{n}$ require $\Omega(\log n)$ bits of space.

▶ **Lemma 4.** *Let $\mathcal{A}$ be any randomized Monte Carlo streaming algorithm solving $\mathsf{LPS}_\Sigma[n]$ with additive error at most $0.99n$ or multiplicative error at most $n^{0.49}$ and error probability $\frac{1}{n}$. Then $\mathcal{A}$ uses $\Omega(\log n)$ bits of memory.*

Combining the reduction with the technical lemma and taking into account that we are reducing to a problem with string length of $\Theta(\frac{n}{E})$, we obtain the following.

▶ **Theorem 5** (Monte Carlo additive approximation)**.** *Let $\mathcal{A}$ be any randomized Monte Carlo streaming algorithm solving $\mathsf{LPS}_\Sigma[n]$ with additive error E with probability $1-\frac{1}{n}$. If $E \leq 0.99n$ then $\mathcal{A}$ uses $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ bits of memory.*

Proof. Define $\sigma = \min\{|\Sigma|, \frac{n}{E}\}$.

Because of Lemma 4 it is enough to prove that $\Omega(\frac{n}{E} \log \sigma)$ is a lower bound when

$$E \leq \frac{\gamma}{2} \cdot \frac{n}{\log n} \log \sigma. \tag{1}$$

Assume that there is a Monte Carlo streaming algorithm $\mathcal{A}$ solving $\mathsf{LPS}_\Sigma[n]$ with additive error E using $o(\frac{n}{E} \log \sigma)$ bits of memory and probability $1-\frac{1}{n}$. Let $n' = \frac{n-E/2}{E/2+1} \geq \frac{n}{E}$ (the last inequality, equivalent to $n \geq E \cdot \frac{E}{E-2}$ holds because $E \leq 0.99n$ and because we can assume that $E \geq 200$). Given a string $x[1]x[2]\ldots x[n']$, we can simulate running $\mathcal{A}$ on $0^E x[1]0^{E/2}x[2]0^{E/2}x[3]\ldots 0^{E/2}x[n']0^{E/2}$ to calculate R (using $\log(E/2) \leq \log n$ additional bits of memory), and then return $\left\lfloor \frac{R}{E/2+1} \right\rfloor$. We call this new Monte Carlo streaming algorithm $\mathcal{A}'$. Recall that $\mathcal{A}$ reports the length of the longest palindrome with additive error E. Therefore, if the original string contains a palindrome of length r, the new string contains a palindrome of length $\frac{E}{2} \cdot (r+1) + r$, so $R \geq r(E/2+1)$ and $\mathcal{A}'$ will return at least r. In the other direction, if $\mathcal{A}'$ returns r, then the new string contains a palindrome of length $r(E/2+1)$. If such palindrome is centered so that $x[i]$ is matched with $x[i+1]$ for some i, then it clearly corresponds to a palindrome of length r in the original string. But otherwise every $x[i]$ within the palindrome is matched with 0, so in fact the whole palindrome corresponds to a streak of consecutive zeroes in the new string and can be extended to the left and to the right to start and end with 0^E, so again it corresponds to a palindrome of length r in the original string. Therefore, $\mathcal{A}'$ solves $\mathsf{LPS}_\Sigma[n']$ exactly with probability $1 - \frac{1}{(n'(E/2+1)+E/2)} \geq 1 - \frac{1}{n'}$ and uses $o(\frac{n'(E/2+1)+E/2}{E/2}\log\sigma) + \log n = o(n'\log\sigma) + \log n$ bits of memory. Observe that by Lemma 3 we get a lower bound

$$\gamma \cdot n' \log\min\{|\Sigma|, n'\} \geq \frac{\gamma}{2} \cdot n' \log\sigma + \frac{\gamma}{2} \cdot \frac{n}{E} \log\sigma \geq \frac{\gamma}{2} \cdot n' \log\sigma + \log n$$

(where the last inequality holds because of Eq.(1)). Then, for large n we obtain contradiction as follows

$$o(n'\log\sigma) + \log n < \frac{\gamma}{2} \cdot n' \log\sigma + \log n.$$ ◀

Finally, we consider multiplicative approximation. The proof follows the same basic idea as of Theorem 5, however is more technically involved. The main difference is that due to uneven padding, we are reducing to $\mathsf{midLPS}_\Sigma[n']$ instead of $\mathsf{LPS}_\Sigma[n']$.

▶ **Theorem 6** (Monte Carlo multiplicative approximation). *Let $\mathcal{A}$ be any randomized Monte Carlo streaming algorithm solving $\mathsf{LPS}_\Sigma[n]$ with multiplicative error $(1+\varepsilon)$ with probability $1-\frac{1}{n}$. If $n^{-0.98} \leq \varepsilon \leq n^{0.49}$ then $\mathcal{A}$ uses $\Omega(\frac{\log n}{\log(1+\varepsilon)} \log\min\{|\Sigma|, \frac{\log n}{\log(1+\varepsilon)}\})$ bits of memory.*

3 Real-Time Algorithms

In this section we design real-time Monte Carlo algorithms within the space bounds matching the lower bounds from Sect. 2 up to a factor bounded by $\log n$. The algorithms make use of the hash function known as the *Karp-Rabin fingerprint* [10]. Let p be a fixed prime from the range $[n^{3+\alpha}, n^{4+\alpha}]$ for some $\alpha > 0$, and r be a fixed integer randomly chosen from $\{1, \ldots, p-1\}$. For a string S, its forward hash and reversed hash are defined, respectively, as

$$\phi^F(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^i\right) \bmod p \quad \text{and} \quad \phi^R(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^{n-i+1}\right) \bmod p\,.$$

Clearly, the forward hash of a string coincides with the reversed hash of its reversal. Thus, if u is a palindrome, then $\phi^F(u) = \phi^R(u)$. The converse is also true modulo the (improbable) collisions of hashes, because for two strings $u \neq v$ of length m, the probability that $\phi^F(u) = \phi^F(v)$ is at most m/p. This property allows one to detect palindromes with high probability by comparing hashes. (This approach is somewhat simpler than the one of [2]; in particular, we don't need "fingerprint pairs" used there.) In particular, a real-time algorithm makes

Algorithm 1 Algorithm ABasic, ith iteration.

if $i \bmod t_E = 0$ **then**
  add I to the beginning of SP
read $S[i]$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
for all elements v of SP **do**
  if $S[v.i..i]$ is a palindrome and $answer.len < i-v.i+1$ **then**
    $answer \leftarrow (v.i, i-v.i+1)$

$\mathcal{O}(n)$ comparisons and thus faces a collision with probability $\mathcal{O}(n^{-1-\alpha})$ by the choice of p. All further considerations assume that no collisions happen. For an input stream S, we denote $F^F(i,j) = \phi^F(S[i..j])$ and $F^R(i,j) = \phi^R(S[i..j])$. Let $I(i)$ denote the tuple $(i, F^F(1,i-1), F^R(1,i-1), r^{-(i-1)} \bmod p, r^i \bmod p)$. The proposition below is immediate from definitions and simple arithmetical manipulations.

▶ **Proposition 7.** ***1.*** *Given $I(i)$ and $S[i]$, the tuple $I(i+1)$ can be computed in $\mathcal{O}(1)$ time.*
2. *Given $I(i)$ and $I(j+1)$, the string $S[i..j]$ can be checked for palindromicity in $\mathcal{O}(1)$ time.*

3.1 Additive Error

▶ **Theorem 8.** *There is a real-time Monte Carlo algorithm solving the problem* LPS(S) *with the additive error $E = E(n)$ using $\mathcal{O}(n/E)$ space, where $n = |S|$.*

First we present a simple (and slow) algorithm which solves the posed problem, i.e., finds in S a palindrome of length $\ell(S) \geq L(S) - E$, where $L(S)$ is the length of the longest palindrome in S. Later this algorithm will be converted into a real-time one. We store the sets $I(j)$ for some values of j in a doubly-linked list SP in the decreasing order of j's. The longest palindrome currently found is stored as a pair $answer = (pos, len)$, where pos is its initial position and len is its length. Let $t_E = \lfloor \frac{E}{2} \rfloor$.

In Algorithm ABasic we add $I(j)$ to the list SP for each j divisible by t_E. This allows us to check for palindromicity, at ith iteration, all factors of the form $S[kt_E..i]$. We assume throughout the section that at the beginning of ith iteration the value $I(i)$ is stored in a variable I.

▶ **Proposition 9.** *Algorithm ABasic finds in S a palindrome of length $\ell(S) \geq L(S) - E$ using $\mathcal{O}(n/E)$ time per iteration and $\mathcal{O}(n/E)$ space.*

Proof. Both the time and space bounds arise from the size of the list SP, which is bounded by $n/t_E = \mathcal{O}(n/E)$; the number of operations per iteration is proportional to this size due to Proposition 7. Now let $S[i..j]$ be a longest palindrome in S. Let $k = \lceil \frac{i}{t_E} \rceil t_E$. Then $i \leq k < i + t_E$. At the kth iteration, $I(k)$ was added to SP; then the palindrome $S[k..j-(k-i)]$ was found at the iteration $j-(k-i)$. Its length is

$$j-(k-i)-k+1 = j-i-2(k-i)+1 > (j-i+1) - 2t_E = L(S) - 2\left\lfloor \frac{E}{2} \right\rfloor \geq L(S) - E,$$

as required. ◀

The resource to speed up Algorithm ABasic stems from the following

▶ **Lemma 10.** *During one iteration, the length $answer.len$ increases by at most $2 \cdot t_E$.*

Algorithm 2 Algorithm A, ith iteration.

1: **if** $i \bmod t_E = 0$ **then**
2:     add I to the beginning of SP
3:     **if** $i = t_E$ **then**
4:         $sp \leftarrow first(SP)$
5: read $S[i]$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
6: $sp \leftarrow previous(sp)$ ▷ if exists
7: **while** $i - sp.i + 1 \le answer.len$ and $(sp \ne last(SP))$ **do**
8:     $sp \leftarrow next(sp)$
9: **for** all existing v in $\{sp, next(sp)\}$ **do**
10:     **if** $S[v.i..i]$ is a palindrome and $answer.len < i{-}v.i{+}1$ **then**
11:         $answer \leftarrow (v.i, i{-}v.i{+}1)$

Proof. Let $S[j..i]$ be the longest palindrome found at the ith iteration. If $i - j + 1 \le 2t_E$ then the statement is obviously true. Otherwise the palindrome $S[j{+}t_E..i{-}t_E]$ of length $i{-}j{+}1{-}2t_E$ was found before (at the $(i{-}t_E)$th iteration), and the statement holds again. ◀

Lemma 10 implies that at each iteration SP contains only two elements that can increase *answer.len*. Hence we get the following Algorithm A.

Due to Lemma 10, the cycle at lines 9–11 of Algorithm A computes the same sequence of values of *answer* as the cycle at lines 4–6 of Algorithm ABasic. Hence it finds a palindrome of required length by Proposition 9. Clearly, the space used by the two algorithms differs by a constant. To prove that an iteration of Algorithm A takes $\mathcal{O}(1)$ time, it suffices to note that the cycle in lines 7–8 performs at most two iterations. Theorem 8 is proved.

3.2 Multiplicative Error

▶ **Theorem 11.** *There is a real-time Monte Carlo algorithm solving the problem* LPS(S) *with multiplicative error* $\varepsilon = \varepsilon(n) \in (0,1]$ *using* $\mathcal{O}\big(\frac{\log n}{\varepsilon}\big)$ *space, where* $n = |S|$.

As in the previous section, we first present a simpler algorithm MBasic with non-linear working time and then upgrade it to a real-time algorithm. The algorithm must find a palindrome of length $\ell(S) \ge \frac{L(S)}{1+\varepsilon}$. The next lemma is straightforward.

▶ **Lemma 12.** *If* $\varepsilon \in (0,1]$, *the condition* $\ell(S) \ge L(S)(1-\varepsilon/2)$ *implies* $\ell(S) \ge \frac{L(S)}{1+\varepsilon}$.

We set $q_\varepsilon = \lceil \log \frac{2}{\varepsilon} \rceil$. The main difference in the construction of algorithms with the multiplicative and additive error is that here all sets $I(i)$ are added to the list SP, but then, after a certain number of steps, are deleted from it. The number of iterations the set $I(i)$ is stored in SP is determined by the time-to-live function $ttl(i)$ defined below. This function is responsible for both the correctness of the algorithm and the space bound.

Let $\beta(i)$ be the position of the rightmost 1 in the binary representation of i (the position 0 corresponds to the least significant bit). We define

$$ttl(i) = 2^{q_\varepsilon + 2 + \beta(i)}. \tag{2}$$

The definition is illustrated by Fig. 1. Now we state a few properties of the list SP.

▶ **Lemma 13.** *For any integers* $a \ge 1$ *and* $b \ge 0$, *there exists a unique integer* $j \in [a, a + 2^b)$ *such that* $ttl(j) \ge 2^{q_\varepsilon + 2 + b}$.

Algorithm 3 Algorithm MBasic, ith iteration.

add I to the beginning of SP
for all v in SP **do**
    if $v.i + ttl(v.i) = i$ **then**
        delete v from SP
read $S[i]$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
for all v in SP **do**
    if $S[v.i..i]$ is a palindrome and $answer.len < i{-}v.i{+}1$ **then**
        $answer \leftarrow (v.i, i{-}v.i{+}1)$

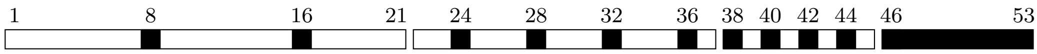

Figure 1 The state of the list SP after the iteration $i = 53$ ($q_\varepsilon = 1$ is assumed). Black squares indicate the numbers j for which $I(j)$ is currently stored. For example, (2) implies $ttl(28) = 2^{1+2+2} = 32$, so $I(28)$ will stay in SP until the iteration $28 + 32 = 60$.

Proof. By (2), $ttl(j) \geq 2^{q_\varepsilon+2+b}$ if and only if $\beta(j) \geq b$, i.e., j is divisible by 2^b by the definition of β. Among any 2^b consecutive integers, exactly one has this property. ◀

Figure 1 shows the partition of the range $(0, i]$ into intervals having lengths that are powers of 2 (except for the leftmost interval). In general, this partition consists of the following intervals, right to left:

$$(i-2^{q_\varepsilon+2}, i], (i-2^{q_\varepsilon+3}, i-2^{q_\varepsilon+2}], \ldots, (i-2^k, i-2^{k-1}], (0, i-2^k], \text{ where } k = \lceil \log n \rceil - 1. \quad (3)$$

Lemma 13 and (2) imply the following lemma on the distribution of the elements of SP.

▶ **Lemma 14.** *After each iteration, the first interval (resp., the last interval; each of the remaining intervals) in* (3) *contains* $2^{q_\varepsilon+2}$ *(resp., at most* $2^{q_\varepsilon+1}$*; exactly* $2^{q_\varepsilon+1}$*) elements of the list* SP*.*

The number of the intervals in (3) is $\mathcal{O}(\log n)$, so from Lemma 14 and the definition of q_ε

▶ **Lemma 15.** *After each iteration, the size of the list* SP *is* $\mathcal{O}(\frac{\log n}{\varepsilon})$*.*

▶ **Proposition 16.** *Algorithm MBasic finds a palindrome of length* $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$ *using* $\mathcal{O}(\frac{\log n}{\varepsilon})$ *time per iteration and* $\mathcal{O}(\frac{\log n}{\varepsilon})$ *space.*

Proof. Both the time per iteration and the space are dominated by the size of the list SP. Hence the required complexity bounds follow from Lemma 15. For the proof of correctness, let $S[i..j]$ be a palindrome of length $L(S)$. Further, let $d = \lfloor \log L(S) \rfloor$.

If $d < q_\varepsilon + 2$, the palindrome $S[i..j]$ will be found exactly, because $I(i)$ is in SP at the jth iteration:

$$i + ttl(i) \geq i + 2^{q_\varepsilon+2} \geq i + 2^{d+1} > i + L(S) > j\,.$$

Otherwise, by Lemma 13 there exists a unique $k \in [i, i + 2^{d-q_\varepsilon-1})$ such that $ttl(k) \geq 2^{d+1}$. Hence at the iteration $j - (k - i)$ the palindrome $S[i{+}(k{-}i)..j{-}(k{-}i)]$ will be found, because $I(k)$ is in SP at this iteration:

$$k + ttl(k) \geq i + ttl(k) \geq i + 2^{d+1} > j \geq j - (k - i)\,.$$

The length of this palindrome satisfies the requirement of the proposition:

$$j-(k-i)-(i+(k-i))+1 = L(S)-2(k-i) \geq L(S)-2^{d-q_\varepsilon} \geq L(S)-\frac{L(S)}{2^{q_\varepsilon}} \geq L(S)\big(1-\frac{\varepsilon}{2}\big)\,.$$

The reference to Lemma 12 finishes the proof. ◀

Now we speed up Algorithm MBasic. It has two slow parts: deletions from the list SP and checks for palindromes. By Lemmas 17, 18, $\mathcal{O}(1)$ checks is enough at each iteration.

▶ **Lemma 17.** *Suppose that at some iteration the list SP contains consecutive elements $I(d), I(c), I(b), I(a)$. Then $b-a \leq d-b$.*

Proof. Let j be the number of the considered iteration. Note that $a < b < c < d$. Consider the interval in (3) containing a. If $a \in (j-2^{q_\varepsilon+2}, j]$, then $b-a=1$ and $d-b=2$, so the required inequality holds. Otherwise, let $a \in (j-2^{q_\varepsilon+2+x}, j-2^{q_\varepsilon+2+x-1}]$. Then by (2) $\beta(a) \geq x$; moreover, any $I(k)$ such that $a < k \leq j$ and $\beta(k) \geq x$ is in SP. Hence, $b-a \leq 2^x$. By Lemma 14 each interval, except for the leftmost one, contains at least $2^{q_\varepsilon+1} \geq 4$ elements. Thus each of the numbers b, c, d belongs either to the same interval as a or to the previous interval $(j-2^{q_\varepsilon+2+x-1}, j-2^{q_\varepsilon+2+x-2}]$. Again by (2) we have $\beta(b), \beta(c), \beta(d) \geq x-1$. So $c{-}b, d{-}c \geq 2^{x-1}$, whence the result. ◀

We call an element v of SP *valuable at ith iteration* if $i-v.i+1 > answer.len$ and $S[v.i..i]$ can be a palindrome. (That is, Algorithm MBasic does not store enough information to predict that the condition in its line 7 is false for v.)

▶ **Lemma 18.** *At each iteration, SP contains at most three valuable elements. Moreover, if $I(d'), I(d)$ are stored in consecutive elements of SP and $i-d' < answer.len \leq i-d$, where i is the number of the current iteration, then the valuable elements are consecutive in SP, starting with the one containing $I(d)$.*

Proof. Let d be as in the condition of the lemma and v be the element containing $I(d)$. If v is followed in SP by at most two elements, we are done. If it is not the case, let the three next elements be v_1, v_2, v_3, containing $I(c), I(b), I(a)$ respectively. If $S[v_3.i..i] = S[a..i]$ is a palindrome then $S[a{+}(b{-}a)..i{-}(b{-}a)]$ is also a palindrome. At the iteration $i{-}(b{-}a)$ the set $I(b)$ was in SP, so this palindrome was found. Hence, at the ith iteration the value $answer.len$ is at least the length of this palindrome, which is $i-a+1-2(b-a)$. By Lemma 17, $b-a \leq d-b$, implying $answer.len \geq i-a+1-(b-a)-(d-b) = i-d+1$. This inequality contradicts the definition of d; hence, $S[a..i]$ is not a palindrome. By the same argument, the elements following v_3 in SP do not produce palindromes as well. Thus, only the elements v, v_1, v_2 are valuable. ◀

Now we turn to deletions. The function $ttl(x)$ has the following nice property.

▶ **Lemma 19.** *The function $x \to x + ttl(x)$ is injective.*

Lemma 19 implies that at most one element is deleted from SP at each iteration. To perform this deletion in $\mathcal{O}(1)$ time, we need an additional data structure. By $BS(x)$ we denote a linked list of maximal segments of 1's in the binary representation of x. For example, the binary representation of $x = 12345$ and $BS(x)$ are as follows:

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	1	0	0	1

$$BS(12345) = \{[0,0],[3,5],[10,10],[12,13]\}$$

Clearly, $BS(x)$ uses $\mathcal{O}(\log x)$ space. The following lemma is easy.

Algorithm 4 Algorithm M, ith iteration.

1: add I to the beginning of SP
2: **if** $i = 1$ **then**
3: $\quad sp \leftarrow first(SP)$
4: compute $BS[i]$ from BS; $BS \leftarrow BS[i]$; compute $\beta(i)$ from BS
5: **if** $QU(\beta(i))$ is not empty **then**
6: $\quad v \leftarrow$ element of SP pointed by $first(QU(\beta(i)))$
7: $\quad$ **if** $v = sp$ **then**
8: $\quad\quad sp \leftarrow next(sp)$
9: $\quad$ delete v; delete $first(QU(\beta(i)))$
10: add pointer to $first(SP)$ to $QU(\beta(i))$
11: read $S[i]$; compute $I(i+1)$ from I; $I \leftarrow I(i+1)$
12: $sp \leftarrow previous(sp)$ $\triangleright$ if exists
13: **while** $i - sp.i + 1 \leq answer.len$ and $sp \neq last(SP)$ **do**
14: $\quad sp \leftarrow next(sp)$
15: **for** all existing v in $\{sp, next(sp), next(next(sp))\}$ **do**
16: $\quad$ **if** $S[v.i..i]$ is a palindrome and $answer.len < i - v.i + 1$ **then**
17: $\quad\quad answer \leftarrow (v.i, i - v.i + 1)$

▶ **Lemma 20.** *Both $\beta(x)$ and $BS(x+1)$ can be obtained from $BS(x)$ in $\mathcal{O}(1)$ time.*

Thus, if we support one list BS which is equal to $BS(i)$ at the end of the ith iteration, we have $\beta(i)$. If $I(a)$ should be deleted from SP at this iteration, then $\beta(a) = \beta(i)$ (see Lemma 19). The following lemma is trivial.

▶ **Lemma 21.** *If $a < b$ and $ttl(a) = ttl(b)$, then $I(a)$ is deleted from SP before $I(b)$.*

By Lemma 21, the information about the positions with the same ttl (in other words, with the same β) are added to and deleted from SP in the same order. Hence it is possible to keep a queue $QU(x)$ of the pointers to all elements of SP corresponding to the positions j with $\beta(j) = x$. These queues constitute the last ingredient of our real-time Algorithm M.

Proof of Theorem 11. After every iteration, Algorithm M has the same list SP (see Fig. 1) as Algorithm MBasic, because these algorithms add and delete the same elements. Due to Lemma 18, Algorithm M returns the same answer as Algorithm MBasic. Hence by Proposition 16 Algorithm M finds a palindrome of required length. Further, Algorithm M supports the list BS of size $\mathcal{O}(\log n)$ and the array QU containing $\mathcal{O}(\log n)$ queues of total size equal to the size of SP. Hence, it uses $\mathcal{O}(\frac{\log n}{\varepsilon})$ space in total by Lemma 15. The cycle in lines 13–14 performs at most three iterations. Indeed, let z be the value of sp after the previous iteration. Then this cycle starts with $sp = previous(z)$ (or with $sp = z$ if z is the first element of SP) and ends with $sp = next(next(z))$ at the latest. By Lemma 20, both $BS(i)$ and $\beta(i)$ can be computed in $\mathcal{O}(1)$ time. Therefore, each iteration takes $\mathcal{O}(1)$ time. ◀

▶ Remark. Since for $\varepsilon \leq 1$ the classes $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ and $\mathcal{O}\big(\frac{\log n}{\varepsilon}\big)$ coincide, Algorithm M uses space within a $\log n$ factor from the lower bound of Theorem 6. Further, let $\varepsilon = \varepsilon(n)$ be a growing function. Algorithm M can be transformed, with some additional technicalities, into a real-time algorithm which solves $\mathsf{LPS}(S)$ with the multiplicative error ε using $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ space. The basic idea of transformation is to replace all binary representations with those in base proportional to $1+\varepsilon$, and thus shrink the size of the lists SP and BS.

3.3 The Case of Short Palindromes

A typical string contains only short palindromes, as Lemma 22 below shows (for more on palindromes in random strings, see [14]). Knowing this, it is quite useful to have a deterministic real-time algorithm which finds a longest palindrome exactly if it is "short", otherwise reporting that it is "long". This idea is formalized in Theorem 23. Its proof is based on a modification of the Manacher's algorithm with a sliding window and lazy computation.

▶ **Lemma 22.** *If an input stream $S \in \Sigma^*$ is picked up uniformly at random among all strings of length n, where $n \geq |\Sigma|$, then for any positive constant c the probability that S contains a palindrome of length greater than $\frac{2(c+1)\log n}{\log |\Sigma|}$ is $\mathcal{O}(n^{-c})$.*

▶ **Theorem 23.** *Let m be a positive integer. There exists a deterministic real-time algorithm working in $\mathcal{O}(m)$ space, which solves $\mathsf{LPS}(S)$ exactly if $L(S) < m$, and otherwise finds a palindrome of length m or $m+1$ as an approximate solution to $\mathsf{LPS}(S)$.*

▶ Remark. Lemma 22 and Theorem 23 show a "practical" way to solve LPS. For example, one can run Algorithm M and Algorithm E, both in $\mathcal{O}(\log n)$ space, in parallel. Then either Algorithm E will give an exact answer (which happens with high probability if the input stream is a "typical" string) or both algorithms will produce approximations: one of fixed length and one with an approximation guarantee (modulo the hash collision).

References

1. A. Apostolico, D. Breslauer, and Z. Galil. Parallel detection of all palindromes in a string. *Theoret. Comput. Sci.*, 141:163–173, 1995.
2. P. Berenbrink, F. Ergün, F. Mallmann-Trenn, and E. Sadeqi Azer. Palindrome recognition in the streaming model. In *STACS 2014*, volume 25 of *LIPIcs*, pages 149–161, 2014.
3. D. Breslauer and Z. Galil. Real-time streaming string-matching. In *CPM 2011*, volume 6661 of *LNCS*, pages 162–172. Springer, 2011.
4. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *ESA 2015*, volume 9294 of *LNCS*, pages 361–372. Springer, 2015.
5. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. The k-mismatch problem revisited. In *SODA 2016*, pages 2039–2052. SIAM, 2016.
6. Funda Ergün, Hossein Jowhari, and Mert Saglam. Periodicity in streams. In *RANDOM 2010*, volume 6302 of *LNCS*, pages 545–559. Springer, 2010.
7. Z. Galil and J. Seiferas. A linear-time on-line recognition algorithm for "Palstar". *J. ACM*, 25:102–111, 1978.
8. Paweł Gawrychowski, Florin Manea, and Dirk Nowotka. Testing Generalised Freeness of Words. In *STACS 2014*, LIPIcs, pages 337–349, 2014.
9. Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *STACS 2015*, LIPIcs, pages 400–411, 2013.
10. R. Karp and M. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.
11. D. E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.
12. G. Manacher. A new linear-time on-line algorithm finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975.
13. B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In *FOCS 2009*, pages 315–323. IEEE Computer Society, 2009.

14 M. Rubinchik and A. M. Shur. The number of distinct subpalindromes in random words. arXiv:1505.08043 [math.CO], 2015.

15 Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *FOCS 1977*, pages 222–227. IEEE Computer Society, 1977.