Bit-parallel algorithms for computing all the runs in a string

Kazunori Hirashima[1], Hideo Bannai[1], Wataru Matsubara[2], Akira Ishino[2,3], and Ayumi Shinohara[2]

[1] Department of Informatics, Kyushu University,
744 Motooka, Nishiku, Fukuoka 819-0395 Japan.
{kazunori.hirashima,bannai}@inf.kyushu-u.ac.jp
[2] Graduate School of Information Science, Tohoku University,
Aramaki aza Aoba 6-6-05, Aoba-ku, Sendai 980-8579, Japan
{matsubara@shino., ishino@, ayumi@}ecei.tohoku.ac.jp
[3] Presently at Google Japan Inc.

Abstract. We present three bit-parallel algorithms for computing all the runs in a string. The algorithms are very efficient especially for computing all runs of short binary strings, allowing us to run the algorithm for all binary strings of length up to 47 in a few days, using a PC with the help of GPGPU. We also present some related statistics found from the results.

1 Introduction

Repetitions in strings is an important element in the analysis and processing of strings. It was shown in [8] that when considering *maximal repetitions*, or *runs*, the maximum number of runs $\rho(n)$ in a string of length n is $O(n)$. The result leads to a linear time algorithm for computing all the runs in a string. Although no bounds for the constant factor was given, it was conjectured that $\rho(n) < n$.

Recently, there has been steady progress towards proving this conjecture [12,13,3]. The currently known best upper bound[1] is $\rho(n) \leq 1.029n$, obtained by calculations based on the proof technique of [3]. On the other hand, it was shown in [6] that the value $\alpha = \lim_{n\to\infty} \rho(n)/n$ exists, but is never reached. A lower bound on α was first presented in [5], where it was shown that $\alpha \geq \frac{3}{1+\sqrt{5}} \approx 0.927$. Although it was conjectured that this bound is optimal [4], a counter example was shown in [10], giving a new lower bound of 0.944565. The currently known best lower bound is $(11z^2+7z-6)/(11z^2+8z-6) \approx 0.94457571235$, where z is the real root of $z^3 = z+1$. This bound was conjectured for a new series of words in [9], and proved independently for a different series of words in [14]. Whether or not the original conjecture $\rho(n) < n$ of [8] holds, or more importantly, the exact constant $\lim_{n\to\infty} \rho(n)/n$ is still not known. On a related note, the average number of runs in a word of a given length has been completely characterized in [11].

In order to better understand the combinatorial properties of runs in strings, an exhaustive calculation of runs in short strings could be very useful. In previous work [7], the maximum number of runs function was shown for binary strings of length up to 31, together with an example of a string which achieves the maximum number of runs for each length. In this paper, we present several algorithms for computing all the runs in short binary strings using *bit-parallel* techniques.

[1] Presented on a website http://www.csd.uwo.ca/faculty/ilie/runs.html

Kazunori Hirashima, Hideo Bannai, Wataru Matsubara, Akira Ishino, Ayumi Shinohara : Bit-parallel algorithms for computing all the runs in a string, pp. 203–213.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2 © Czech Technical University in Prague, Czech Republic

In [2], a fast suffix array based algorithm for calculating all the runs in a string was presented. However, as the algorithm relies on a Lempel-Ziv factorization, our algorithm is much simpler and efficient for binary strings whose length fits in a computer word. The simple algorithm also allows us to implement a very efficient massively parallelized version using General Purpose Graphics Processor Unit Programming (GPGPU). We successfully compute the maximal number of runs, as well as several other related statistics, for binary strings of length up to 47.

2 Preliminaries

Let Σ be a finite set of symbols, called an *alphabet*. Strings x, y and z are said to be a *prefix*, *substring*, and *suffix* of the string $w = xyz$, respectively. The length of a string w is denoted by $|w|$. The i-th symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. A string w has period p if $w[i] = w[i + p]$ for $1 \leq i \leq |w| - p$.

A string u is a *run* of period p if it has period $p \leq |u|/2$. A substring $u = w[i : j]$ of w is a *run in w* if it is a run of some period p and neither $w[i-1 : j]$ nor $w[i : j+1]$ is a run of period p, that means the run is maximal. We denote the run $u = w[i : j]$ in w by the pair $\langle i, j\rangle$ of its begin position i and end position j.

For example, the string aabaabaaaacaacac contains the following 7 runs: $\langle 1, 2\rangle = \mathtt{a}^2$, $\langle 4, 5\rangle = \mathtt{a}^2$, $\langle 7, 10\rangle = \mathtt{a}^4$, $\langle 12, 13\rangle = \mathtt{a}^2$, $\langle 13, 16\rangle = (\mathtt{ac})^2$, $\langle 1, 8\rangle = (\mathtt{aab})^{\frac{8}{3}}$, and $\langle 9, 15\rangle = (\mathtt{aac})^{\frac{7}{3}}$. A run in w is called a *prefix run* if it is also a prefix of w. Among the above 7 runs, the prefix runs are $\langle 1, 2\rangle$ and $\langle 1, 8\rangle$.

3 Algorithms

From the definition of run in a string, we have only to consider the periods of length at most $|w|/2$ in order to count all runs in w. In the next subsections, we introduce 3 bit-parallel algorithms for counting all runs in w.

We will use the bitwise operations AND, OR, NOT, XOR, SHIFT _RIGHT, and SHIFT_LEFT, denoted by &, |, ∼, ^, >>, and <<, respectively, as in the C language.

3.1 Counting prefix runs

Let us begin by counting all prefix runs in a given string. Table 1 shows the continuations of each period in the string w = aabaabaaaacaacac, which has two prefix runs $\langle 1, 2\rangle$ and $\langle 1, 8\rangle$. In the table, the value at row p and column j is 1 if and only if p is a period of prefix $w[1 : j]$. The cell (p, j) is shadowed if $2p < j$, and is said to be in the *active area*. In each row, the first (leftmost) position where the period discontinued is emphasized by displaying **0** in bold face. If its position (p, j) is in the active area, it implies that the prefix $u = w[1 : j - 1]$ becomes a *run* of period p since $p \leq |u|/2$. Moreover, if any period continues to the end (rightmost), it means that the whole string w itself is a (prefix) run. In the example, $\langle 1, 16\rangle$ is not a prefix run since no period continues to the end.

We will efficiently compute the table by representing a column as a bit vector named *alive*, and keep tracking it by clever bit operations in the spirit of *bit parallelism* [1,16]. We first represent the occurrences of each character in the string w as

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
period	a	a	b	a	a	b	a	a	a	a	c	a	a	c	a	c
1	1	1	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0	0
4	1	1	1	1	1	**0**	0	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	**0**	0	0	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0	0
7	1	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0
8	1	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0

Table 1. Continuations of each period in the string `aabaabaaaacaacac`.

```
w[1] = a
                1 1 1 1 1 1 1 1   alive
w[2] = a      & 1 1 1 1 1 1 1 1   bitmask_2
                1 1 1 1 1 1 1 1   alive
w[3] = b      & 1 1 1 1 1 1 0 0   bitmask_3
                1 1 1 1 1 1 0 0   alive
w[4] = a      & 1 1 1 1 1 1 1 0   bitmask_4
                1 1 1 1 1 1 0 0   alive
w[5] = a      & 1 1 1 1 1 1 0 1   bitmask_5
                1 1 1 1 1 1 0 0   alive
w[6] = b      & 1 1 1 0 0 1 0 0   bitmask_6
                1 1 1 0 0 1 0 0   alive
   ⋮                  ⋮              ⋮
```

Occurrences of each character in w.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
occ	a	a	b	a	a	b	a	a	a	a	c	a	a	c	a	c
occ[a]	1	1	0	1	1	0	1	1	1	1	0	1	1	0	1	0
occ[b]	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
occ[c]	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1

Figure 1. Bit operations to compute the period continuations, for the string w = `aabaabaaaacaacac` (left). The representation here is rotated 90° compared with Table 1 to show the bit representation horizontally. The active area is shadowed in the same way. The italic *1*'s in the prefix part of the bitmasks are always 1 regardless of the string w. Remark that the essential part of the bitmask can be directly derived from the occurrence table (right) of each character in w. For example, in $bitmask_5$, the essential part 1101 equals to $occ[\texttt{a}][1\!:\!4]$ since $w[5] = \texttt{a}$, and 00100 in $bitmask_6$ is equal to $occ[\texttt{b}][1\!:\!5]$ since $w[6] = \texttt{b}$.

a bit vector $occ[]$, as shown in Fig. 1 (right), where $occ[c][i] = 1$ if $w[i] = c$, and 0 otherwise, for any $c \in \Sigma$. The bit vector will be used to generate bitmasks to compute the next *alive* by a logical AND operation demonstrated in Fig. 1 (left). The desired $bitmask_i$ for $i \geq 2$ is obtained by $occ[c][1 : i-1]$ where $c = w[i]$, and filling sufficient numbers of preceding *1*'s. The initial value of *alive* is $\sim 0 = 11\ldots1$. When the length of w is at most twice the word size of the computer, each bit vectors *alive* and *bitmask* fit in a single register, and can be processed very efficiently. Whenever the value of *alive* becomes 0, (in the current example, at $w[11]$) we can immediately quit the computation since no bit in *alive* can turn from 0 into 1 by AND operations.

We now turn our attention to count other runs that are not a prefix of the given string w. In principle, we would use the above procedure at each starting positions for $2 \leq i \leq |w|-1$. However, a little care must be taken to avoid duplicated counting. Let us consider the string $v = w[3\!:\!16] =$ `baabaaaacaacac` which is a suffix of w starting at position 3. Although $\langle 1, 6\rangle$ is a prefix run in v, it does *not* immediately imply that $\langle 3, 8\rangle$ is a run in w, since it is properly included in the run $\langle 1, 8\rangle$ in w. How to avoid duplicated counting of runs *effectively* is the main subject of this algorithm, as well as the subsequent two algorithms. To solve this problem, we focus on the fact that the character a $= w[2]$, that is the left neighbor of the starting position 3, appears in $v = w[3:16]$ at 2, 3, 5, 6, 7, and 8. (The occurrences at 10, 11 and 13 are not important for the purpose.) Even if v has a prefix run of period either $p = 2, 3, 5, 6, 7$, or 8, it *never* becomes a run in w since it continues to the left at least one position. Therefore we have only to consider the periods either 1 or 4 (up to 8). In our bit vector implementation, we have only to initialize *alive* = `00001001`. The bit vector can be easily obtained by the complement of reversal of *occ*[a]$[3:10]$ = `01101111`. Since the reversal operation is required at each starting positions, we compute the bit vectors both *occ*$[c]$ and its reversal *occ_reversal*$[c]$ for each $c \in \Sigma$ in the pre-processing phase, given string w.

Algorithm 1: counting prefix runs at each starting position

```
Input: w, length: string to count runs, and its length.
Result: number of runs in w.
// construct bit vectors representing the occurrences of each character
foreach $c \in \Sigma$ do
    occ[$c$] := 0 ;                                          // occurrence bit vector
    occ_reversal[$c$] := 0 ;                      // reversal of occurrence bit vector
end
for i := 1 to length do
    $c$ := w[i];
    occ[$c$]  := occ[$c$] | (1 << length − i − 1);
    occ_reversal[$c$]  := occ_reversal[$c$] | (1 << $i$);
end
// now count all prefix runs at each beginning position
count := 0;
for begPos := 1 to length − 1 do
    activeArea := 0;
    restLength := length − begPos;
    alive := (1 << (restLength/2)) − 1;
    if begPos > 0 then
        leftChar := w[begPos − 1];
        alive := alive & ((∼occ_reversal[leftChar]) >> begPos);
    end
    for i := 1 to restLength do
        nextChar := w[begPos + i];
        bitmask := ((occ[nextChar] >> (restLength − i)) | (∼0) << i);
        lastAlive := alive;
        alive := alive & bitmask;
        if (lastAlive ^ alive) & activeArea ≠ 0 then
            count++ ;                     // some bit in alive is changed in active area
        end
        if  alive = 0 then  break ;                                  // all runs ended
        if  i mod 2 = 1 then
            activeArea := (activeArea << 1) | 1 ;             // widen active area by one
        end
    end
    if  alive ≠ 0 then
        count++ ;                     // the run is continued to the rightmost position
    end
end
return count
```

The full description of the algorithm is in Algorithm 1. The correctness of the algorithm can be verified based on the above mentioned facts. If the length n of the given string is at most the word size, the running time is $O(n^2)$ with $O(|\Sigma|)$ space.

The time complexity for general n is $O(n^3/m)$, where m is the length of the machine word.

3.2 Efficient algorithms for binary strings

In this section, we take another approach to efficiently count the number of runs for binary strings, given as bit vectors.

We assume that the length of the string does not exceed the word size. For a binary string $w \in \{\mathtt{0}, \mathtt{1}\}^+$ of length n and an integer $p \leq n/2$, we denote by $\alpha(w, p)$ the bit vector whose i-th bit is defined by

$$\alpha(w,p)[i] = \begin{cases} 1 & (w[i-p] = w[i]), \\ 0 & (i \leq p \text{ or } w[i-p] \neq w[i]). \end{cases}$$

For instance, Table 2 shows $\alpha(w, p)$ for $w = \mathtt{1111010101001001}$, who has 5 runs $\langle 1, 4\rangle$, $\langle 11, 12\rangle$, $\langle 14, 15\rangle$, $\langle 4, 11\rangle$, and $\langle 9, 16\rangle$.

										w							
		1	1	1	1	0	1	0	1	0	1	0	0	1	0	0	1
	1	*0*	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0
	2	*0*	*0*	1	1	0	1	1	1	1	1	1	0	0	1	0	0
	3	*0*	*0*	*0*	1	0	1	0	0	0	0	0	1	1	1	1	1
	4	*0*	*0*	*0*	*0*	0	1	0	1	1	1	1	0	0	0	1	0
p	5	*0*	*0*	*0*	*0*	*0*	1	0	1	0	0	0	1	1	1	0	0
	6	*0*	*0*	*0*	*0*	*0*	*0*	0	1	0	1	1	0	0	0	1	1
	7	*0*	*0*	*0*	*0*	*0*	*0*	*0*	1	0	1	0	1	1	1	0	0
	8	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	0	1	0	0	0	0	1	1

Table 2. $\alpha(w, p)$ for $w = \mathtt{1111010101001001}$. In each p-th row, consecutive 1's of length at least p is shadowed. The leading p elements at each p-th row are always *0* regardless of w, and shown in italic form.

We note that $\alpha(w, p)$ can be implemented efficiently by the bit operations

$$(w\hat{\ }((\sim w)\ \texttt{>>}\ p))\ \texttt{\&}\ (1^n\ \texttt{>>}\ p)$$

Notice that for any bit vector x, $x\ \texttt{\&}\ (1^n\ \texttt{>>}\ p)$ sets the p leading bits of x to 0. It is easy to see that the following property holds.

Lemma 1. *For any binary string w of length n, the following two conditions are equivalent.*

1. *$\langle s, t\rangle$ is a run of period p in w.*
2. *$s+2p \leq t+1$, and $\alpha(w, p)[i] = 1$ for every $s+p \leq i \leq t$, and $\alpha(w, p)[s+p-1] = 0$. Moreover, if $t < n$ then $\alpha(w, p)[t + 1] = 0$.*

Since a run must be at least as long as twice its period, $s + 2p \leq t + 1$ holds for any run $\langle s, t\rangle$. Therefore, the lemma states that each run of period p in w corresponds to a stretch of consecutive 1's with length at least p in $\alpha(w, p)$. The problem now is how this can be counted efficiently for each period.

Notice that for any bit vector x, the operation $x\ \texttt{\&}\ (x\ \texttt{>>}\ 1)$ reduces the length of each stretch of consecutive 1's in x by one. Therefore, we can detect stretches of consecutive 1's of length at least p in $\alpha(w,p)$ by counting the number of stretches of 1's in $x = \alpha(w,p)$ `&` $(\alpha(w,p)$ `>>` $1)$ `&` $\cdots$ `&` $(\alpha(w,p)$ `>>` $(p-1))$. It is not difficult to see that calculating x can be done with $O(\log p)$ operations, as shown by `selfAND` in Algorithm 2. Further, the number c of stretches of 1's in a bit vector can be computed in $O(c)$ steps as shown by `oneRuns` in Algorithm 2. Details and alternate implementations for these operations can be found in [15].

However, as with counting prefix runs, we must be careful not to count the same run multiple times. This could happen, for example, when a run is longer than 4 times its minimal period p, and a run would be detected for period p and period $2p$. For example, a run `abababab` with period 2 (`ab`) will also be counted as a run at period 4 (`abab`). Below, we consider two methods for removing such duplicates in the process.

Algorithm 2: Common subroutines

```
selfAND(v,k): calculate v = v&(v>>1)&···&(v>>(k − 1)) (with fewer steps)
begin
    while k > 1 do  s = k >> 1; v & = (v >> s); k −= s;
    return v;
end
oneRuns(v): count the number of stretch of 1's in v
begin
    c:= 0;
    while v do
        v & = (v | (v − 1)) + 1 ;            // remove rightmost stretch of consecutive 1's
        c ++;
    end
    return c;
end
```

Removing duplicate runs by position The first approach utilizes the fact that runs with different minimal periods cannot begin and end at the same positions. Therefore, for each beginning position of a run, we use a bit vector to mark its end position. This way, we can check if we have already considered the run via a different period. This can be implemented efficiently using bit operations as shown in Algorithm 3. The time complexity is $O(n^3/m)$, where m is the length of the machine word. Note that `lsb(x)` computes the least significant set bit of `x`, and can be computed in $O(1)$ for a machine word, or $O(n/m)$ time for general bit strings.

Removing duplicate runs by sieve The second approach to eliminate duplicate counting is based on the following observation.

Lemma 2. *Let $\langle s,t\rangle$ be a run of period p in w. For any k with $2kp \leq t-s+1$, the run $\langle s,t\rangle$ is also a run of period kp in w.*

For example, consider again the runs in $w =$ `1111010101001001` (see Table 2). The run $\langle 1,4\rangle =$ `1111` is a run of period both 1 and 2. The run $\langle 4,11\rangle =$ `10101010` is a run of period both 2 and 4.

Therefore, if we count each run only at its minimum period, we can avoid duplications. Our strategy is somewhat similar to the *Sieve of Eratosthenes* to generate prime numbers. From the smallest period $p = 1$ to the maximum possible period $p = |w|/2$ in this order, if a run of period p is found in x, we will sieve out all runs of period kp satisfying the length condition in Lemma 2. The sieving procedure can be implemented by tricky bit operations, shown in Algorithm 4. The time complexity is $O(n^3/m)$, where m is the length of the machine word.

Algorithm 3: Removing duplicate runs by position

```
Input: w, length: bit vector to count runs, and its length.
Result: number of runs in w.
runEndsByBegPos[length − 1] ;                     // array of bitvectors (initialized to 0)
for period := 1 to length/2 do
    v:= (w ^ ((~w) >> period)) & ($1^{\mathrm{length}}$ >> period) ;      // calculate $\alpha$(w, period)
    x:= selfAND(v, period) ;
    while x $\neq$ 0 do
        begPos:= lsb(x) ;                          // position index of rightmost 1
        y:= x + (1 << begPos) ;                    // if x =...0111100 then y = ...1000000
        x:= x & y ;                                // clear rightmost consecutive 1's in x
        y:= y & (−y) ;                             // clear all but rightmost 1 in y
        y:= y << ((period − 1) << 1) ;             // convert to actual position in w
        if (runEndsByBegPos[begPos] & y) = 0 then
            // a run starting at begPos doesn't already end here
            count ++;
            runEndsByBegPos[begPos] = runEndsByBegPos[begPos] | y;
        end
    end
end
return count
```

Algorithm 4: Removing duplicates by Sieve

```
Input: w, length: bit vector to count runs, and its length.
Result: number of runs in w.
pvec[length/2 + 1] ;                               // array to store bitvectors
for period := 1 to length/2 do
    pvec[period] := (w^((~w) >> period)) & ($1^{\mathrm{length}}$ >> period) ;   // calculate $\alpha$(w, period)
end
for period := 1 to length/2 do
    x := selfAND(pvec[period], period);
    count := count + oneRuns(x) ;                  // number of runs of this period.
    for p := 2 ∗ period to length/2 step period do
        x := x & (x >> period);
        if x = 0 then break;
        pvec[p] := pvec[p]^x;
    end
end
return count
```

4 Computational Experiments

4.1 Running time

Table 3 compares the running times of the three algorithms. All experiments were conducted on an Apple Mac Pro (Early 2008) with 3.2 GHz dual core Xeons and 18 GB of memory, running MacOSX 10.5 Leopard, using only one thread. Programs were compiled with the Intel C++ compiler 11.0. The algorithms were run on all binary strings of length n for $n = 20, \ldots, 30$. However, only strings ending with 0 are considered, since a complementary binary string will always have runs in the same position. That is, all runs for 2^{n-1} strings are calculated for each n.

For the sieve approach, we further developed a GPGPU version and measured its performance on the same computer. The video card used for GPGPU was NVIDIA GeForce 8800 GT, and the GPGPU environment used was CUDA[2]. Although GPUs

[2] http://www.nvidia.com/object/cuda_home.html

contain many processing units, there are very strict limitations to the resources that each processing unit may use. The simple bit-parallel algorithm presented in this paper only requires a small amount of resources, and is an ideal example for efficient processing on GPUs. We note that the sieve approach using GPGPU was developed to deal with 64 bits, and requires some overhead compared to the other three algorithms that were developed for only 32 bits.

n	20	21	22	23	24	25	26	27	28	29	30
prefix	0.32	0.69	1.49	3.13	8.02	15.6	32.4	66.4	150.2	296.5	625.4
position	0.09	0.18	0.36	0.73	1.49	3.0	6.2	12.6	25.6	52.1	106.0
sieve	0.10	0.18	0.37	0.75	1.50	3.0	6.0	11.9	23.9	48.1	96.7
GPGPU	0.01	0.02	0.04	0.08	0.18	0.4	0.7	1.4	3.0	5.9	12.2

Table 3. Running times in seconds of each algorithm for calculating the runs in all binary strings (excepting complementary strings) of length n.

4.2 Results

Using the GPGPU implementation, we computed the maximum number of runs function $\rho(n)$ for binary strings of length up to $n = 47$, as shown in Table 4. It has been known that $\rho(14) = \rho(13) + 2$. However, as noted in [5], it is not known whether this is an asymptotic property of $\rho(n)$, that is, if there exists infinitely many n for which $\rho(n + 1) = \rho(n) + 2$. To the best of our knowledge this is the second example satisfying this property, namely, $\rho(42) = \rho(41) + 2$, provided that $\rho(n)$ is achieved by binary words.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$\rho(n)$	1	1	2	2	3	4	5	5	6	7	8	8	10	10	11	12	13	14	15	15	16	17	18	19	20	21	22	23	24	25	26	27	27	28	29	30	30	31	32	33	35	35	36	37	38	38

Table 4. The maximum number of runs function $\rho(n)$ for binary strings calculated for n up to 47.

Figure 2 plots the maximum number of runs function obtained by our exhaustive computation, the conjectured upper bound ($y = x$) and the current best asymptotic lower bound ($y = 0.94457571235x$).

Although the problem of finding the maximum number of runs function is still difficult, we have found the following empirical characteristics in the distribution of the number of runs in binary strings, which could give insight in further analyses. Let $f(n, r)$ denote the number of binary strings of length n with r runs. Table 5 shows the values of $f(n, r)$ for $n = 2, \ldots, 42$ and $r = 1, \ldots, 4$.

- $f(n, 1) = 20$ for $n \geq 7$.
- for $n \geq 7$,

$$f(n, 2) = \begin{cases} 36n - 190 & \text{if } n \text{ is even,} \\ 36n - 186 & \text{if } n \text{ is odd.} \end{cases}$$

Furthermore, $f(n, 2) = f(n - 2, 2) + 72$ for $n \geq 9$.

n	$f(n,1)$	$f(n,2)$	$f(n,3)$	$f(n,4)$
2	2	0	0	0
3	6	0	0	0
4	14	2	0	0
5	18	14	0	0
6	18	38	8	0
7	20	66	38	4
8	20	98	102	34
9	20	138	202	130
10	20	170	376	306
11	20	210	596	682
12	20	242	880	1314
13	20	282	1220	2296
14	20	314	1622	3736
15	20	354	2080	5686
16	20	386	2598	8260
17	20	426	3174	11562
18	20	458	3808	15642
19	20	498	4502	20626
20	20	530	5252	26574
21	20	570	6064	33590
22	20	602	6930	41754
23	20	642	7860	51184
24	20	674	8842	61898
25	20	714	9890	74070
26	20	746	10988	87732
27	20	786	12154	103000
28	20	818	13368	119922
29	20	858	14652	138664
30	20	890	15982	159216
31	20	930	17384	181764
32	20	962	18830	206308
33	20	1002	20350	233012
34	20	1034	21912	261896
35	20	1074	23550	293138
36	20	1106	25228	326696
37	20	1146	26984	362804
38	20	1178	28778	401434
39	20	1218	30652	442762
40	20	1250	32562	486776
41	20	1290	34554	533702
42	20	1322	36580	583470

Table 5. Number of binary strings of length n with r runs.

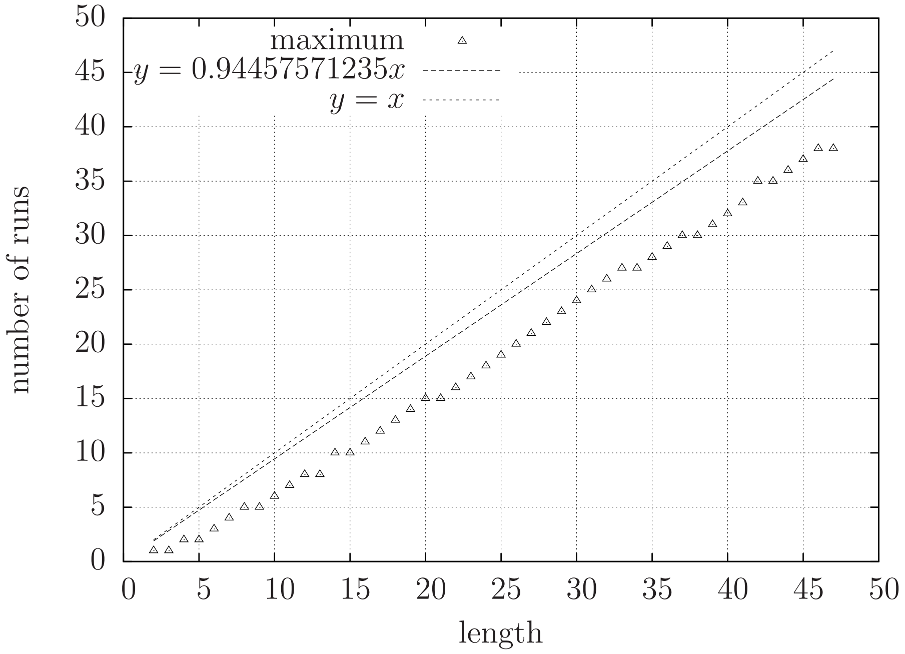

Figure 2. The maximum number of runs in a binary string obtained from exhaustive calculations.

– for $n \geq 12$,

$$f(n,3) = \begin{cases} (117n^2 - 1558n + 5368)/4 & \text{if } n \text{ is even,} \\ (117n^2 - 1556n + 5335)/4 & \text{if } n \text{ is odd.} \end{cases}$$

Furthermore, $f(n,3) = 2f(n-2,3) - f(n-4,3) + 234$ for $n \geq 16$.

We can see that $f(n,1) = 20$ will hold for any $n > 7$, since a binary string with only one run can only be one of $(01)^{n/2}$, $x0^{n-4}y$ for $x, y \in \{00, 01, 10\}$, or their bitwise complements.

5 Discussion and Conclusion

We presented 3 bit-parallel algorithms for computing all the runs in short strings. The two latter algorithms specialized for binary strings are very efficient, while the first algorithm can be used for strings with larger alphabet size at the cost of some efficiency. Through exhaustive computations, the algorithms have enabled us to obtain various statistics concerning runs in strings up to a certain length.

Although it seems that many researchers believe it to be true, it is still unknown whether $\rho(n)$ can always be achieved by a binary string.

References

1. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Comm. ACM, 35(10) 1992, pp. 74–82.
2. G. Chen, S. Puglisi, and W. Smyth: *Fast and practical algorithms for computing all the runs in a string*, in Proc. CPM 2007, vol. 4580 of LNCS, 2007, pp. 307–315.

3. M. Crochemore and L. Ilie: *Maximal repetitions in strings.* J. Comput. Syst. Sci., 74 2008, pp. 796–807.
4. F. Franěk, R. Simpson, and W. Smyth: *The maximum number of runs in a string*, in Proc. 14th Australasian Workshop on Combinatorial Algorithms (AWOCA2003), 2003, pp. 26–35.
5. F. Franěk and Q. Yang: *An asymptotic lower bound for the maximal-number-of-runs function*, in Proc. Prague Stringology Conference (PSC'06), 2006, pp. 3–8.
6. M. Giraud: *Not so many runs in strings*, in Proc. LATA 2008, 2008, pp. 245–252.
7. R. Kolpakov and G. Kucherov: *Maximal repetitions in words or how to find all squares in linear time*, Tech. Rep. Rapport Interne LORIA 98-R-227, Laboratoire Lorrain de Recherche en Informatique et ses Applications, 1998.
8. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS'99), 1999, pp. 596–604.
9. W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara: *A series of run-rich strings*, in Proc. LATA 2009, vol. 5457 of LNCS, 2009, pp. 578–587.
10. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *New lower bounds for the maximum number of runs in a string*, in Proc. Prague Stringology Conference (PSC'08), 2008, pp. 140–145.
11. S. J. Puglisi and J. Simpson: *The expected number of runs in a word.* Australasian Journal of Combinatorics, 42 2008, pp. 45–54.
12. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006), vol. 3884 of LNCS, 2006, pp. 184–195.
13. W. Rytter: *The number of runs in a string.* Inf. Comput., 205(9) 2007, pp. 1459–1469.
14. J. Simpson: *Modified padovan words and the maximum number of runs in a word.* Australasian Journal of Combinatorics, to appear.
15. H. S. Warren: *Hacker's Delight*, Addison-Wesley Professional, 2002.
16. S. Wu and U. Manber: *Fast text searching allowing errors.* Comm. ACM, 35(10) October 1992, pp. 83–91.