Practical Space-efficient Linear Time Construction of FM-index for Large Genomes

Elena Y. Harris
California State University, Chico
Chico, California, 95929, USA
eyharris@csuchico.edu

Abstract
The Burrows-Wheeler Transform (BWT) and Full-text index in Minute space (FM-index) are indispensable data structures that are used in the next generation sequencing data analysis to efficiently map reads to a reference genome. Recently developed algorithms SA-IS and BWT-IS allowed construction of a Suffix Array and Burrow-Wheeler Transform, respectively, for mammalian-size genomes in less than an hour. In practice, BWT-IS algorithm outperforms SA-IS in terms of RAM usage. Building an FM-index from a BWT requires LF-mapping that is a relatively time-consuming step. Here, we present a space-efficient linear time algorithm called BWT-ISFM that builds an FM-index concurrently with a construction of BWT. Our algorithm supports a genome size of up to 8Gb (giga-basepairs) in length while a publicly available BWT-IS has a limit on a genome size of 4Gb. Moreover, in practice, our algorithm requires only 2.3n bytes of RAM for a genome size of n compared to 4n bytes of RAM used by SA-IS algorithms.

keywords: next generation sequencing; BWT; FM-index

1 Introduction
The Burrows-Wheeler Transform, BWT, introduced by Burrows and Wheeler [1] together with a Full-text index in Minute space, FM-index, proposed by Ferragina and Manzini [3], are used to align sequenced reads generated by the next generation sequencing instruments to a reference genome. Suffix Array, SA, a data structure introduced by Manber and Myers [9] is a concise representation of sorted suffixes of a given string T. Given a string T, SA is an array that stores positions of the suffixes of T sorted in lexicographic order. Given a SA, BWT can be constructed in linear time by scanning the SA and retrieving characters of T at the previous positions of the stored in the SA positions. Recent advances in linear time construction of a SA (see the survey by Dhamiwal et al. [2]) allow building a SA for large mammalian genomes such as human genome in tens of minutes using practically affordable space of about 15GB. It has been shown in Harris et al. [4] that building the BWT on two strands of a genome, positive strand and negative strand, speeds up read alignment two times. In this case and, in general, for larger genomes, space of at least 4n bytes (hereafter, n is the length of a genome) required by the most time-efficient SA algorithms, becomes not so practical. For example, building a SA over two strands of a human genome of 6Gb would require at least 24GB.

Only few aligners use a SA to map sequenced reads to a reference genome, e.g. Hoffmann et al. [5]. Most aligners rely on using BWT together with a FM-index. Recently, a linear time algorithm BWT-IS for building BWT directly was developed by Okanohara and Sadakane [11]. It extends on the ideas of the recursive linear time SA-IS algorithm by Nong et al. [10] that uses induced sorting to build a suffix array. In practice, BWT-IS requires only up to 2.2n bytes of memory, which is a significant advantage over 4n required for building a SA. Building BWT directly (without building a SA) for a human genome size is also time efficient – it takes less than a half of an hour on a human genome of size 3Gb. To build an FM-index from BWT, first, a succinct data structure is built in linear time that stores for each position i of BWT the total number of occurrences of each character of the given alphabet in BWT from the beginning to i. This structure is used in the next step, LF-mapping, to retrieve and explicitly store genomic positions equidistant from each other at a fixed range, called step. In practice, O(n)-time LF-mapping is relatively time consuming (compared to the building of the BWT). Some algorithms (e.g. by Kärkkäinen [6]) build BWT and SA in consecutive blocks, so the genomic positions for each block are available for output on the fly. The downside of this approach is Ω(nlogn) time requirement for building the BWT and FM-index. The linear time algorithm BWT-IS does not store genomic positions while constructing a BWT. In order to construct an FM-index, we must first build BWT and then use LF-mapping to construct an FM-index, which requires additional time. Moreover, the existing implementation of BWT-IS provided by the authors and publicly available implementations of SA-IS have a limitation on the genome size of less than 4Gb.

Here, we propose a practical, space-efficient, linear time algorithm, BWT-ISFM, that calculates the FM-index while building the BWT directly from a given genome. For two-strands of human genome of 6Gb, it requires only 2.3n bytes of memory and builds the BWT and FM-index in less than an hour. Our algorithm supports genome size of up to 2^{33} characters long, 8Gb. In addition, it introduces a new algorithmic approach to optimize one step of SA-IS algorithm that may be used for a pool of ideas on optimization of BWT and SA constructions for small,
constant alphabets. Thus, our algorithm offers a memory advantage over SA-IS algorithms, extends genome size limitation, and constructs an FM-index on the fly while building the BWT in practical time.

2 Preliminaries and Related Work

Let T be a string of length n over the given ordered alphabet Σ of size σ. Let $ be the sentinel character that is the smallest in Σ and occurs in T only once at the end of the string. Let T[i] be a character of T, for 0 ≤ i ≤ n-1, and T[i…j] be a substring of T of consecutive characters starting with character T[i] and ending with T[j]. A suffix of T, denoted by T_p, is a substring T[i…n-1], i.e. T_i ends with $. A suffix array SA for a given string T is an array of size n such that SA[i] stores the starting position p of suffix T_p for 0 ≤ i ≤ n-1 and T_SA[0] < T_SA[1] < …< T_SA[n-1]. In other words, SA holds the starting positions of all suffixes of T sorted in lexicographic order. Given a string T of length n over alphabet Σ of size σ, a suffix array can be built in linear time. Some of the recently developed algorithms for SA construction that use recursion and induce sorting of suffixes are by Ko and Aluru [7] and by Nong et al. [10]. Hereafter, we will refer to the algorithm by Nong et al. [10] called SA-IS in our detailed discussion.

SA-IS uses a concept of LMS substrings to induce-sort suffixes of a given string T. To understand an LMS substring, we need to categorize characters of T by L-type and S-type (stands for Large and Small). The sentinel character $ is of S-type. For 0 ≤ i ≤ n-2, character T[i] is of S-type if it is lexicographically smaller than the next character T[i+1] and T[i] is of L-type if it is greater than T[i+1]. If T[i] and T[i+1] are equal, then T[i] has the same type as T[i+1]. Enumeration of characters of T by S- or L-type can be done in linear time by scanning T in Right-to-Left fashion. An LMS character stands for the left most S-type character and it is character T[i] of S-type that has previous character T[i-1] of L-type. Character T[0] is considered to have sentinel as the previous character, so T[0] cannot be an LMS character. Further, suffixes of T are named L-type, S-type and LMS-type after their first character’s type, e.g. if T[i] is of L-type, then T_i is an L-type suffix. An LMS substring of T is a substring T[i…j] such that T[i] and T[j] are LMS characters and no other character between indices i and j are LMS characters. The sentinel character is the only LMS substring of length one.

When suffixes of T are arranged in lexicographic order, suffixes starting with the same character c occur in consecutive range, and their positions are stored in consecutive entries of SA. We will refer to the consecutive range SA[i…j] that stores starting positions of all suffixes of T starting with character c as a c-bucket. Sorted in lexicographic order suffixes in the same c-bucket have the following order: L-type suffixes precede S-type suffixes in the c-bucket (please refer to the original paper for proof of this and other statements regarding SA-IS algorithm).

Furthermore, SA-IS algorithm uses a notion of fronts and ends of c-buckets. If a SA[i…j] is a c-bucket, initially, index i is the head of c-bucket pointing to the front of the bucket and index j is the tail of c-bucket pointing to the end of the bucket; as entries of SA fill in, the heads and the tails of the buckets are incremented and decremented respectively.

Figure 1 shows the outline of SA-IS algorithm and Figure 2 demonstrates the execution of the steps of SA-IS for a string T='DABRACADABRACABRAB$. Given a string T, SA-IS calculates the type array t that stores S- and L-type for characters of T. Using t, it identifies the starting positions of LMS substrings and places them into ends of the corresponding c-buckets. This is done in linear time by scanning t Right-to-Left (see Figure 2, A and B). Next, InduceSort(T, SA, t) procedure consists of two steps: (1) Left-to-Right scanning of SA with head pointers initialized to the fronts of the corresponding c-buckets, and (2) Right-to-Left scanning of SA with tail pointers initialized to ends of the corresponding c-buckets. During Left-to-Right scanning of SA, for each position p at SA[i], it checks whether the character T[p-1] at the previous position p – 1 is of L-type, and if so, it places position (p – 1) into the current front of the c-bucket, where c is T[p – 1], and increments the head of c-bucket (see Figure 2 C).

During Right-to-Left scanning of SA, for position p at SA[i], it checks whether the character T[p – 1] at the previous position p – 1 is of S-type, and if so, it places position (p – 1) into the current end of the c-bucket, where c is T[p – 1], and decrements the tail of c-bucket. By the end of this step, LMS substrings are correctly sorted in lexicographic order relative to each other.

Figure 2 D shows SA after this step and shows LMS substrings sorted in lexicographic order relative to each other.

**ALGORITHM 1:** SA by induced sorting

0: **Input:** string T of length n over alphabet Σ of size σ

1: **Output:** Suffix Array for T

1: Check for termination condition: if n is equal to σ, calculate SA directly

2: Calculate S/L-type array t

3: Place the starting positions of LMS substrings into the ends of c-buckets of SA

4: InduceSort(T, SA, t)

5: Assign names to LMS substrings

6: Build the shortened string T_1

7: Recursively calculate SA_1 for T_1

8: Induce positions of LMS substrings of T from positions of suffixes T_1 stored at SA_1

9: Place the starting positions of LMS substrings in the sorted order in their corresponding c-buckets of SA

10: InduceSort(T, SA, t)

11: return: SA

Figure 1: SA-IS outline
The next step of SA-IS is to assign a new integer-name to each LMS substring. The sentinel LMS is assigned name 0. The naming of the rest of LMS substrings is done by Left-to-Right scanning of SA, and comparing two consecutive LMS substrings: if two substrings are the same, then they are assigned the same name, otherwise, the current LMS is assigned the next integer-name than the previous LMS.

Finally, to build a shortened string T₁ that consists of integer-names of LMS substrings of T, the integer-names must be placed in the same order as their corresponding LMS substrings occur in T. To clarify, if the i-th LMS substring of T has been assigned integer-name dᵢ and the k-th LMS substring of T has been assigned integer-name dₖ, then character Tᵢ[i] = dᵢ and Tᵢ[k] = dₖ. Figure 1 E shows the resulting T₁ for the given T in our example.

To make sure this step is done in linear time, the original algorithm [10] proposed to keep a bit array of length n with 1s denoting the starting positions of LMS substrings. In addition, a succinct data structure supporting Rank(i) operation must be prebuilt in linear time, where given the starting position i of an LMS substring in T, Rank(i) returns the rank of the LMS substring in T (the order of the LMS substring in T from left to right).

Thus, once the LMS substring T[i...j] is assigned name d, we set T₁[Rank(i)] = d.

The recursive call to SA-IS on the shortened string T₁ returns the suffix array SA₁ for T₁. The next step of the algorithm is to induce the positions of LMS substrings of T from the suffix positions of T₁ stored in SA₁. Since each character Tᵢ[j] corresponds to the j-th LMS substring of T, we can convert positions of suffixes of T₁ to starting positions of the corresponding LMS of T as follows. Let j = SA₁[i] be the position of the suffix of T₁, starting with T₁[j], which corresponds to the j-th LMS in T. Then we can use a prebuilt in linear time succinct data structure that supports operation Select(j) that returns the position of the j-th LMS in T in constant time, given j. After inducing the starting positions of the LMS substrings in T from suffix positions of T₁, we have positions of LMS suffixes of T sorted in lexicographic order and place them into SA at the ends of the corresponding c-buckets.

The last step InduceSort is used again to induce SA from the LMS positions. Each step of SA-IS takes linear time. Since two LMS characters cannot be consecutive characters by definition, then the length of T₁ is at most n/2, half of the length of T, and, hence, the recursive algorithm SA-IS takes linear time.

Figure 2: Execution of Algorithm 1 applied to the given string T='DABRACADABRACABRAB$'. (A) Type array t is shown for T and the starting positions of LMS substrings are marked with * character. (B) Suffix array SA is shown after the starting positions of LMS substrings have been placed at the end of the c-buckets, where c is the starting character of an LMS. (C) SA is shown after Left-to-Right scanning of SA and after the order of L-type suffixes has been induced from LMS and L-type suffixes. The induced L-type suffix positions are shown in bold. (D) SA is shown after Right-to-Left scanning and after the order of S-type suffixes has been induced from L-type and S-type suffixes. The induced S-type suffix positions are shown in bold. (E) The shortened string T₁ is shown: each integer-character Tᵢ[i] corresponds to the i-th LMS substring of T, whose names have been assigned according to their lexicographic order in SA.
BWT-IS simulates *InduceSort* procedure with the help of four queues: LMS, L, S and LS (each of the four queues for each character in the alphabet). Instead of keeping a suffix array of size $n$ that holds positions of all suffixes, BWT-IS keeps LMS substrings of $T$ directly and uses circular shift of characters in LMS so that the front character directs the next step of induce-sorting. Initially all LMS substrings are reversed and placed into LMS queues (by their last character), e.g. the reverse of an LMS $T[i...j]$ is stored in LMS$_{[j]}$ queue for $T[j]$ character.

Left-to-Right scanning of *InduceSort* procedure processes $c$-buckets in increasing order of characters $c$.

BWT-IS simulates this step by considering characters of the alphabet in increasing order, and for each character $c$, first, it processes $L_c$ queue and then $LMS_c$ queue (just as SA-IS algorithm processes L-type suffixes and then LMS suffixes of a $c$-bucket). While $L_c$ queue is not empty, a current LMS is popped at the front of the queue, and if the front character $b$ of the current LMS substring is greater than or equal to $c$, then $b$ is of L-type, so the LMS substring is pushed to the back of $L_b$ queue (in this case, $b$ is shifted to the back of the LMS substring). Otherwise, the LMS substring is pushed to the back of LS$_c$ queue. This simulates processing of L-type suffixes of a $c$-bucket. Next, while $LMS_c$ queue is not empty, a current LMS substring is popped from the front of the queue, and the front character $b$ of the current LMS substring is placed to the back of the LMS substring. The character $b$ is of L-type, so the LMS substring is pushed to the back of the L$_b$ queue. These two steps simulate Left-to-Right scanning of SA and induce sorting of L-type suffixes.

The Right-to-Left scanning of SA of SA-IS algorithm is simulated by processing characters of the alphabet in decreasing order. Prior to this step, LS queues are reversed. Next, for each character $c$, first $S_c$ queue is processed and then $LS_c$ queue is processed. In SA-IS this corresponds to processing of S-type suffixes and then L-type suffixes of a $c$-bucket during Right-to-Left scanning of SA. While $S_c$ queue is not empty, pop a current LMS substring from the front of the queue, move the front character $b$ of the LMS substring to the back of the LMS substring, and if $b$ is less than or equal to $c$ (i.e. $b$ is of S-type), then push the LMS substring onto the back of $S_b$ queue. Next, while $LS_c$ is not empty, pop a current LMS substring from the front of the queue, move the front character $b$ to the back of the substring, and push LMS substring onto the back of $S_b$ queue. During these movements involving the four types of queues, BWTs of L-type suffixes and of S-type suffixes are built separately for each character $c$, and at the end of the algorithm, BWT for $T$ is constructed from these shorter BWT substrings.

Hereafter, we will refer to BWT-IS as it is implemented by the authors of the original paper. Given a string $T$, (1) BWT-IS sorts LMS substrings using *quick sort* and builds a shortened string $T_1$, then (2) calls SA-IS algorithm to recursively calculate $SA_i$ for $T_1$; (3) deduces positions of the LMS substrings of $T$ from positions of suffixes of $T_1$ in $SA_i$; and finally, (4) simulates *InduceSort* using the four types of queues to build the BWT for $T$. BWT-IS saves space by avoiding storing SA for $T$.

In the presented here algorithm, we use BWT-IS as the basis for our algorithm BWT-ISFM. The major difference between the proposed algorithm and BWT-IS is that we keep the starting positions of LMS suffixes in the queues instead of LMS substrings. This allows accessing induced-sorted positions of suffixes directly on the fly while constructing the BWT, which allows building an FM-index on the fly. In addition, our implementation of sorting distinct LMS substrings and building a shortened $T_1$ of the given string $T$ differs from the BWT-IS’s implementation. The rest of the paper is organized as follows. In section 3, we describe our algorithm and analyze its time requirements. In section 4, we convey a benchmarking that demonstrates the performance of our algorithm in terms of time and RAM, and compares it with the performance of BWT-IS and existing FM-index building tools: the most popular tool Bowtie-2 by Langmead et al. [8] and another tool called BRAT-nova by Harris et al. [4], an aligner for bisulfite-treated reads used to identify methylation within a DNA sequence.

### 3 Construction of BWT and FM-Index

#### 3.1 Implementation of Induce Sorting and Calculation of Explicitly Stored Positions

FM-index constructed by our algorithm BWT-ISFM consists of (1) a succinct data structure *Character Occurrences* that for each character $c$ in $\Sigma$ allows calculating of the total number of occurrences of $c$ in $\text{BWT} [0...i]$ in constant time, for $0 \leq i \leq n-1$; (2) a succinct data structure *Positions Occurrences* that consists of a bit array called $\text{bwtMarked}$ (with bit 1 at index $i$ indicating that the suffix position corresponding to $SA[i]$ is explicitly stored) together with a succinct structure that calculates $\text{Rank}(i)$ in constant time; this structure is used to retrieve an explicitly stored genomic position; (3) an array called *Positions* with explicitly stored suffix positions. In addition to this classical FM-index, our program constructs a bit array called $\text{posBit}$ with 1 at index $i$ indicating that stored position corresponding to $SA[i]$ is greater than maximum value of an unsigned integer, $\text{MAXUI} = 2^{32} - 1$. 

The array Positions stores unsigned integers (requiring 4B per integer). To retrieve the correct position, algorithm uses 4B stored at Positions and one bit stored at posBit: if a bit at posBit is 1, then to the value stored at Positions, we need to add MAXUI.

Our algorithm follows the outline of BWT-IS. Our algorithm uses four types of deques: LMS, L, S and LS (similarly to the queues used by BWT-IS described above). Each deque supports four operations: push front and push back (inserts a suffix position at front and back respectively), and pop front and pop back (removes a suffix position from the front and back of a deque respectively). At any time of the algorithm, there are at most $X$ positions stored in all deques, where $X$ is the total number of LMS substrings of the original string $T$.

The procedure InduceSort of BWT-ISFM is shown in Figure 3. As in BWT-IS, InduceSort is used only once. We keep BWT array of length $2n$ bits (2 bits per character). Initially, the starting positions of sorted LMS substrings of $T$ are pushed back onto the corresponding LMS, deques for each character $c$, the starting character of an LMS substring; and the head pointers for each character $c$ in $\Sigma$ are set to the fronts of corresponding $c$-buckets. The moves of LMS positions between the deques exactly simulate induce sorting using an SA in SA-IS algorithm. First, characters of alphabet are processed in increasing order in the first for loop (simulating processing of c-buckets in Left-to-Right order), and then characters are processed in decreasing order in the second for loop (simulating processing of c-buckets in Right-to-Left order). In the first for loop, L_deque is processed before LMS_deque. In each of these dequees, a current front position $p$ is popped from a deque, and if the character $T[p-1]$ is greater or equal to $T[p]$, then $T[p-1]$ is of L-type, and position $(p – 1)$ is pushed back onto L$\text{[}p-1\text{]}$ deque. The corresponding BWT character is calculated as $T[p-2]$. In addition, we check whether the position $(p – 1)$ is a position that we explicitly store (mod step is equal to 0, where step is log(n)). If so, then we output to a file the position $p – 1$ and the BWT index, head$\text{[}p-1\text{]}$. In addition, we set bits of bwtMarked (if position $p – 1$ is explicitly stored) and posBit (if $p – 1$ is greater than MAXUI) to 1. At the end, the head pointer is incremented. In case when the front of $L_c$ deque induces position $p – 1$ that corresponds to S-type suffix, then our algorithm pushes $p$ to the back of LS$\text{[}p\text{]}$ deque.

To simulate Right-to-Left scanning of SA of SA-IS algorithm, our algorithm sets tail pointers to the back of c-buckets for each character $c$ in $\Sigma$. Next, BWT-ISFM processes the second for loop, in which for each character $c$ taken in decreasing order, S, deque is processed first and then LS, deque is processed. Until a deque is not empty, a current suffix position $p$ from the back of the deque is popped (scanning Right-to-Left), and if the previous position, $p – 1$, is of S-type, then the position $p – 1$ is pushed to the front of S$\text{[}p-1\text{]}$ deque. In addition, BWT character $T[p-2]$ is set at index tail$\text{[}p-1\text{]}$ and if needed bits of

**Algorithm 2: InduceSort of BWT-ISFM**

0: Input: string $T$ and for each character, four types of deques: LMS, L, S and LS; LMS deques are initialized with positions of LMS substrings

Output: BWT, bwtMarked, posBit, explicitly stored positions of suffixes of $T$

1: for each character $c := 0,1,2,\ldots,\sigma-1$, do
2:    $Q_c := L_c, LMS_c$
3:    while $Q_c$ is not empty do
4:       $p \leftarrow Q_c.popFront()$
5:       if $T[p] \leq T[p-1]$ then
6:          $L_{[p-1]}\.pushBack( p-1 )$
7:       else if $(p-1) \mod step = 0$ then
8:          bwtMarked[head$\text{[}p-1\text{]}$] := 1
9:       end of if
10:      $Q_c.popFront()$
11:    end of while
12: end of for

end of for

17: for each character $c := \sigma-1, \ldots, 2, 1, 0$, do
18:    $Q_c := S_c, L_c$
19:    while $Q_c$ is not empty do
20:       $p \leftarrow Q_c.popBack()$
21:    if $T[p-1] \leq T[p]$ then
22:       $S_{[p-1]}\.pushFront( p-1 )$
23:       BWT[tail$\text{[}p-1\text{]}$] := $T[p-2]$
24:    else if $(p-1) \mod step = 0$ then
25:       bwtMarked[tail$\text{[}p-1\text{]}$] := 1
26:    end of if
27:    if $(p – 1) > MAXUI$ then
28:       posBit[tail$\text{[}p-1\text{]}$] := 1
29:    tail$\text{[}p-1\text{]}$ := tail$\text{[}p-1\text{]}$ – 1
30: end of while
31: end of for
32: end of for

Figure 3: InduceSort of BWT-ISFM

bwtMarked and posBit at index tail$\text{[}p-1\text{]}$ are set to 1, and an explicitly stored position $p – 1$ together with index tail$\text{[}p-1\text{]}$ are printed to a file. At the end, the tail pointer is decremented.

Once InduceSort procedure is finished, we need to place explicitly stored SA positions in the correct order, i.e. in increasing order of BWT indices. First, BWT-ISFM pre-builds in linear time the succinct data structure Positions Occurrences using bwtMarked. Recall that given a BWT index $i$, this structure calculates Rank(i) in constant time. Then, our algorithm reads in the outputted to the file
indices one at a time (suffix position and the corresponding BWT index). Using the BWT index \(i\), it places the corresponding suffix position into the array \(Positions\) at index \(Rank(i)\). We chose to output positions and their BWT indices into the file to save memory. There are total of \(n/\log(n)\) explicitly stored positions, and we need \(\log(n)\) bits to store each position or BWT index. Hence, the total space to store positions and BWT indices would be \(2\log(n)\times n/\log(n)\), which results in \(2n\) bits. In case of keeping these positions and indices in memory, I/O operations would not affect linear time of the algorithm. In practice, I/O operations do not add much to the total time (not more than about 3 minutes in our experiments – results are not shown), but printing out positions and BWT indices allows saving memory for larger genomes and offers extra flexibility for users (our program allows users to select the value for \(step\), which regulates the total number of explicitly stored positions).

3.2 Sorting Distinct LMS Substrings, Assigning Names and Building a Shortened String \(T_1\)

In SA-IS, the \(InduceSort\) procedure is used for the first time to sort LMS substrings in lexicographic order relative to each other. One can use \(InduceSort\) that guarantees theoretical linear time, but in practice, another method to sort all distinct LMS substrings is much faster. For example, in our experiments on a human genome using two strands, \(InduceSort\) takes about 20 minutes, whereas sorting distinct LMS substrings using a \(quick sort\) and building a shortened \(T_1\) string takes less than 2 minutes. Here, we will describe our method of building a shortened \(T_1\) using a \(quick sort\). We used ideas similar to those of the BWT-IS algorithm, but devised a different implementation for this procedure.

First, we collect the starting positions of distinct LMS substrings into a separate array. LMS are categorized as short substrings and long substrings dependent on their lengths. An LMS substring of length at most 12 characters is considered to be short, and the rest of LMS substrings are long. We would like to clarify that by the term all \(distinct\) LMS substrings we mean all long LMS substrings and distinct short LMS substrings. To identify the starting positions of distinct short LMS substrings, we keep a hash table of size \(4^{12}\) entries, where \(4\) is the size of the DNA alphabet \(\{A, C, G, T\}\). The length 12 for a short LMS substring was chosen to keep a good balance between the space required for the hash table and the total number the long LMS substrings that is at most \(n/12\). Each character is represented using 2 bits (A is 00, C is 01, G is 10 and T is 11). We scan \(T\) Right-to-Left, and if \(T[i]\) is the starting character of a short LMS substring \(s\), then we convert \(s\) to its hash index \(h\) equal to the complement of \(s\). We must use the complement of \(s\) to distinguish between LMS substrings such as AATA and ATA; since A is represented as 0 in binary, both strings AATA and ATA in binary are represented as the same integer, namely, 12 (00001100 and 001100 respectively). By taking the complement of these strings, their binary representations become distinguishable (11110011 and 110011 respectively). For all identical short LMS substrings whose hash index is \(h\), we store a single position that is the greatest. For example, if \(T[i...i+m]\) and \(T[j...j+m]\) correspond to a short LMS substring \(s\), and the position \(i\) is greater than \(j\), then we store \(i\) at index \(h\) in the hash table and in the array with the distinct LMS positions. If a current LMS is long, we store its starting position in the array with distinct positions.

Once the positions of distinct LMS substrings are collected, we sort the distinct LMS substrings using a \(quick sort\). Depending on implementation, theoretically, it takes \(\Omega(n\log n)\) time, but in practice, it is much faster than \(InduceSort\) that takes theoretical linear time (e.g. it takes on average 9 seconds to sort all distinct LMS substrings of two strands of human genome). Next, we assign names to the distinct LMS substrings by scanning the sorted array and comparing two consecutive LMS substrings. If the current LMS substring is the same as previous one, it is assigned the same integer-name, otherwise, it is assigned the next integer-name, starting with integer-name of 0 for the sentinel character. The integer-names are stored in another array such that the corresponding starting position of an LMS and its integer-name are stored at the same index of the corresponding arrays.

Finally, to build a shortened string \(T_1\) whose characters are integer-names of LMS substrings of \(T\), we use another \(quick sort\) that sorts the integer-names of all distinct LMS substrings according to the increasing order of their corresponding starting positions in \(T\). To make it clear, we use \(quick sort\) to sort pairs \(<\text{position}, \text{integer-name}>\) in increasing order of positions. This places integer-names

---

**ALGORITHM 3:** Building a shortened \(T_1\)

0: **Input:** string \(T\), integer arrays \(positionsDistinctLMS\) and \(namesDistinctLMS\), integer array \(hashTable\)

**Output:** shortened string \(T_1\) of size equal to the total number of LMS substrings in \(T\)

1: Scan \(T\) Right-to-Left:

- keep pointer \(p\) to point to the last position in \(positionsDistinctLMS\);
- fill in \(T_1\) Right-to-Left using index \(j\)

2: If \(index\ \(i\) is the starting position of LMS substring \(s\):

3: \(\text{if } i = \text{positionsDistinctLMS}[p] ;\)

4: \(\text{then } T_1[j] = \text{namesDistinctLMS}[p] ;\)

5: \(j--; p--; \)

6: Else if \(i \neq \text{positionsDistinctLMS}[p] \)

7: \(\text{then } T_1[j] = \text{hashTable}[-s];\)

8: return \(T_1\)

---

Figure 4: Building a shortened \(T_1\)
into appropriate slots within $T_1$. Figure 4 shows the procedure that builds $T_1$.

We fill in $T_1$ Right-to-Left while scanning $T$ Right-to-Left. We maintain the pointer $p$ that points to the currently processed position in the array $positionsDistinctLMS$ (processed Right-to-Left). If the currently processed position $i$ in $T$ corresponds to the starting position of LMS substring $s$, we compare $i$ with the current position stored at $positionsDistinctLMS[p]$. If these positions are equal, then this means that the currently processed LMS substring $s$ is either long or the representative of identical LMS substrings. This also means that in case $s$ is a short LMS, then no other LMS the same as $s$ has been processed yet. If $s$ is a short LMS, we retrieve the integer-name of $s$ stored at $namesDistinctLMS[p]$ and place this name in the corresponding entry of the hash table using hash index equal to $\neg s$ (the complement of $s$). Either $s$ is long or short, we fill in the current slot of $T_1$ with its integer-name stored in the hash table at index $\neg s$ and place this name into the current slot of $T_1$. This procedure takes linear time.

4 Experimental Result

The major motivation for our algorithm was to time-efficiently construct an FM-index together with BWT for a large genome in practical space (less than 16GB of RAM). Taken this into account, to benchmark the performance of our algorithm, we chose the existing tools that build an FM-index or BWT for large genomes in practical space. We chose BWT-IS algorithm (implemented by the authors) because it can calculate the BWT for a human-size genome. We wrote our own script that given BWT-IS’s output BWT, calculates an FM-index. The other chosen tool was Bowtie-2, [8] that builds an FM-index using algorithm by Kärkkäinen [6]. Bowtie-2 can build FM-index for small size genomes (less than or equal to 4GB) and large genomes (greater than 4GB). Finally, we show the results of BRAT-nova [4] that builds an FM-index exclusively for mapping bisulfite-treated reads to identify methylation, an important epigenetic marker.

We intentionally did not choose any of SA-IS algorithms because they run in space greater than 16GB and because publicly available implementations of SA-IS have limitation to work with strings less than $2^{31}$. We used BWT-IS algorithm that was kindly provided by the authors, and we used versions bowtie2-2.3.2 (for a human genome) and bowtie-2.2.5 (for two strands of a human genome). All programs were compiled using the provided Makefiles. No additional options were used with Bowtie2.

We used total of three data sets. The first data set was the human genome GRCh38, from which we removed long runs of consecutive N characters leaving at most 49 of consecutive Ns. The size of this resulting genome was 2,934,896,319. The second data set was concatenation of two strands (positive and negative) of the same human genome. We took the reverse of the negative strand concatenated with the reverse of the positive strand. The reason for this choice of concatenation was the way the DNA reads are mapped to the FM-index: this way allows mapping reads starting with the starts of reads that have the least number of sequencing errors. Finally, we used two strands of GRCh38 just as in the second step, but with all Cs converted to Ts. This index is used to map bisulfite-treated reads to identify methylation within a genome; in particular, this index is used in BRAT-nova, a mapping tool for bisulfite-treated reads. Hereafter, we will call these three data sets as $hg$-$one$-$strand$, $hg$-$two$-$strands$ and $hg$-$two$-$strands$-$bs$ respectively. Table 1 shows the running time of the tools on all three data sets measured in seconds and RAM usage. For BWT-IS, the running time is the sum of the time needed to run BWT-IS and the time to build the FM-index from the resulted BWT.

Bowtie-2 builds an FM-index for forward strand and reverse strands separately (first, it builds the FM-index for forward strand and then for the reverse strand). To make comparison fair, in Table 2 for Bowtie-2 we show time as reported by the program required to build the FM-index for forward strand only. For BWT-IS and our algorithm BWT-ISFM, we report the time as reported by the Linux command /usr/bin/time -v by summing up user and system times, and for Bowtie-2 as reported by the tool. BWT-IS does not support strings of length greater than $2^{32}$-1, so there are no results for data sets that use two strands of a human genome. To make our report complete, here we report time spent by BWT-IS to build BWT for $hg$-$one$-$strand$: it took 728sec. The rest 1536sec is required to build

<table>
<thead>
<tr>
<th>Tool</th>
<th>$hg$-$one$-$strand$</th>
<th>$hg$-$two$-$strands$</th>
<th>$hg$-$two$-$strands$-$bs$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>RAM</td>
<td>Time</td>
<td>RAM</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------</td>
<td>----------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>BWT-IS + FM-index</td>
<td>2264sec</td>
<td>5.45GB 1.86n</td>
<td>-</td>
</tr>
<tr>
<td>Bowtie2</td>
<td>3190sec</td>
<td>5.58GB 1.90n</td>
<td>7154sec</td>
</tr>
<tr>
<td>BRAT-nova</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BWT-ISFM</td>
<td>1397sec</td>
<td>6.15GB 2.10n</td>
<td>2736sec</td>
</tr>
</tbody>
</table>
an FM-index. Our program BWT-ISFM shows the best time on hg-one-strand among the three tools compared. BWT-ISFM is 2.3, 2.6 and 3.3 times faster than Bowtie-2 on these three data sets; and it is 15.9 times faster than BRAT-nova. It shows comparable results with BWT-IS, but in addition our tool supports larger input strings.

All experiments were run on a 6-core Intel Xeon Processor 2.8GHz, 198GB RAM, and 216TB of raw storage space running Linux Ubuntu. RAM usage was measured as the maximum resident size. For BWT-IS we report memory usage while running BWT-IS. Space is reported in GB and as a function relative to the genome length $n$. For example, on hg-one-strand, BWT-IS used total of 5.45GB, which is 1.86$n$ of bytes.

On hg-one-strand, our program uses slightly more memory than BWT-IS and Bowtie-2. On two strands of a human genome, BWT-ISFM shows better results than Bowtie-2 in terms of RAM. Compared to BRAT-nova, BWT-ISFM uses 1.5 times more space, but is 15.9 times faster. Overall, our program demonstrates a good practical tradeoff between space and time performance.

The source code for BWT-ISFM together with the scripts used in this benchmarking as well as the User Manual can be found at: https://drive.google.com/drive/folders/0Bx79W9h8ZBHeZTlZRzFPSVVaRjQ

5 Conclusion and Future Improvements

In our work we extended BWT-IS algorithm to build an FM-index on fly while constructing BWT. We tuned our implementation to achieve a good balance between the running time and RAM usage while running on large genomes (of size at most $2^{32} - 1$) as input. We think our algorithm can be extended to work with genomes of size up to $2^{234}$ (instead of up to $2^{33}$) as long as the total number of LMS substrings in a given genome-string $T$ fits into 32 bits. For the genome of size $2^{34} - 1$, this would mean that the total number of LMS substrings (i.e. the length of $T$) should be no more than $n/4$. In this case, our algorithm instead of using 1 bit to indicate whether a position is greater than MAXUI, will have to keep 2 most significant bits of the position.

6 Acknowledgement

The author thanks Stefano Lonardi and Tim Close (UC Riverside) for access to their computing server.

This work was supported in part by a 2017-1018 California State University Research, Scholarly, and Creative Activities Grant.

References