A Fast Suffix-Sorting Algorithm

R. Ahlswede, B. Balkenhol, C. Deppe, and M. Fröhlich

1 Introduction

We present an algorithm to sort all suffixes of $x^n = (x_1, \dots, x_n) \in \mathcal{X}^n$ lexicographically, where $\mathcal{X} = \{0, \dots, q-1\}$. Fast and efficient sorting of a large amount of data according to its suffix structure (suffix-sorting) is a useful technology in many fields of application, front-most in the field of Data Compression where it is used e.g. for the Burrows and Wheeler Transformation (BWT for short), a block-sorting transformation ([3],[9]).

Larsson [4] describes the relationship between the BWT on one hand and suffix trees and context trees on the other hand. Then Sadakane [8] suggests a well referenced method to compute the BWT more time efficiently. Then the algorithms based on suffix trees have been improved ([6],[5],[1]).

In [3] it was observed that for an input string of size n, this transformation can be computed in $O(n)$ time and space[1] using suffix trees. While suffix trees are considered to be greedy in space – even small factors hidden in the O-notation may decide on the feasibility of an algorithm – sorting was accomplished by alternative non-linear methods: Manber and Myers [7] introduced an algorithm of $O(n \log n)$ in worst case time and $8n$ bytes of space and in [2] an algorithm based on *Quicksort* is suggested, which is fast on the average but its worst case complexity is $O(n^2 \log n)$. Most prominent in this case is the Bendson-Sedgewick Algorithm which requires $4n$ bytes and Sadakane's example of a combination of the Manber-Myers Algorithm with the Bendson-Sedgewick Algorithm with a complexity of $O(nlogn)$ worst case time using $9n$ bytes [8].

The reduction of the space requirement due to an upper bound on n seems trivial. However, it turns out that it involves a considerable amount of engineering work to achieve an improvement, while retaining an acceptable worst case time complexity. This paper proposes an algorithm, efficient in the terms described above, ideal for handling large blocks of input data. We assume that the cardinality of the alphabet (q) is smaller than the text-string (n). Our algorithm computes the suffix sorting in $O(n)$ space and $O(n^2 \log n)$ time in the worst case. It has also the property that it sorts the suffixes lexicographically according to the prefixes of length $t_2 = \log_q \lceil \frac{n}{2} \rceil$ in the worst case in linear time. After the initial sorting of length t_2, we use a Quick-sort-variant to sort the remaining part. Therefore we get the worst time $O(n^2 \log n)$. It is also possible to modify our algorithm by using Heap-sort. Then we will get a worst case time $O(n(\log n)^2)$.

[1] This only holds, if the space complexity of a counter or pointer is considered to be constant (e.g. 4 Bytes) and the basic operations on them (increment, comparison) are constant in time. This assumption is common in the literature and helpful for practical purposes.

We use Quick-sort, because it is better in practice and has an average time of $O(n \log n)$ like Heap-sort, but with a smaller factor.

The elements of $\mathcal{X}$ are called *symbols*. We denote the symbols by their *rank* w.r.t. the order on $\mathcal{X}$. We assume that $\$ = q-1 \in \mathcal{X}$ is a symbol not occurring in the first $n-1$ symbols in x^n, the *sentinel* symbol.

x_i is the ith element in the sequence x^n. If $i \leq j$, then $(x_i, \ldots, x_j)$ is the factor of x^n beginning with the ith element and ending with the jth element. If $i > j$, then $(x_i, \ldots, x_j)$ is the empty sequence. A factor v of x begins at position i and ends at position j in x if $(x_i, \ldots, x_j) = v$. To conveniently refer to the factors of a sequence, we use the abbreviation x_i^j for $(x_i, \ldots, x_j)$.

2 The Initial Sorting Step

Before we tackle the problem of sorting all suffixes of a given sequence in lexicographical order we start to consider the case where we only sort the suffixes looking at the prefixes of a fixed length correctly. The simplest case is to look at all prefixes of length one, which is the case to sort all symbols occurring in the input sequence lexicographically.

2.1 Sorting of the Symbols

The sorting of the symbols of a given input sequence x^n with symbols out of a finite alphabet $\mathcal{X}$ can be done linearly in time and space complexity as follows:

We define q counters $(counter_0[0], \ldots, counter_0[q-1])$ and count for each symbol in $\{0, \ldots, q-1\}$ how often it occurs in x^n. In each step i we have to increase exactly one counter $(counter_0[x_i])$ by one. Therefore to get the frequencies of the symbols requires $O(n)$ operations. Now our alphabet is given in lexicographic order and we generate the output in the following way: First output $counter_0[0]$ many zeros, followed by $counter_0[1]$ many ones,... Obviously the generated output sequence is produced in $O(n)$ operations and the sorting is done.

2.2 Sorting a Given Prefix Length

We would like to continue the sorting of all suffixes in an iterative way by using the counting idea of the previous section. In a later step of the algorithm we need n counters. We have to take the memory already at the beginning, which allows us to use it already in the initial sorting phase. We choose t_1 such that $2^{t_1-1} < q \leq 2^{t_1}$ and t_2 such that $2^{t_1 t_2} \leq \lfloor \frac{n}{2} \rfloor < 2^{t_1(t_2+1)}$. For simplicity we assume from now on that $q = 2^{t_1}$ and $n = 2^{t_1 t_2 + 1}$.

We like to sort all suffixes such that the first t_2 symbols of each suffix are sorted lexicographically correctly.

Now we will count the number of occurrences of factors of length t_2 in our sequence x^n. We assume that $x_{n+1}, \ldots, x_{n+t_2-1} = q-1$ and count the factors as follows. The counter$[a_1 k^{t_2-1} + a_1 k^{t_2-2} + \cdots + a_{t_2} k^0]$ counts the number of occurrences of the factor $(a_1, \ldots, a_{t_2})$. Let us define a temporary value $tmp = 2^{t_1 t_2} - 1$ and $i = n$. This is the position n of the sequence, with the factor

$(q-1,\dots,q-1)$. Now starting at the end down to the beginning of the input sequence x^{n+t_2-1} in each step we increase *counter*[*tmp*] by one, decrease i by one and we calculate:

$$tmp \to \left\lfloor \frac{tmp}{2^{t_1}} \right\rfloor + x_i 2^{t_1(t_2-1)}.$$

Notice that multiplications and divisions by powers of two can be represented by **shifts**. Let us denote

$$a >> b = \left\lfloor \frac{a}{2^b} \right\rfloor \text{ and } a << b = a2^b.$$

Furthermore notice that the + operation can be replaced by a binary logical or-operation which we denote as $|$. Hence in total we need $O(n)$ operations.

By construction *tmp* will only take values less than $\lfloor \frac{n}{2} \rfloor = (n >> 1)$, such that we can calculate the partial sums of the entries *counter*[j] and store them in the second half of the memory for the array *counter*

$$counter[\frac{n}{2}+j] \to \sum_{i=0}^{\frac{n}{2}-1} counter[i].$$

Obviously this calculation can also be done linearly in time:

```
i->1
counter[(n>>1)]->0
while i< (n>>1) do
   counter[(n>>1)+i] -> counter[(n>>1)+i-1] + counter[i-1]
   i-> i+1
done
```

Finally we have to write back the result of the sorting. In order to continue we introduce two further arrays of size n, one, which we denote as *pointer*, in order to describe the starting points of the suffixes, and the second one, denoted as *index*, to store the partial results of the sorting.

Again we start with $tmp = 2^{t_1 t_2} - 1$ and at position $i = n$.

```
while i>n-t_2 do
   i->i-1
   tmp->(tmp>>t_1)|x_i<<t_1(t_2-1)
   counter[tmp]->counter[tmp]-1
   index[i]->counter[tmp+(n>>1)]+counter[tmp];
   pointer[index[i]]->i;
done
while i>0 do
   i->i-1
   tmp->(tmp>>t_1)|x_i<<t_1(t_2-1)
   counter[tmp]->counter[tmp]-1
   index[i]->counter[tmp+(n>>1)]
   pointer[index[i]+counter[tmp]]->i
done
```

In the first loop we consider the cases where we have to take the sentinel into account (we assume that $x_{n+i} = \$$). With the starting definition of *tmp* the sentinel will be taken as a number greater or equal to $|\mathcal{X}| - 1$. Using the fact that it occurs only at the end of the sequence, that is with the largest entry of *count*, we can fix the position of the last t_2 entries to the starting-point of the prefix of suffixes, represented as integer *tmp* at that moment, plus the number of occurrences of that value *tmp*. In all other cases (second loop) we set $index[i]$ to the starting position of the interval of suffixes with prefix *tmp*.

In other words after these loops $pointer[1], \dots, pointer[n]$ represent the starting positions of the suffixes in lexicographical order according to the prefixes of length t_2. If $index[pointer[i]] < index[pointer[j]]$ ($index[pointer[i]] > index[pointer[j]]$) then the suffix starting in $pointer[i]$ is lexicographically smaller (larger) than the suffix starting in position $pointer[j]$. If the two values are equal, then the two suffixes have a common prefix of length greater or equal to t_2.

Notice that to finish the lexicographic order in total we can continue using the two arrays *pointer* and *index* only, that is there is no need to look at the original input sequence to calculate the defined total order, such that the continuation is independent of the alphabet size.

3 Only Three Elements

In order to continue the sorting we first analyze how to sort and how to calculate the median of three given numbers.

3.1 Median-Position-Search of Three Elements

The median m of a triple $(n_1, n_2, n_3) \in \mathbb{N}_0^3$ is a value equal to at least one of them which is in between the two others, i.e.

$$m = n_1 \Rightarrow n_2 \le n_1 \le n_3 \text{ or } n_3 \le n_1 \le n_2,$$

$$m = n_2 \Rightarrow n_1 \le n_2 \le n_3 \text{ or } n_3 \le n_2 \le n_1,$$

$$m = n_3 \Rightarrow n_2 \le n_3 \le n_1 \text{ or } n_1 \le n_3 \le n_2.$$

Notice that we are not interested in the value itself, only in the position relative to the two others, i.e. for us there is no difference between the case $(1, 1, 1)$ and $(2, 2, 2)$. Therefore we partition the set of triples in the following way. We define 13 subsets $\mathcal{A}_1, \dots, \mathcal{A}_{13} \subset \mathbb{N}_0^3$ in the following way: For $k \in \mathbb{N}_0$ and $l, m \in \mathbb{N}$ we define

$$\mathcal{A}_1 = \{(k, k, k)\} \quad \mathcal{A}_8 = \{(k, k+l, k+l+m)\}$$

$$\mathcal{A}_2 = \{(k, k, k+l)\} \quad \mathcal{A}_9 = \{(k, k+l+m, k+l)\}$$

$$\mathcal{A}_3 = \{(k, k+l, k)\} \quad \mathcal{A}_{10} = \{(k+l, k, k+l+m)\}$$

$$\mathcal{A}_4 = \{(k+l, k, k)\} \quad \mathcal{A}_{11} = \{(k+l, k+l+m, k)\}$$

$$\mathcal{A}_5 = \{(k, k+l, k+l)\} \quad \mathcal{A}_{12} = \{(k+l+m, k, k+l)\}$$
$$\mathcal{A}_6 = \{(k+l, k, k+l)\} \quad \mathcal{A}_{13} = \{(k+l+m, k+l, k)\}$$
$$\mathcal{A}_7 = \{(k+l, k+l, k)\}.$$

For a given triple (n_1, n_2, n_3) the median is known to us, if we know the index i with $(n_1, n_2, n_3) \in \mathcal{A}_i$. Therefore we define the following questionnaire of yes–no–questions where a question is of the following form: $a \leq b, a < b, a = b$.

```
if n_1 <= n_2 then
   if n_2 <= n_3 then m=n_2
   else if n_1 <= n_3 then m=n_3
        else m=n_1
        endif
   endif
else
   if n_3 <= n_2 then m=n_2
   else if n_1 <= n_3 then m=n_1
        else m=n_3
        endif
   endif
endif
```

Notice that we need at most three yes-no-questions and we need only two in case where the median is already in the middle.

3.2 Sorting of Three Elements

Using questions of the form mentioned in the previous section we can sort three elements using at most four questions:

```
if n_1 <= n_2 then
   if n_2 <= n_3 then
      if n_1 = n_2 then
         if n_2 = n_3 then (n_1,n_2,n_3) in A_1
         else (n_1,n_2,n_3) in A_2
         endif
      else
         if n_2 = n_3 then (n_1,n_2,n_3) in A_5
         else (n_1,n_2,n_3) in A_8
         endif
      endif
   else
      if n_1 <= n_3 then
         if n_1 = n_3 then (n_1,n_2,n_3) in A_3
         else (n_1,n_2,n_3) in A_9
         endif
      else
         if n_1 = n_2 then (n_1,n_2,n_3) in A_7
         else (n_1,n_2,n_3) in A_11
         endif
      endif
   endif
else
   if n_1 > n_3 then
      if n_2 = n_3 then (n_1,n_2,n_3) in A_4
      else if n_2 < n_3 then  (n_1,n_2,n_3) in A_12
           else  (n_1,n_2,n_3) in A_13
           endif
      endif
   else
      if n_1 = n_3 then  (n_1,n_2,n_3) in A_6
      else  (n_1,n_2,n_3) in A_10
      endif
   endif
endif
```

4 The Main Loop of the Sorting Algorithm

After the initial sorting phase we have the array *pointer*, which points to the starting positions of the suffixes lexicographically correctly sorted according to the prefixes of length t_2. *index* contains the partial ordering, that is if the values are different, then the larger one is lexicographically larger than the smaller one, if they are equal then the two suffixes have a common prefix of length greater or equal to t_2. Finally we can calculate with the second half of the array *counter* the positions of the intervals with common prefixes of length t_2. We use now *counter*[0] to count the number of intervals where we have to continue with the sorting, more precisely *counter*[0] points to the first free place in memory where we can store a further interval, which is in the beginning 1 (*counter*[1] is free).

```
  counter[0]->1
```

Starting the loop to get the not necessarily correctly sorted intervals *counter*[0] is initialized with 1 because we need it in this way later and we are working on "unsigned int".

```
  i->0
  while i< 2^(t_1*t_2) do
    if counter[(n>>1)+i+1]-counter[(n>>1)+i]>1 then
       counter[counter[0]+1]->counter[(n>>1)+i];
       counter[counter[0]+2]->counter[(n>>1)+i+1]-1
       while index[pointer[counter[counter[0]+2]]]>
             index[pointer[counter[counter[0]+1]]] do
             counter[counter[0]+2]->counter[counter[0]+2]-1
```

```
        done
        if counter[counter[0]+2]!=counter[counter[0]+1] then
            counter[0]->counter[0]+2
        endif
    endif
done
```

Notice that during the loop we reuse the memory in *counter* from $(n >> 1)$ to n.

4.1 Split an Interval

We have to sort an interval from position *begin* to *end* that is $pointer[begin]$ to $pointer[end]$ has to be sorted but they are already of equal length *length*. We like to do the sorting by a 3 part quick-sort. The array 'smaller' contains all pointers which are smaller than the first entry (smaller defined by *index* !), the array 'equal' the pointers which are equal in the first $2length$ positions with the first one and the array 'bigger' the remaining ones. After we have split this part we have to continue with 'smaller' and 'bigger' of length *length* and with 'equal' of length $2 \cdot length$. These intervals (starting point, end point) we return to the calling function using two arrays x and y.

Given a value *val*, the index for the interval stored in *counter* at positions $counter[val - 1]$ and $counter[val]$, the value *length* which is the length of the common prefix already known from the previous steps (after the initial sorting it is t_2) and a flag *flag* which describes whether the intervals are stored at the beginning of counter or at the end (after the initial part at the beginning).

Now the beginning of the interval is given by $begin = counter[val - 1]$ and the end position by $end = counter[val]$. Notice that the last *length* pointers of the original sequence can not occur inside this interval because they are correctly inserted in one of the previous steps due to the (virtual) sentinel symbol at the end of the input sequence. Therefore if we look at the suffixes starting at $pointer[begin]$ and $pointer[end]$, then we know they have by construction a common prefix of length at least *length*. But if we look at the two suffixes without the prefix of length *length*, then theses two suffixes have been sorted correctly also according to the prefix of length *length*. In other words the result of the comparison of the two pointers $pointer[begin]$ and $pointer[end]$ is equal to the result of the comparison of $pointer[begin]+length$ and $pointer[end]+length$. We can get the result by using the values stored in the array *index*. Let us denote that a is lexicographic smaller than b with $a \prec b$ for two pointers a, b where a pointer is smaller than another one if the corresponding suffix starting at that pointer is lexicographic smaller than the other suffix. Then

$$pointer[begin] \prec pointer[end] \Leftrightarrow$$

$$pointer[begin] + length \prec pointer[end] + length.$$

Therefore if now $index[pointer[begin] + length] = index[pointer[end] + length]$ then the suffixes starting at $pointer[begin]$ and $pointer[end]$ have a common prefix of length at least $2 \cdot \mathit{length}$. Otherwise we can use the result to get the right comparison result. Notice that in this way we double the length of the comparison in each step.

Now for a given interval we like to split the interval into several parts similar to quick-sort. Therefore we take three values and calculate the median as mentioned in Section 3.1

```
n_1->index[pointer[begin]+length];
n_2->index[pointer[(begin+end)>>1]+length];
n_3->index[pointer[end]+length];

median-> (n_1 <= n_2 ?
            (n_2 <= n_3 ? n_2 : (n_1 <= n_3 ? n_3 : n_1 ) )
            : (n_3 <= n_2 ? n_2 : (n_1 <= n_3 ? n_1 : n_3 )))
```

With $\mathit{currentindex} = \mathit{index}[\mathit{pointer}[\mathit{begin}]]$ we have the value of $\mathit{index}[\mathit{pointer}[i]]$ for all $\mathit{begin} \leq i \leq \mathit{end}$. Now we like to split the interval into three parts, one for the pointers which are smaller than the *median* one for those which are equal and one for those which are larger. We divide the parts by changing the values of the pointers as follows:

First we need two further variables which we set to *begin* and *end* respectively.

```
s->begin
b->end
```

And we need yet another variable k for the actual position inside the interval. As long as the values of $\mathit{index}[\mathit{pointer}[k] + \mathit{length}] < \mathit{median}$ and $k \leq b$ the current end of the interval we increase k by one:

```
k->begin;                /* the starting point */
while index[pointer[k]+length]<median && k<=b do
   k->k+1
done
s->k;
```

We set s to the actual value of k such that s points to the first position which is greater or equal to the *median*. In a similar way we reduce b at the end, give the first pointer which is less or equal than the *median*.

```
while index[pointer[b]+length]>median && k<=b do
  b->b-1
done
```

Remember that we have stopped the first loop in a case where

$$\mathit{index}[\mathit{pointer}[k] + \mathit{length}] \geq \mathit{median}$$

and the second one where

$$\mathit{index}[\mathit{pointer}[b] + \mathit{length}] \leq \mathit{median}.$$

Now let us continue in the following way:

```
if index[pointer[k]+length]>median then
  SWAPPOINTER(k,b)
  b->b-1
```

where we denote with $SWAPPOINTER(k, b)$ the following operations:

$$tmp- > pointer[k] \quad pointer[k]- > pointer[b] \quad pointer[b]- > tmp$$

such that the two values are simply exchanged. Now we have that $index[pointer[k] + length] \leq median$ and we continue:

```
  if index[pointer[k]+length]=median then
    k->k+1
    while index[pointer[k]+length]=median do
      k->k+1
    done
  else
    k->k+1 s->s+1
  endif
else
  k->k+1
  while index[pointer[k]+length]=median && k<=b do
     k->k+1
  done
endif
```

Now if $s > begin$ then the part from $begin$ to $s - 1$ stores the pointers which are smaller than the $median$ and if $b < end$ then the part from $b + 1$ to end are the pointers which are larger than the $median$. Furthermore if $s < k$ then the part from s to $k - 1$ are pointers which are equal to the $median$. Let us first continue with the case where $s = k$:

```
if s=k then
  s->end+1     /* we make the value impossible, in other   */
               /* words larger then end                     */
  while k<=b && s>end do
    if index[pointer[k]+length]<median then
      k->k+1   /* one further pointer which is smaller     */
    else
      if index[pointer[k]+length]>median then
        SWAPPOINTER(k,b);
        b->b-1 /* add to bigger interval                    */
      else
        s->k   /* s is getting a value <= end and          */
               /* the loop stops.                           */
        k->k+1 /* they are equal                            */
      endif
    endif
  done
endif
```

Now we have found at least one pointer which is equal to the median. We have to continue similarly as before but if $index[pointer[k] + length] < median$ then we have to exchange in addition the pointers in positions k and s and we have to increase also s. Furthermore the only stop situation for the loop occurs if $k > b$.

```
while k<=b do
  if index[pointer[k]+length]<median then
    SWAPPOINTER(k,s);
    k->k+1
    s->s+1
  else
    if index[pointer[k]+length]>median then
      SWAPPOINTER(k,b);
      b->b-1 /* add to bigger */
    else
      k->k+1 /* they are equal */
    endif
  endif
done
```

Now we have the three parts

$$begin, \ldots, s-1, \text{ the pointers which are smaller}$$

$$s, \ldots, b, \text{ the pointers which are equal}$$

$$b+1, \ldots, end, \text{ the pointers which are larger.}$$

If $s - 1 < begin$ or $b + 1 > end$ then the corresponding intervals are empty. In order to use these parts in the future, we have to update the values of index for the current pointers. Notice that equal to the *median* means that they have a common prefix of a length at least $2 \cdot length$.

For the first interval (if it exists) nothing has to be done, because the values of *index* are already at the starting point of the interval. The new starting point of the second part is

```
currentindex->currentindex+s-begin
```

Of course the second part contains at least one pointer by construction (the pointer which is used to calculate the median has a common prefix to itself !).

```
if s>begin && s<=b then
  k->s
  while k<=b do
```

```
      index[pointer[k]]->currentindex;
      k->k+1
    done
  endif
```

Finally we have to calculate the starting point of the last interval (if it exists)

```
  currentindex->currentindex+b+1-s;

  if b+1<=end then
    k->b+1
    while k<=end do
      index[pointer[k]]->currentindex
      k->k+1
    done
  endif
```

Now we have to continue with our sorting algorithm on the constructed intervals. But before we start to consider the interval from s to b of length $2 \cdot \mathit{length}$ we like to finish all intervals of length *length* in order to double the compared lengths of the prefixes again. For that reason we store that interval at the opposite end of the array *counter* on which we are working at the moment. After the initial part we are working at the beginning to store our intervals, such that we store the interval from s to b at the end. After we have finished all intervals which we have to compare of length *length* we start to work at the end with the intervals sorted correctly of length $2 \cdot \mathit{length}$ and store all intervals we produce of length $4 \cdot \mathit{length}$ at the beginning. Notice that the total number of intervals we have to store is always less than n such that if we need more space at the end it is free at the beginning of the array *counter* and vice versa. To add these intervals we define a function $\mathit{INSTOCOUNTER}(\mathit{FROM}, \mathit{TO}, \mathit{FLAG})$ where FROM,TO are the boundaries of the interval which we have to add and FLAG describes where. If we are working at the end of *counter* we use $\mathit{counter}[n]$ similarly to $\mathit{counter}[0]$ for the beginning part. To delete one interval at the end we have to increase $\mathit{counter}[n]$ such that we need two different rules to add an interval at the end:

```
INSTOCOUNTER(FROM,TO,FLAG) {
   switch(FLAG) {
    case 0: {
      counter[counter[0]]->(FROM);
      counter[0]->counter[0]+1
      counter[counter[0]]->(TO);
      counter[0]->counter[0]+1
      break;
    }
    case 1: {
      counter[n]->counter[n]-1
      counter[counter[n]]_>(TO);
      counter[n]->counter[n]-1
      counter[counter[n]]->(FROM);
      break;
    }
   default: { /* case 2 */
      counter[counter[n]]->(FROM);
      counter[counter[n]+1]->(TO);
      counter[n]->counter[n]-2;
      break;
    }
   }
}
```

Now the insertion of the intervals using the function $INSTOCOUNTER$ can be done as follows:

```
  if s-begin>1 then
    INSTOCOUNTER(begin,s-1,2-(flag<<1))
  endif
  if b-s>0 then
    INSTOCOUNTER(s,b,flag)
  endif
  if end-b>1 then
    INSTOCOUNTER(b+1,end,2-(flag<<1))
  endif
```

4.2 Calling the Sorting Procedure

To conclude the description of the whole algorithm it remains to describe the step between the initial sorting phase and the calling of the procedure to split a given interval.

We are starting in a situation where we have given the three arrays *counter*, *pointer* and *index* and we know, that if we use the values stored in *index* as rule for the comparison of two pointers then the result is correct according to the first t_2 symbols (from the initial sorting part).

As mentioned earlier we like to use the array *counter* from both sides. At the beginning we use a variable *length* which describes the length of the common prefix correctly sorted. This variable is initialized with t_2 from the initial sorting phase. In order to double the length in each loop we have to use the information stored in *index* to sort all suffixes according to the first $2 \cdot length$ symbols correctly. After that we use the information to double the length again and so on. $counter[0]$ is already used to describe the first free position in memory at the beginning of *counter*. Analogously we use $counter[n]$ in order to do the same procedure at the end. Therefore we have to store at the same time intervals sorted with prefixes of length *length* and of length $2 \cdot length$. If there is no further one of length *length* we start to sort them of length $2 \cdot length$ and produce new ones of length $4 \cdot length$. Notice that the total number of intervals can not be more

then $n >> 1$ such that to store them with starting and ending point we need at most n values in the memory. Furthermore out of the initial sorting part some pointers at the end of the input sequence (exactly t_2 many) are already correctly sorted such that the memory requirement is strictly less than $n - 2$ (we make an initial sorting at least of length 2). For typical files we need only something like $n >> 2$ entries in memory, but in worst case $n - t_2$ is needed as we can see by the following example:
Take a deBruijn sequence of length 2^{n-1} copy the sequence and concatenate the two. The property of a deBruijn sequence is, that if we are looking at a linear shift-register of length $n-1$ then these sequences have maximal period, or more precisely every binary sequence of length $n-1$ occurs exactly once. Now if we have a length of $t_2 = n-1$ then each prefix occurs in the constructed sequence exactly twice and hence we have n intervals from which only $n-1$ are getting correctly sorted at the initial phase.

Now at the beginning we have no interval to sort of length $2 \cdot$ *length*:

```
counter[n]->n;
```

We are starting the main loop.

```
  /* as long as there is something to compare              */
while(counter[0]>1) do
  /* starting with the beginning part (at the end)         */
```

We call this loop twice because first we like to sort every interval of length *length* correctly, after that we continue at the end of *counter* and sort the intervals of length $2 \cdot$ *length*. If there are further intervals of length $4 \cdot$ *length* then we can find them at the beginning of *counter*.

```
/* as long as we have something to compare of length "length" */
    while counter[0]>1 do
      counter[0]->counter[0]-2

      switch(counter[counter[0]+1]-counter[counter[0]]) {
                 /* +1 is the number of elements !          */
```

Notice that using the procedure of Section 4.1 the calculation of the median is only efficient if we have enough elements to sort. Therefore in case where we have intervals of a small length we sort directly:

With only two entries we need in the worst case two questions in order to sort them

```
      case 1: {  /* only two entries                        */
        m1->index[pointer[counter[counter[0]]]+k]
                 /* a shortcut to store them in order not   */
        m2->index[pointer[counter[counter[0]+1]]+k]
                 /* to calculate them twice                 */
        if m1=m2 then
```

The two values are equal, that means the two suffixes are equal of length "2*lengthänd therefore we add it at the end of counter.

```
          INSTOCOUNTER(counter[counter[0]],
                       counter[counter[0]+1],1)
        else
```

They are different so that we can compare them

```
          if m1<m2 then
```

The beginning value of the interval is smaller than the end, therefore we do not have to exchange the order and we can update the index.

```
            index[pointer[counter[counter[0]+1]]]->
                      index[pointer[counter[counter[0]+1]]]+1;
          else
```

We have to swap them and to update the index of the beginning pointer.

```
            SWAPPOINTER(counter[counter[0]],
                        counter[counter[0]+1])
            index[counter[counter[0]]]->
                        index[counter[counter[0]]]+1
          endif
        endif
        break;
      }  /* end of case interval of length 2                */
```

An interval with three elements we can sort as described in Section 3.2. We call a function *sort*3 which needs as parameters the array *counter*, the position (*counter*[*counter*[0]]) in counter to get the boundaries for the interval to sort, the arrays *pointer* and *index*, a *flag* which describes how to insert a new interval to continue with, the *length* of the already compared prefixes and finally the length n (necessary to insert a new interval using the function *INSTOCOUNTER*).

```
      case 2: { /* interval of length 3                     */
        sort3(counter,counter[counter[0]],pointer,
                      index,1,length,n)
```

Either everything is sorted or we are getting an interval back which starts with the same first $2 \cdot length$ symbols and that is we have to add them to the end of counter.

```
        break;
      } /* end of interval of length 3                      */
```

In all other cases we call the function described in Section 4.1 which we denote as *splitcount*.

```
    default: { /* the general case                              */
          splitcount(counter,counter[0]+1,pointer,index,
                            1,length,n);
      break;
    } /* end of the general case                                */
    } /* end of the switch                                      */
```

Now we can stop the loop for sorting intervals with length *length* and look at the intervals of length $2 \cdot length$.

```
  done /* inner loop: counter[0]>1                                */
  length->(length<<1)
```

The length llengthïs finished, that is we can continue with "2*lengthïn order not to copy the end to the beginning and continue the main loop we repeat the whole procedure with exchanging the role of the beginning of the array counter and the end of it. Of course $counter[0] = 1$, in other words at the beginning there is no interval of $4 \cdot length$ which we have to compare. Now we have to start the loop at the end:

```
    while counter[n]<n do
      switch(counter[counter[n]+1]-counter[counter[n]]) {
        /* +1 is the number of elements !                      */
      case 1: { /* only two elements                           */
        /* two shortcuts                                       */
        m1=index[pointer[counter[counter[n]]]+length];
        m2=index[pointer[counter[counter[n]+1]]+length];
        if m1=m2 then
```

The two values are equal and we have to add a new interval at the beginning of the array *counter* using the function *INSTOCOUNTER*.

```
          INSTOCOUNTER(counter[counter[n]],
                       counter[counter[n]+1],0);
        else /* we can compare them                            */
          if m1<m2 then
            index[pointer[counter[counter[n]+1]]]++;
          else
            SWAPPOINTER(counter[counter[n]],
                                    counter[counter[n]+1]);
            index[counter[counter[n]]]++;
          endif
        endif
        break;
      }  /* end of the case with only two elements.           */
```

As before we also consider a separate case with only three elements using the function *sort*3 as before but with $flag = 0$.

```
        case 2: {
          sort3(counter,counter[counter[n]],pointer,
                                          index,0,length,n);
          break;
          }  /* and of case 2.                            */
```

Again in all others cases we use the function *splitcount*.

```
        default: {
              splitcount(counter,counter[n]+1,pointer,
                                          index,0,length,n);
          break;
        }
        } /* and of the switch                            */
          /* continue with the next interval.             */
        counter[n]->counter[n]+2
      done /* end of the loop counter[n]<n                */
      length->(length<<1) /* double again and return to the  */
       /* first loop: counter[0]>1.                      */
   done
```

References

1. B. Balkenhol and S. Kurtz, Space efficient linear time computation of the Burrows and Wheeler transformation, Number, Information and Complexity, Special volume in honour of R. Ahlswede on occasion of his 60th birthday, editors I. Althöfer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sárközy, I. Wegener, and Z. Zhang, Kluwer Acad. Publ., Boston, Dordrecht, London, 375–384, 1999.
2. J. Bentley and R. Sedgewick, Fast algorithm for sorting and searching strings, Proceedings of the ACM–SIAM Symposium on Discrete Algorithms, 360–369, 1997.
3. M. Burrows and D.J. Wheeler, A block–sorting lossless data compression algorithm, Technical report, Digital Systems Research Center, 1994.
4. N.J. Larsson, The context trees of block sorting compression, Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 30 – April 1, IEEE Computer Society Press, 189–198, 1998.
5. N.J. Larsson, Structures of string matching and data compression, PhD thesis, Dept. of Computer Science, Lund University, 1999.
6. N.J. Larsson and K. Sadakane, Faster suffix–sorting, Technical Report LU–CS–TR: 99–214, LUNDFD6/(NFCS–3140)/1–20/(1999), Dept. of Computer Science, Lund University, 1999.
7. U. Manber and E.W. Myers, Suffix arrays: A new method for on–line string searches, SIAM Journal on Computing, 22, 5, 935–948, 1993.
8. K. Sadakane, A fast algorithm for making suffix arrays and for Burrows–Wheeler transformation, Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 30 – April 1, IEEE Computer Society Press, 129–138, 1998.
9. M. Schindler, A fast block–sorting algorithm for lossless data compression, Proceedings of the Conference on Data Compression, 469, 1997.