# Sorting a Compressed List

Nathan Brunelle
University of Virginia
njb2b@virginia.edu

Gabriel Robins
University of Virginia
robins@cs.virginia.edu

abhi shelat
University of Virginia
shelat@cs.virginia.edu

July 16, 2012

**Abstract**

We consider the task of sorting and performing $k$th order statistics on a list that is stored in compressed form. The most common approach to this problem is to first decompress the array (usually in linear time), and then apply standard algorithmic tools. This approach, however, ignores the rich information about the input that is implicit in the compressed form. In particular, exploiting this information from the compression may eliminate the need to decompress, and may also enable algorithmic improvements that provide substantial speedups. We thus suggest a more rigorous study of what we call compression-aware algorithms.

Already the string-matching community has applied this idea to developing surprisingly efficient pattern matching and edit distance algorithms on compressed strings. In this paper, we begin to study the problem of sorting on compressed lists. Given an LZ77 representation of size $C$ that decompresses to an array of length $n$, our algorithm can output an LZ77-compressed representation of the sorted dataset in $O(C + |\Sigma| \log |\Sigma| + n)$ time, with $\Sigma$ as the alphabet. Secondly, we consider a compression scheme in which an n-integer array is represented as the union of $C$ arithmetic sequences. Using priority queues, we can sort the array in $O(n \log C)$ time. Lastly, given an array compressed with a context free grammar of size $C$ we can find the sorted array in $O(C \cdot |\Sigma|)$, where $\Sigma$ is the alphabet of the string. Additionally we present algorithms for indexing an LZ77 compressed string in $O(C)$, and finding the $k$th order statistic in $O(C \log C)$ in the arithmetic sequences model.

# 1 Introduction

## 1.1 Motivation

As observed in the real world, data tends to exhibit various regularities, thus allowing for compression. For example, the schematic of an office building will likely feature a grid-like pattern of windows, offices, etc. Therefore, a CAD drawing of this schematic could gain much space benefit by compressing these regularities. There is a gap, however, between how classical algorithms operate, and the potential benefits attainable by exploiting these regularities. Many state-of-the art methods still require a user to decompress data, operate some method, and then potentially recompress as necessary. We seek to work towards closing this gap between the operation of classical algorithms, and the potential benefits for operating directly on compressed data.

As a result of their compression-independence, classical algorithms' complexities are often expressed in terms of their performance on some worst case example, or some average case analysis. This, however, also ignores the regularity of the data. Since the data is far from random, an average case analysis considering the data's regularity would be a more accurate representation of its real-world run time. For operations on such data, it is more useful to express run time in terms of the size of the compressed input. This measure will be a greater indicator of how well an algorithm utilizes its data regularities.

A further benefit is gained with knowledge that as the underlying complexity of the compression grows, the volume of data represented explodes faster. This means that for large inputs of compressed data, the benefit of the compression-aware algorithm will also grow as compared to the functionally-equivalent classical algorithm.

## 1.2 Problem Statement and Results

In this paper we begin with some of the most well-understood and fundamental problems to solve for any data: sorting and $k$th order statistic. We present algorithms answering each of these questions under three models of compression. The first represents a sequence in terms of a union of embedded arithmetic sequences, the second is the LZ77 compression scheme presented by Abraham Lempel and Jacob Ziv in 1977 [25], and the third represents a string as a context free grammar.

For sorting a list compressed by its arithmetic sequences we present an algorithm which uses priority queues. This runs in $O(n \log C)$ time, where $n$ is the number of points to be sorted, and $C$ is the number of sequences in the compression. This is to compared with the standard approach which would require decompression in time $O(n)$ and sorting in time $O(n \log n)$. We also present an algorithm which finds the $k$th order statistic in $O(C \log C)$ time. The classical approach would require at least $O(k)$ to decompress, and $O(k)$ to find the $k$th order statistic.

For sorting an LZ77-compressed sequence we present a sorting algorithm which can operate in $O(C \log C + n)$. Where $C$ is the compression size, and $n$ is the length of the sequence. In most instances of use, it will be the case that $C \ll n$, thus our algorithm can give dramatic time benefits over the classical sorting algorithm, as in many cases $O(n)$ will be the dominating term. This is an improvement over the $O(n \log n)$ run time of the classical approach. Additionally, at no cost to its asymptotic time complexity, the output data can be expressed in LZ77-compressed form. We also present a way of indexing into the sequence in $O(C)$ time. By combining these two, we have a method for obtaining the $k$th order statistic in $O(C + |\Sigma| \log |\Sigma| + n)$ time, meaning that in the case when $C \ll n$ we have the dominating term as $O(n)$.

For sorting a list compressed by a context-free grammar we present an algorithm which can

find the sorted sequence in $O(C \cdot |\Sigma|)$ time. Here, $C$ is the size of the compression, which in this case is the total number of symbols in all of the grammar's substitution rules, and $\Sigma$ is the set of characters in the alphabet of the string. This result has the advantage of being independent of the size of the uncompressed list. From here, we can produce a grammar for the sorted list which will be of size $|\Sigma| \log n$, where $n$ is the length of the decompressed list. The classical approach would require $O(n \log n)$ time to decompress and then sort.

## 2 Related Work

There has already been much research done on compression aware algorithms, and it is well-known that there is frequently a time and space benefit by doing so. One of the most well-explored areas is pattern matching on compressed strings, including both exact pattern matching [2, 15, 16, 19, 23], and approximate pattern matching [1, 4, 7, 14, 15, 18, 20]. Others have studied compression-aware edit-distance algorithms [3, 8, 11, 6, 17]. There have also been algorithms presented which act directly on JPEG-compressed images [24, 9].

In this paper we focus on sorting under three different compression schemes. The first is one in which a sequence of colinear points is represented by a union of embedded arithmetic sequences. Algorithms for detecting these hidden arithmetic sequences have been presented in [12, 13, 21, 10]. The second that we cover is Lempel-Ziv '77 (called LZ77 throughout), which was first presented in [25]. The final compression scheme considered is one in which a string is represented as a context free grammar, as presented in [19].

## 3 Arithmetic Sequences Compression

Under this model of compression, a set of natural numbers $S \subseteq \mathbb{N}$ is represented by some set of underlying arithmetic sequences. That is, we say that $S = A_1 \cup A_2 \cup ... \cup A_C$ for some number of arithmetic sequences $C$. Here, we will assume for the sake of simplicity that all arithmetic sequences start at 0, and subsequent values in arithmetic sequence $A_i$ are all $\delta_i$ apart. So, for example, if $C = 2$ we may have arithmetic sequences $A_1 = \{0, 5, 10, 15, 20, ...\}$ and $A_2 = \{0, 12, 24, 36, 48, ...\}$ thus giving $S = A_1 \cup A_2 = \{0, 5, 10, 12, 15, 20, 24, ...\}$. We will denote an arithmetic sequence with interval $\delta$ as $A(\delta)$. Additionally, to simplify notation, we will say that $A_i = A(\delta_i)$.

Note that with the arithmetic sequences compression there is not necessarily a finite number of points defined. Therefore we do not use $n$ to refer to the decompressed size of the list, but rather ask for the number of elements to sort as additional input. A query to our sorting algorithm will intuitively read as "With this list of arithmetic sequences, what are, in order, the first $n$ points in the combined sequence?".

### 3.1 Priority Queue Sorting

The simplest way for sorting a sequence of points under an arithmetic sequences compression uses priority queues. This method, shown in Algorithm 1, begins by adding the first element of each sequence to a priority queue. The priority queue will therefore be built in $O(C \log C)$ time. From here we extract the element of minimum priority from the queue, we will call the value of this point $v$. This is put into an output data structure.

Since it is possible for two points from the same sequence to be consecutive in the sorted list, we must then replace the element just removed from the queue with the next from the same

sequence. This maintains the invariant which requires the smallest element from each sequence always be present in the priority queue.To accomplish this we first check from which sequence the extracted point came. Then, assume the point came from $A(\delta)$, we will simply add $\delta$ to this point and insert this new point into the queue. We repeatedly query the queue $n$ times, until the $n$ smallest elements are found in sorted order.

In the end, we will have performed $n$ inserts into and $n$ deletes from the priority queue. Since the queue is guaranteed to be no greater than size $C$ at any time, our algorithm will run in time $O(n \log C)$ time.

We also achieve linear space complexity, in terms of $n$. The largest data structure for this algorithm will be the output list, which is of size $n$. The priority queue will be no larger than $C$ at any time, and therefore as long as $C \ll n$ it will not be the dominant occupant of memory. Note that there is no benefit to giving the result compressed by arithmetic sequences, as that would undo the work just done. Therefore a reduction in space complexity is only achievable if a secondary form of compression is used.

In order to simplify the association of a point to its derivative sequence we use a data type `pair`, which is a pair of a value $v$ and the interval of its source sequence $\delta$. An object of this type will have priority $v$. Using this it is easy to see that the lookup will be constant time.

**Algorithm 1**: `pq_Sort`-A method for sorting arithmetic sequences using a priority queue. Here, `pair` contains an element $v$ which is the value of a point, and $\delta$ which is the interval for its source arithmetic sequence.

**Input**: set of $C$ arithmetic sequences $\mathcal{A} = \{A_i, ..., A_C\}$, the number of values to sort $n$
**Output**: set of $n$ ordered smallest values

initialize priority queue $pq$;
intialize an array of size $n$ *sorted*;
**foreach** $A_i \in \mathcal{A}$ **do**
    $pq$.insert(`pair`$(0, \delta_i)$);
**for** $i = 0; i < n; ++i$ **do**
    `pair` $p$ = $pq$.poll();
    $sorted[i] = p.v$;
    $pq$.insert(`pair`$(p.v + p.\delta, p.\delta)$);
**return** *sorted*;

### 3.2 $k$th Order Statistic

The regularity of the data for arithmetic-sequences compressed data allows for fast computation of the $k$th order statistic. The intuition used for performing this calculation is that an arithmetic sequence has similar appearance to a Poisson arrival process, which is the approach presented in Algorithm 2. To begin, we consider each sequence as a Poisson process with rate $\lambda = \frac{1}{\delta}$. Conceptually, this is just a shift from saying that "all points are $\delta$ units apart" to "$\frac{1}{\delta}$ points appear every unit".

We must now combine all of the processes (sequences) into a single process (sequence). The combined process can be seen as a single process with rate $\Lambda$ where each event will have one of $C$ types. If the probability of a point being of type $i$ is $p_i$, and $\sum_{i=1}^{C} p_i = 1$, then the combined process is equivalent to the combination of $C$ slower concurrent processes, each having rate

$\Lambda \cdot p_i$. Therefore we can say that $\Lambda = \sum_{i=1}^{C} \lambda_i = \sum_{i=1}^{C} \frac{1}{\delta_i}$. Conceptually, we are summing together all the sequences' arrival rates in order to say that, on average points appear every $\Lambda$ units, or equivalently, on average points are $\frac{1}{\Lambda}$ units apart.

We can now find the location of position $k$ in the combined sequence using this combined process. For this Poisson process the expected time of arrival of the $k$th event is simply $\frac{k}{\Lambda}$, or conceptually if points are an average of $\frac{1}{\Lambda}$ units apart then to get $k$ points one must look about $\frac{k}{\Lambda}$ units deep. We will label this "guess" of $\frac{k}{\Lambda}$ as $g$. We now must count the actual number of points which occur in $[0, g]$. For each sequence $A_i$ there will be $\lfloor \frac{g}{\delta_i} \rfloor$ points in the range. Therefore the actual number of points in $[0, g]$ is $\sum_{i=1}^{C} \lfloor \frac{g}{\delta_i} \rfloor$. By construction we know that $\sum_{i=1}^{C} \frac{g}{\delta_i} = k$, and it is also clear that for any sum of positive rationals $\sum_{i=1}^{j} \frac{a}{b} \leq \sum_{i=1}^{j} \lfloor \frac{a}{b} \rfloor + j$. Therefore, the number of points actually occurring in $[0, g]$ is between $k$ and $k - C$. The computation time bottlenecks of the above are the calculations of $\Lambda$ and $\sum_{i=1}^{C} \lfloor \frac{g}{\delta_i} \rfloor$, each of which can be calculated in $O(C)$ time.

**Algorithm 2**: `Index_Arithmetic`-Finds the $k$th element in the combined sequence. Here, `pair` contains an element $v$ which is the value of a point, and $\delta$ which is the interval for its source arithmetic sequence. The method `next_mult`$(a, b)$ finds the next multiple of $b$ which is greater than $a$.

**Input**: set of $C$ Arithmetic Sequences $\mathcal{A} = \{A_i, ..., A_C\}$, the index queried $k$
**Output**: value of the $k$th smallest element

```
k = k − C;
initialize priority queue pq;
Λ = 0;
foreach i < C do
    Λ += 1/δ_i;
d = k/Λ;
count = 0;
foreach i < C do
    count += ⌊d/δ_i⌋;
    pq.add(pair(next_mult(d, δ_i),δ_i));
Initialize value = 0
for i = 0; i < k − count; ++i do
    pair p = pq.poll();
    value = p.v;
    pq.insert(pair(p.v + p.δ, p.δ));
return value;
```

From here we can use a method similar to `pq_sort` presented in Section 3.1. We add the next element from each sequence into a priority queue. Then we remove the lowest priority point, calculate the next point from that sequence and insert it into the queue, and then repeat until we have reached $k - \sum_{i=1}^{C} \lfloor \frac{g}{\delta_i} \rfloor$ points. Finding the next multiple of some $\delta_i$ after $g$ can be done in constant time, as it is simply $\delta_i + g - (g \mod \delta_i)$. Since the priority queue will never be of size greater than $C$, and $0 < k - \lfloor \frac{g}{\delta_i} \rfloor < C$, the run time of this section is $O(C \log C)$. Therefore the k-th order statistic can be calculated in $O(C \log C)$ time, and is constant with respect to $k$. In contrast, a simple application of Algorithm [**?** ] would require $O(k \log C)$.

Note that since Poisson processes require that there be 0 events at time 0, we must actually

find event $k - C$ in the Poisson process to get the $k$th point in the sequence. More details on Poisson processes as well as proofs to the claims mentioned can be found in [22].

# 4 Lempel-Ziv '77 Compression

The LZ77 compression scheme was presented in [25] by Jacob Ziv and Abraham Lempel in 1977. With this scheme, a compression is a sequence of terms of one of two types: terminals and back pointers. A terminal is a character from the original alphabet of the string, call this $\Sigma$, and a back pointer is of the form $(back, length)$ where *back* is the index (in the uncompressed string) from which to start a substring and *length* is the number of characters to copy starting from *back*. As an example, consider the string **a b (1, 2) (2, 3) c (1, 5)**. The term **(1,2)** instructs the user to start at index 1 and copy 2 characters, and therefore **(1,2)** = **a b**. The whole sequence will therefore decompress into **a b ab bab c ababb**. It is possible for a back pointer to have *length* larger than the depth of *back* (that is $back - current\ location$). In this case it is understood that the referenced string is repeated to fill in the gap. For example, if we have the compression **a b (1,6)**, this would decompress into **a b ab ab ab**.

## 4.1 Sorting

The intuition behind Algorithm 3, which sorts a LZ77 compressed string, is that for any string $s$, where $lz(s)$ is a LZ77-compression of $s$, the set of terminals present in $s$ is equal to those in $lz(s)$. This means that in order to sort the decompressed string it suffices to sort all of $\Sigma$, and then count the number of each character which appears in the decompression. Therefore we begin by first copying all literals to a list. Next we sort the literals. These steps can be done in $O(C + |\Sigma| \log |\Sigma|)$ time.

The next step is to count the number of each type of literal in the decompressed string. To do this, we count characters while mimicking the action of the decompression. First we scan through the compression looking for the length of the deepest back pointer (the maximum $back - current\ location$). We then create a circular buffer of this size (we will call this variable *size*). Now we perform a normal decompression of the string, except whenever we would append a character to the decompression we instead write that character to the next space in the buffer, and iterate a counter for that character. When reading a back pointer we begin copying from that location in the circular buffer. Since we defined the length of the circular buffer to be depth of the deepest back pointer, we are guaranteed that the reference be in the circular buffer. In all, this step can be completed in $O(n)$ time, as it is effectively a decompression, giving the overall running time of the algorithm to be $O(C + |\Sigma| \log |\Sigma| + n)$. The advantage gained by using the circular buffer over a complete decompression is that this reduces the space complexity of the algorithm is $O(C + size)$.

With the multiplicity of each character we can then return a *LZ77* compressed sorted string in time $O(|\Sigma|)$. Assume $\Sigma = \{\sigma_1, \sigma_2, ...\sigma_{|\Sigma|}\}$, and that $\sigma_i$ has multiplicity $m_i$ in the string. Then the compressed string will be: $\sigma_1$ **(**1**,** $m_1 - 1$**)** $\sigma_2$ **(**$m_1 + 1$**,** $m_2 - 1$**) ...** $\sigma_{|\Sigma|}$ **(**$n - m_{|\Sigma|}$**,** $m_{|\Sigma|}$**)**.

## 4.2 Indexing

Algorithm 4 gives a method for finding the character at index $i$ of a LZ77 compressed string. The algorithm keeps track of two read heads. The one labeled $j$ is the current location in the compressed list, the one labeled *count* is the location in the decompressed string if all terms up to $j$ were decompressed.

**Algorithm 3**: `LZ77_Sort`-A method for sorting a LZ77 compressed string. Here *back* is the location of the back pointer index, and *length* is the number of characters to copy. It is assumed that if an index is not in $\Sigma$ then it is a back pointer.

**Input**: An LZ77-compressed list $LZ$
**Output**: An LZ77-compressed sorted list

```
1  initialize a list Lit;
2  initialize a table map where key ∈ Σ, value = 0;
3  initialize a circular buffer b with size = length of longest back reference;
4  foreach α ∈ LZ do
5      if α ∈ Σ ∧ α ∉ Lit then
6          Lit.insert(α);
7  Lit.sort();
8  initialize j = 0;
9  for i = 0; i < C do
10     if LZ[i] ∈ Σ then
11         b[j mod size] = LZ[i];
12         ++map.value(LZ[i]);
13         ++j;
14     else
15         for m = LZ[i].back; m < (LZ[i].back + LZ[i].length) do
16             b[j mod size] = b[m mod size];
17             ++map.value(b[m mod size];
18             ++j;
19 return map;
```

The first step in the algorithm is to scan the compression until we reach or pass the index $i$. This is done by advancing *count* by 1 if $LZ[j]$ is a literal, and by $LZ[j].length$ otherwise. If we reach index $i$ on a literal then we simply output that literal and terminate. If we end on a back pointer then we search backward in the compression until we go back by one term in the compressed data, and from this location we scan in reverse until we reach or pass $LZ[j].back + LZ[j].length - (count - i) - 1$. Again, if we end on a literal then we output that literal. Otherwise we repeat until we reach a literal. Assuming that the compression is well-formed, we will be guaranteed to eventually terminate on a literal. As an example, consider the compression **a b (2,1) (1,3) (4,3)** at index $i = 8$. The back pointer **(4,3)** occupies positions 7, 8, and 9 in the uncompressed string. Therefore to find position 8 we will look backward for position 5 in the uncompressed string. Position 5 is in the **(1,3)** term, which is not a literal. Therefore we will again look backward to position 2, which is **b**.

The most difficult step in the algorithm deals with those back pointers where the depth of the reference is less than the number of characters to be copied. This condition is handled in lines 11 and 12 of Algorithm 4. We know that we are in such a situation if $i > count - LZ[j].length$, as this says that our new index is still within the back pointer referenced by $j$. If we are in such a situation we first figure out the depth into the copy (given by $i - LZ[j].back$). We then must find the length of the string copied (given by $count - LZ[j].length - LZ[j].back + 1$). We are then able to figure out how far to go into the copied string by performing $(i - LZ[j].back) \mod (count -$

$LZ[j].length - LZ[j].back + 1)$. This new number is the distance we must go from the location of the back reference ($LZ[j].back$), thus our new index will be $\big((i - LZ[j].back) \mod (count - LZ[j].length - LZ[j].back + 1)\big) + LZ[j].back$.

The total running time of this algorithm will be $O(C)$. We first use time $O(C)$ to scan forward into the compression. Also, since no pointers point forward in the compression, we will be guaranteed to traverse backward in the compression to find a terminal in no more than $O(C)$ time. Thus the total running time of the algorithm is $O(C)$.

**Algorithm 4**: `LZ77_Index`-A method for indexing a LZ77 compressed string. Here *back* is the location of the back pointer index, and *length* is the number of characters to copy. It is assumed that if an index is not in $\Sigma$ then it is a back pointer.

**Input**: An LZ77-compressed list $LZ$, a query index $i$
**Output**: The $i$th element of a decompressed $LZ$

```
1  initialize count = 1;
2  initialize j = 1;
3  while count < i do
4      ++j;
5      if LZ[j] ∈ Σ then
6          ++count;
7      else
8          count+ = LZ[j].length;
9  while LZ[j] ∉ Σ do
10     i = LZ[j].back + LZ[j].length − (count − i) − 1;
11     if i > count − LZ[j].length then
12         i = ((i − LZ[j].back) mod (count − LZ[j].length − LZ[j].back + 1)) + LZ[j].back;
13     while count ≥ i do
14         if LZ[j] ∈ Σ then
15             count = count − 1;
16             j = j − 1;
17         else
18             count = count − LZ[j].length;
19             j = j − 1;
20     ++j;
21     if LZ[j] ∉ Σ then
22         count = count + LZ[j].length;
23     else
24         ++count;
25 return LZ[j];
```

# 5 Context Free Grammar Compression

Using a context free grammar, one is able to represent a long string as a relatively short grammar. That is, we list a series of variables (call this set $V$), and literals (call this set $\Sigma$), as well as a list of rules for variable substitution. For example, consider the string **aababbabbb**. This can be translated into the context free grammar:

$$\begin{aligned} A_0 &\to \mathbf{a}A_1A_2A_3 \\ A_1 &\to \mathbf{ab} \\ A_2 &\to A_1\mathbf{b} \\ A_3 &\to A_2\mathbf{b} \end{aligned}$$

## 5.1 Sorting

Similar to Algorithm 3 for sorting LZ77 compressed strings, Algorithm 5 relies on the intuition that all literals in the uncompressed string must occur in the compression. Therefore, again, we begin by first finding and sorting $\Sigma$, which will take $O(C + |\Sigma| \log |\Sigma|)$ time. Next we turn the context free grammar into a dependency graph. For this, we say that if the variable $v_0 \in V$ has in its substitution rule $v_1 \in V$, then $v_0$ depends on $v_1$. Since the context free grammar may only produce a single string, we are guaranteed that the resulting dependency graph be acyclic. The dependency graph for the above context free grammar is shown in Figure 1. Since the graph is acyclic, we are able to compute a topological sort on the graph. Since there are $|V|$ vertices and $C$ edges, and $|V| \leq C$, the topological sort will take $O(C)$ time.

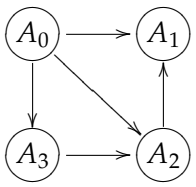


Figure 1: The dependency graph for the context free grammar given in Section 5.

We will then consider each literal as a vector of $|\Sigma|$ dimensions in such a way that for the minimal element $\sigma_1 \in \Sigma$, $\sigma_1 = (1, 0, ..., 0)$, and the next smallest element $\sigma_2 = (0, 1, 0, ..., 0)$, and so on. As a notational convenience we will say that $\langle\sigma\rangle$ refers to the respective vector for symbol $\sigma$.

The final step is to follow backwards through the topological sort and sum up each symbol's respective vector as we come to it. In the example given we begin with $a = (1, 0)$ and $b = (0, 1)$. We then calculate the vector $\langle A_1\rangle = \langle\mathbf{a}\rangle + \langle\mathbf{b}\rangle = (1, 0) + (0, 1) = (1, 1)$. We can then calculate $A_2 = (1, 2)$. Eventually we will calculate the start symbol $A_0 = (4, 6)$, which says that in the decompressed string there are 4 $a$'s and 6 $b$'s. In this final step we must do exactly $C$ vector additions, each one taking $|\Sigma|$ time, thus giving a total sort time $O(C \cdot |\Sigma|)$.

We are now able to return to the user a context free grammar of size $|\Sigma| \log n$. This is done by writing a grammar in which for each letter in the alphabet, we have $\log n$ variables, where each doubles the number of that letter represented. So, for example, if we wanted a grammar which represents the string $\mathbf{a}^8\mathbf{b}^{16}$, our grammar would be:

$$\begin{aligned} S_0 &\to A_2B_3 & B_0 &\to \mathbf{bb} \\ A_0 &\to \mathbf{aa} & B_1 &\to B_0B_0 \\ A_1 &\to A_0A_0 & B_2 &\to B_1B_1 \\ A_2 &\to A_1A_1 & B_3 &\to B_2B_2 \end{aligned}$$

**Algorithm 5**: `Sort_CFG`-Sorts a context free grammar. Here, $\Sigma$ is the alphabet of the string represented by the grammar, and $V$ is the set of variables.

**Input**: Context free grammar $CFG$
**Output**: The sorted string represented by $CFG$ with start variable $A_0$

```
Convert the Variables in CFG into a dependency graph G;
perform a topological sort on G;
reorder rules in CFG to be the reverse of G;
Sort Σ;
transform each literal into a |Σ|-dimensional vector;
foreach rule r ∈ CFG do
    initialize a vector sum = ⟨0⟩;
    foreach Symbol S ∈ Σ ∪ V listed in r do
        sum+ = ⟨S⟩
    Σ = Σ ∪ {A + r}, where A_r is the variable associated with rule r;
    ⟨A_r⟩ = sum;
return ⟨A_0⟩;
```

# 6 Lempel-Ziv '78 Compression

In addition to the LZ77 compression scheme presented above, Lempel and Ziv in 1978 presented a secondary compression scheme (here on out called LZ78) [26]. Each term in this scheme is a pair of a natural number $back \in \mathbb{N}$ and a character $\sigma \in \Sigma$. The rule for decompression is to copy the string represented by term $i$, then append $\sigma$. For example **(0,a) (1,b) (0,b) (2,a) (3,a) (2,b)** will become **a ab b aba ba abb**.

## 6.1 Sorting

This algorithm acts similarly to `sort_cfg` in that we sort the symbols, and then accumulate vectors respresenting symbol multiplicity. As before we will sort the alphabet $O(|\Sigma| \log |\Sigma|)$. We will then define $\langle \sigma_i \rangle$, where $\sigma_i$ is the $i$th symbol in sorted order, to be a $|\Sigma|$-dimensional vector where all terms are 0, save the $i$th term which is 1. Thus $\langle \sigma_1 \rangle = (1, 0, 0, ...)$.

The reader should be aware that LZ78 compression is actually a restricted version of context-free grammar compression. The only difference (other than notation) between the two is that while cfg compression allows an unbounded number variables and literals be concatenated together in a substitution rule, LZ78 only allows for a single variable and a single literal be present. The example above can be expressed as as cfg as follows:

$$\textbf{(0,a) (1,b) (0,b) (2,a) (3,a) (2,b)} \longrightarrow \begin{array}{ll} A_0 \to A_1A_2A_3A_4A_5A_6 & A_4 \to A_2\mathbf{a} \\ A_1 \to \mathbf{a} & A_5 \to A_3\mathbf{a} \\ A_2 \to A_1\mathbf{b} & A_6 \to A_2\mathbf{b} \\ A_3 \to \mathbf{b} & \end{array}$$

With this construction it is clear that the variables in this cfg are already in topological-sorted order. Therefore the algorithm for sorting such a compression will be a simplified version of `cfg_sort`.

We will first create a second list of terms to parallel the compression. In this list, instead of pairs, we will have these $|\Sigma|$-dimensional vectors as described above. Next we will scan through the compressed sequence. For each term we first check if $back == 0$. If this is so then we know that this term in the compression represents a single character, call this $\sigma$. Thus we add into the array at this index the vector $\langle\sigma\rangle$. Otherwise if $back > 1$ we will add together $map[back]$, the count for the referenced string, and $\langle\sigma\rangle$, where $\sigma$ is the character to append for this term.

The data structure which uses the most space in this algorithm is the array of vectors. In total there will be $C$ items in this vector, each with $|\Sigma|$-dimensional vector. Therefore the space complexity of this algorithm is $O(|\Sigma| \cdot C)$. The time complexity is also $O(|\Sigma| \cdot C)$, as we scan through the compressed list at most once, and at each turn we will do at most one constant-time array access, and one vector addition (each of which will take $|\Sigma|$ time).

**Algorithm 6**: `LZ78_Sort`-A method for sorting a LZ78 compressed string. Here *back* is the location of the back pointer index, and $\sigma$ is the symbol to append.

**Input**: An LZ77-compressed list $LZ$
**Output**: An LZ77-compressed sorted list

initialize an array $map$ where elements are in $\mathbb{N}^{|\Sigma|}$;
initialize a $|\Sigma|$-dimensional vector $sum = \langle 0 \rangle$;
**for** $i = 1; i < |LZ|; i++$ **do**
    **if** $LZ[i].back == 0$ **then**
        $map[i] = \langle LZ[i].\sigma \rangle$;
    **else**
        $map[i] = map[(LZ[i].back)] + \langle LZ[i].\sigma \rangle$
    $sum$ += $LS[i]$;
**return** $sum$;

# 7 Conclusion

The primary contribution of this paper is its presentation of various sorting algorithms which operate on compressed data. Not only does this save the user time from decompressing and then recompressing data in order to perform a sort, but operating on the compressed data may give a performance benefit. In fact, the algorithms presented in this paper will guarantee large speed up whenever $C \ll n$.

Algorithm 1 (priority queue sorting on data compressed by arithmetic sequences) will run in $O(n \log C)$ time, and will therefore be no worse than a decompression followed by a sort, even for inputs with a poor compression ratio. Algorithm 3 (sorting on data compressed by LZ77), by running in $O(C + |\Sigma| \log |\Sigma| + n)$, effectively gives linear sorting time in $n$. Finally, Algorithm 5 (sorting on data compressed by a context free grammar), by running in $O(C \cdot |\Sigma|)$, gives a sorting time independent of $n$.

We also present other noteworthy algorithms for indexing and finding $k$th order statistics. Algorithm 2 provides $O(C \log C)$ running time for computing the $k$th order statistic of a set of data compressed by arithmetics sequences. Algorithm 4 provides a method for indexing into data compressed by LZ77 in $O(C)$ time. If this indexing is done into a sorted LZ77 compressed data then this will give the $k$th order statistic. One could use previous work done on random

access on sorted grammar-compressed data, such as the method presented in [5], in order to find the $k$th order statistic for grammar-compressed data.

## References


[1] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1993.

[2] A. Amir, G. M. Landau, and D. Sokol. Inplace 2d matching in compressed images. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 853–862, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[3] O. Arbell, G. M. Landau, and J. S. B. Mitchell. Edit distance of run-length encoded strings. *Inf. Process. Lett.*, 83(6):307–314, September 2002.

[4] P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Trans. Algorithms*, 6(1):3:1–3:14, December 2009.

[5] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 373–389. SIAM, 2011.

[6] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Inf. Process. Lett.*, 54(2):93–96, April 1995.

[7] P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *Proceedings of the First international computer science conference on Theory and Applications*, CSR'06, pages 127–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] M. Crochemore, G. M. Landau, and M. Ziv-ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *In Symposium of Discrete Algorithms (SODA*, pages 679–688, 2002.

[9] R. Dugad and N. Ahuja. A fast scheme for image size change in the compressed domain. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11(4):461 –474, apr 2001.

[10] B. L. Robinson G. Robins and B. S. Sethi. On detecting spatial regularity in noisy images. *Information Processing Letters*, 69:189–195, 1999.

[11] D. Hermelin, G. M. Landau, S. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)*, volume 3 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 529–540, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] A.B. Kahng and G. Robins. Optimal algorithms for determining regularity in pointsets. In *Proc. Canadian Conference on Computational Geometry*, pages 167–170, 1991.

[13] A.B. Kahng and G. Robins. Optimal algorithms for extracting spatial regularity in images. In *Pattern Recognition Letters, 12*, pages 757–764, 1991.

[14] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over ziv-lempel compressed text. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, COM '00, pages 195–209, London, UK, UK, 2000. Springer-Verlag.

[15] J. Kärkkäinen and E. Ukkonen. Lempel-ziv parsing and sublinear-size index structures for string matching (extended abstract). In *Proc. 3rd South American Workshop on String Processing (WSP'96*, pages 141–155. Carleton University Press, 1996.

[16] Y. Lifshits. Processing compressed texts: a tractability border. In *Proc. CPM 2007*, pages 228–240. Springer, 2007.

[17] J. J. Liu, G. S. Huang, Y. L. Wang, and R. C. T. Lee. Edit distance for a run-length-encoded string and an uncompressed string. *Inf. Process. Lett.*, 105(1):12–16, January 2008.

[18] V. Mäkinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, CPM '01, pages 31–49, London, UK, UK, 2001. Springer-Verlag.

[19] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, CPM '94, pages 113–124, London, UK, UK, 1994. Springer-Verlag.

[20] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proceedings of the Data Compression Conference*, DCC '01, pages 459–, Washington, DC, USA, 2001. IEEE Computer Society.

[21] G. Robins and B. L. Robinson. Landmine detection from inexact data. In *Proc. International Symp. on Aerospace/Defence Sensing and Dula-Use Photonics*, pages 189–195, 1994.

[22] S. M. Ross. *Introduction to Probability Models*, chapter Chapter 5, pages 312–339. Academic Press, 10th edition edition, 2010.

[23] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, CIAC '00, pages 306–315, London, UK, UK, 2000. Springer-Verlag.

[24] R. V. Babu and K.R. Ramakrishnan. Compressed domain human motion recognition using motion history information. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on*, volume 3, pages III – 41–4 vol.3, april 2003.

[25] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343, 1977.

[26] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.*, 24(5):530–536, September 2006.