On Practical Data Structures for Sorted Range Reporting

Golnaz Badkobeh[1], Sehar Naveed[2], and Simon J. Puglisi[3]

[1] Department of Computer Science, City University of London, United Kingdom
golnaz.badkobeh@city.ac.uk
[2] Department of Computer Science, Goldsmiths University of London, United Kingdom
snave001@gold.ac.uk
[3] Department of Computer Science, University of Helsinki
Helsinki Institute for Information Technology (HIIT), Helsinki, Finland
simon.puglisi@helsinki.fi

Abstract. Given an array $A[1, n]$ of integers, the sorted range reporting problem is to preprocess A in order to later answer queries of the form sort(i, j), which should return an array of length $j-i+1$ that contains the contents of $A[i, j]$ sorted in ascending order. When applied to the suffix array, sorted range reporting can be applied to solve several string processing problems, including, for example, non-overlapping pattern matching and variable-length gapped pattern matching. In this paper we explore the practical performance of solutions for sorted range reporting. Our experiments show that the choice of solution depends on the interval size. For very short intervals, manually sorting the range at query time beats asymptotically optimal methods, while for longer intervals, a data structure that we describe that precomputes sorted blocks of varying sizes which are then merged at query time is the method of choice.

Keywords: sorted range reporting, suffix array, gapped pattern matching

1 Introduction

The suffix array, $SA[1, n]$, of a string $T[1, n]$ is a permutation of the integers $[1, n]$ such that suffix $T[SA[i]..n]$ is the ith suffix in lexicographical order amongst all suffixes of T. For example, the suffix array of string

$$T = actagtatctcccgtagtac\$$$

is

$$SA = 21, 19, 1, 16, 4, 7, 20, 11, 12, 13, 2, 9, 17, 14, 5, 18, 15, 3, 6, 10, 8$$

Many string processing problems can be solved efficiently both in theory and in practice using the suffix array. A property of the SA that makes it useful for many problems is that all positions of occurrence of any substring that occurs in T are contiguous in SA. For example, in the above string, the substring ta occurs at positions 3, 6, 15, and 18 — and these positions occur together in SA, in the subarray $SA[16, 19] = 18, 15, 3, 6$. This is a consequence of placing the suffixes in lexicographical order. Observe, however, that while the occurrences of the substring do form a contiguous range in SA, the lexicographical order also means that in general the positions do not necessarily occur in the same order as they are in the string (string order).

For several important string processing problems it is desirable (even necessary) to have the occurrences in string order. A compelling example from computational

Golnaz Badkobeh, Sehar Naveed, Simon J. Puglisi: On Practical Data Structures for Sorted Range Reporting, pp. 42–49.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5 © Czech Technical University in Prague, Czech Republic

biology is the variable-length gap pattern matching (VLG) problem [5], in which we seek matches to a regex-like pattern that is composed of subpatterns (strings) and bounds on the distance allowed between subpatterns. For example, $ta[2,6]ac$ asks for positions in the string where an occurrence of ta is followed by an occurrence of ac at least 2 positions, but at most 6 positions away.

The suffix array allows us to easily find the occurrences of the two subpatterns, but it is not until we sort those occurrences into ascending order that we can easily apply the gap constraint to arrive at the answer (there is one such occurrence in the example string above, starting at position 15).

Another example is the *non-overlapping pattern matching* problem, given a pattern, the task is to output a maximal set of pattern occurrences in T that do not overlap each other. Clearly, being able to enumerate pattern occurrences in string order would be advantageous. Another compelling set of problems comes from document retrieval [9].

In this paper we examine methods that allow subarrays of an integer to be returned in sorted order—a problem known in the literature as *sorted range reporting* [4]. The methods we will consider can be applied to general arrays of integers, not necessarily the suffix array—our motivation, however, is to eventually apply these methods to problems in string matching, in particular VLG matching. Our focus is on methods that are performant in practice. We find that manually sorting the query range with each query is often faster in practice than an asymptotically optimal data structure due to Brodal et al. [4]. We also describe an alternative data structure to Brodal et al. [4] which presorts blocks of the input array and merges a relevant subset of them covering the query. Our experiments show our method beats the above methods on all but the short query ranges.

1.1 Related Work

The sorted range reporting problem is stated formally as follows. We are given an array, $A[1, n]$ of integers in the range $[1, n]$. We are allowed to preprocess the array and build a data structure to later help us answer queries of the form $\text{sort}(i, j)$ that should return an array containing the contents of $A[i, j]$ sorted in ascending order.

The problem was first directly studied by Mäkinen and Navarro [8], and later by Navarro and Nekrich [10]. The related problem of range predecessor queries was studied by Belazzougui and Puglisi [3]. Their $O(n)$ space $O(\sqrt{\log n})$-time query data structure implies a solution to sorted range reporting with query time $O(k\sqrt{\log n})$, where $k = j - i + 1$ is the size of the query range.

Brodal et al. [4] describe an optimal solution to sorted range reporting, proposing a RAM model data structure that takes $O(n)$ words of space and allows for queries to be answered in optimal $O(k)$ time, i.e., the query time is directly proportional to the number of elements reported. The preprocessing phase, which organizes the data to enable these efficient queries, also takes $O(n \log n)$, aligning with the optimal sorting time for n elements. We examine this data structure further in Section 2.2.

Another related problem is range selection queries (also known as range quantile queries). Akram and Saxena [2] present a linear space solution with $O(k \log k)$ query time and $O(n)$ preprocessing time, reporting output elements in non-decreasing order, using a data structure consisting of an RMQ structure and a binary min-heap and offering a straightforward and easy-to-implement alternative to the solution by Brodal

et al [4]. A broader range selection problem, where the output elements are not required to be reported in sorted order, is previousy studied in [1]. Skala [11] provides a comprehensive survey of array range query problems.

2 Sorted Range Reporting

In this section we describe the three solutions to the sorted range reporting problem that we implemented and measured in this study.

2.1 Simple Sorted Range Reporting

Perhaps the simplest approach to sorted range reporting one can think of is to just copy and sort the query range in order to answer each query. This method represents a natural baseline against which the performance of more sophisticated data structures for sorted range reporting should be measured.

We implemented this baseline, which we refer to as Simple Sorted Range Reporting (ssrr) as follows. For each index in the range from i to j (inclusive), the elements of A are copied to an array vector v. Once the elements are copied, the contents of v are sorted using the `std::sort` function and then v is returned.

The running time is $O(k \log k)$, where $k = j - i$ is the length of the query range.

Algorithm 1 Simple Sorted Range Reporting

```
function QUERY(i, j, v)
    Step 1: Copy elements from A to v within the specified range [i, j]
    std::copy( A.begin() + i, A.begin() + j + 1, v.begin())
    Step 2: Sort the elements in the v vector
    std::sort(v, j − i + 1)
    return v
end function
```

2.2 Brodal et al.'s Data Structure

We implemented the asymptotically optimal sorted range reporting approach of Brodal et al. [4]. To our knowledge this is the first implementation of their data structure. In the interest of being self contained, we provide an overview of the data structure below, but refer the reader to [4] for full details.

We construct local rank labellings for each $r \in \{0 \cdots \log\log n\}$ as follows (the rank of an element x in a set X is defined as $|\{y \in X | y < x\}|$). For each r, the input array is divided into $\lceil n/2^{2^r} \rceil$ consecutive subarrays each of size 2^{2^r} (except possibly the last subarray), and for each element $A[x]$ the r'th local rank labelling is defined as its rank in the subarray $A[\lfloor x/2^{2^r} \rfloor 2^{2^r} \cdots (\lfloor x/2^{2^r} \rfloor + 1)2^{2^r} - 1]$. Thus, the r'th local rank for an element $A[x]$ consists of 2^r bits. All local rank labels of length 2^r can be stored using space $O(n2^r + w)$ bits, where by w we denote the word length in bits, and assume that $w \geq logn$. For all $\lceil \log\log n \rceil$ local rank labellings, the total number of bits used is $O(w \log\log n + n \log n) = O(nw)$ bits. All local rank labellings can be built in $O(n \log n)$ time while performing mergesort on A. The r'th structure is built by writing out the sorted lists, when we reach level 2^r . Given a query for $k = j - i + 1$ elements, we find the r for which $2^{2^r} - 1 < k \leq 2^{2^r}$. Since each subarray in the r'th local rank labelling contains 2^{2^r} elements, we know that i and j are either in the same

or in two consecutive subarrays. If i and j are in consecutive subarrays, we compute the start index of the subarray where the index j belongs, i.e. $x = \lfloor j/2^{2^r} \rfloor 2^{2^r}$. We then radix sort the elements in $A[i, x-1]$ using the local rank labels of length 2^r. This can be done in $O(k)$ time using two passes by dividing the 2^r bits into two parts of 2^{r-1} bits each, since $2^{2^{r-1}} < k$. Similarly we radix sort the elements from $A[x, j]$ using the labels of length 2^r in $O(k)$ time. Finally, we merge these two sorted sequences in $O(k)$ time, and return the k smallest elements. If i and j are in the same subarray, we just radix sort $A[i, j]$.

2.3 Sorted Range Reporting via Presorted Blocks

We now describe a simple alternative to Brodal et al.'s data structure that takes $O(n \log n)$ words of space and answers queries in $O(k \log k)$ time, where k is the length of the query range. The structure and accompanying query algorithm are very simple, but, to our knowledge, novel.

2.4 Data Structure

We assume for ease of exposition that n is a power of 2 (the general case is easy to deal with in practice and without affecting asymptotic bounds).

The data consists of $\log n$ levels, each an array of n elements. Denote the array at the ith as L_i. At the ith level, the input array A is conceptually divided into $n/2^i$ blocks, each of length 2^i. At each level, the elements of each block are sorted and concatenated to form an array of length n and thus the overall data structure size is $n \log n$ words and construction time is $O(n \log^2 n)$. We keep an array of $\log n$ pointers so that the start of each level can be accessed in constant time. At a given level, the start of the jth block can be access in constant time at position $j2^i$.

2.5 Query Algorithm

To answer queries, we proceed in two phases. The first phase identifies a set of presorted blocks from different levels of the data structure that allow the range to be covered. For query range $[i, j]$, the cover can be determined recursively as follows. Let $k = j - i + 1$. We first determine the largest full block contained in the range. This takes $\log k$ time by simply testing each block size that can possibly fit in the range. We then recurse on the uncovered prefix and suffix.

We remark that to save space, lower levels of the data structure, where blocks are very short, can be omitted from the data structure and reconstructed on demand at query time. This idea is illustrated in Fig. 1.

The second phase of the query algorithm merges the contents of the sorted blocks that cover the query range as determined in the first phase. There are $O(\log n)$ ranges to merge. We allocate an output buffer of k elements, which will eventually contain the elements of query range sorted in ascending order. We insert pointers to the start of each of ranges into a min heap with key equal to the element at the start of each range. A standard multi-way merge is then performed: we extract min from the heap, add the element to the output array, increment the pointer to point to the next element of the block, and then (if elements remain in the block) reinsert the block to the heap with the next element as the sort key. This process is repeated until the output buffer is full. Fig. 2 shows an example.

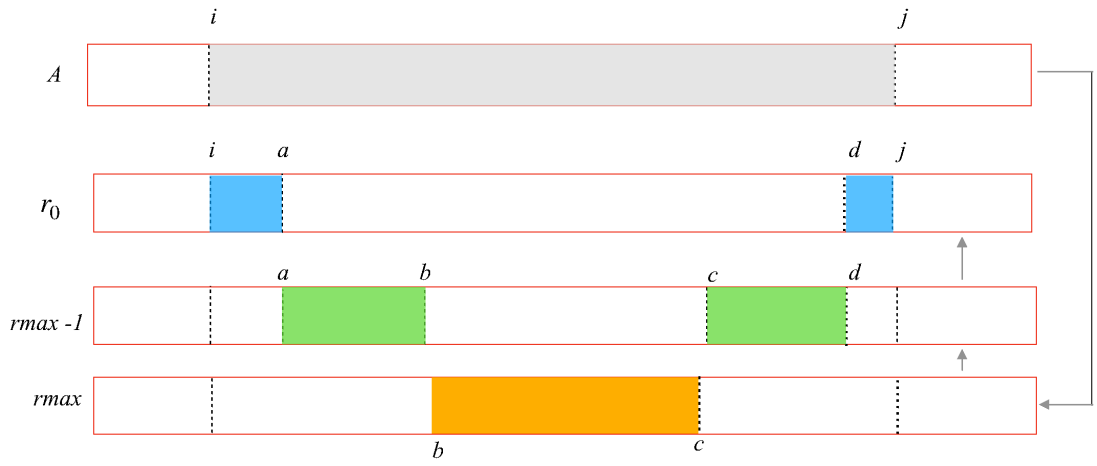

Figure 1. Block selection process in Block Sorted Range Reporting data structure. The orange and green ranges are the presorted blocks that best cover the range as identified in the first phase of the query algorithm. Blue areas at the start and end are manually sorted. Then all five regions are merged and returned in answer to the query.

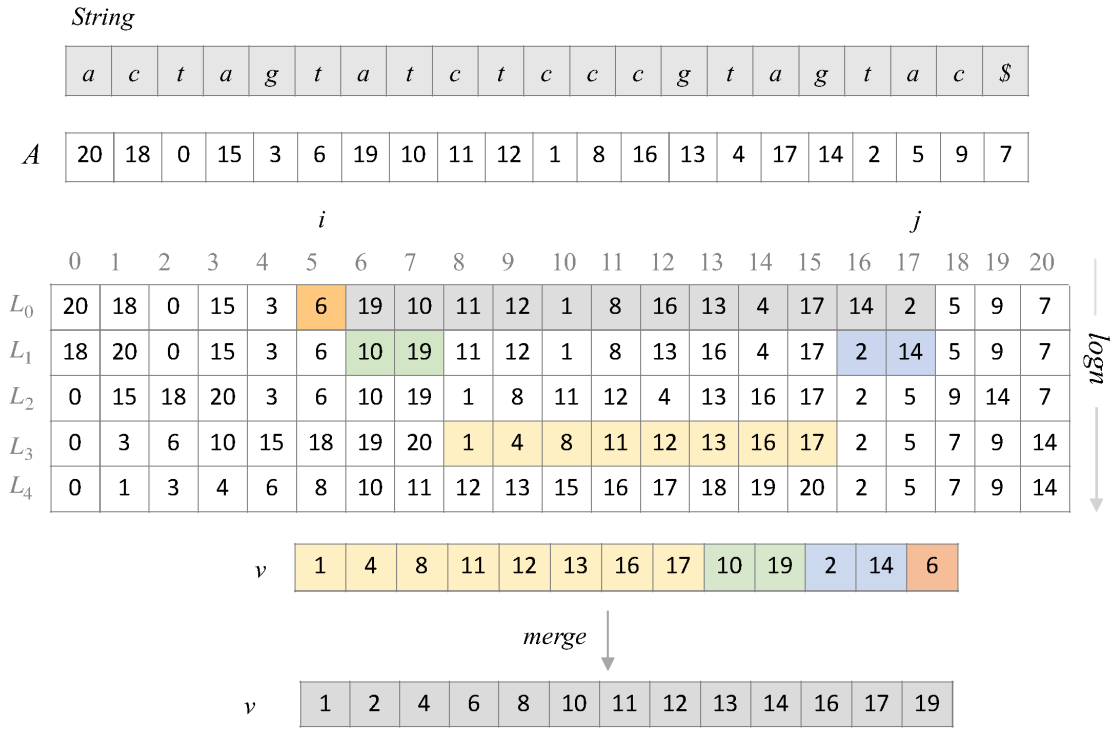

Figure 2. An example of using Block Sorted Range Reporting to report the $k = 13$ elements in sorted order for the text *actagtatctcccgtagtac* for a query $[i, j] = [5, 17]$.

3 Experimental Evaluation

We implemented the three methods described in the previous section and measured there performance in experiments described below.

3.1 Experimental setup

Environment. Our test machine comprises an Intel Core i7-4790 CPU @ 3.60GHz with a CPU cache size of 8 MiB (1 instance) and a total memory of 23Gi operated on Ubuntu 24.04 with an x86_64 architecture, running kernel version 6.8.0-31-generic and employing g++ 13.2.0 as the compiler version.

Test data. We generated suffix arrays from the texts below, from the Pizza & Chilli Corpus, which are available at `https://pizzachili.dcc.uchile.cl/texts.html`. Each text is 100MiB long. A brief description of each follows:

- *english* is the concatenation of English text files selected from various collections of Gutenberg Project. The file *english* are prefixes of the original text.
- *proteins* dataset comprises newline-separated protein sequences, devoid of descriptions, sourced from the Swissprot database. Each of the 20 amino acids is encoded as a single uppercase letter.
- *Para* dataset, provided by the Saccharomyces Genome Resequencing Project contains 36 sequences of Saccharomyces paradoxus. It includes four nucleotide bases: A, C, G, T.
- *kernel* dataset comprises the source code for all 332 Linux kernel versions downloaded from kernel.org. The dataset exhibits high redundancy, as only minor modifications are present between subsequent versions.
- *ecoli* is a dataset for protein localization. It contains 336 E.coli proteins split into 8 different classes.

Queries. Our experiments aimed to gauge the effect of query range size on query time. The queries were randomly generated with the fixed query lengths: $k_{100} = 100$, $k_{500} = 500$, $k_{1k} = 1000$, $k_{5k} = 5000$, $k_{10k} = 10000$, $k_{50k} = 50000$, where k is defined as $k = j - i + 1$, totaling a set of 10,000 queries for each separate run. To ensure that the evaluation covered a wide spectrum of potential real-world scenarios, a set of *mixed* random queries of varied lengths was also generated to test the performance and efficiency of the multi-level data structure.

3.2 Results

Measured runtimes are shown in Table 1. The experimental results demonstrate that the block-sorted range reporter (bsrr) significantly outperforms both simple sorted range reporter (ssrr) and the asymptotically optimal sorted range reporter (srr) across all datasets and query lengths. The performance of all methods is quite stable. High constant factors in the asymptotically optimal srr method cause it to yield to the others.

The gap between bsrr and ssrr widens as the query length increases, to about a factor of four. bsrr reports markedly better performance in handling mixed query lengths compared to ssrr and srr, achieving speeds that are three times as fast as ssrr and almost five times faster than srr.

Dataset	Query Lengths						
	k_{100}	k_{500}	k_{1k}	k_{5k}	k_{10k}	k_{50k}	mixed
english							
bsrr	**1.3691**	**5.6404**	**11.177**	**58.5259**	**120.516**	**597.531**	**129.371**
ssrr	1.9743	12.1864	26.6443	173.665	357.007	2004.59	412.025
srr	3.5908	27.1614	53.4557	263.955	520.584	2723.81	592.948
proteins							
bsrr	**1.3915**	**5.7007**	**11.1467**	**58.3191**	**120.345**	**597.666**	**129.51**
ssrr	1.9881	12.1868	28.2491	161.127	349.596	2002.54	411.85
srr	3.5665	27.0364	53.3736	263.589	520.454	2724.27	592.913
para							
bsrr	**1.3946**	**5.6618**	**11.4455**	**58.8152**	**121.534**	**608.432**	**131.641**
ssrr	2.0393	11.4455	27.4848	166.621	359.859	2056.76	424.777
srr	3.5805	27.2729	53.5191	262.754	519.137	2723.27	593.17
kernel							
bsrr	**1.2543**	**5.2102**	**9.912**	**53.956**	**111.399**	**582.988**	**123.459**
ssrr	1.9395	12.0159	26.5149	160.357	347.817	1985.71	408.186
srr	3.5612	27.0947	53.442	262.831	518.654	2698.25	588.144
E.coli							
bsrr	**1.4047**	**6.1667**	**11.3705**	**61.4457**	**122.473**	**610.411**	**132.034**
ssrr	2.0352	12.5351	27.4683	167.365	372.692	2068.67	423.631
srr	3.5742	27.0499	53.3785	263.879	518.184	2729.38	593.692

Table 1. The average query time in microseconds for the three methods block-sorted range reporter (bsrr), simple sorted range reporter (ssrr) and the asymptotically optimal sorted range reporter (srr) for query lengths $k_{100} = 100$, $k_{500} = 500$, $k_{1k} = 1000$, $k_{5k} = 5000$, $k_{10k} = 10000$, $k_{50k} = 50000$ and *mixed* random queries of varied lengths over a set of 10,000 queries.

4 Conclusions

This paper has explored the performance of sorted range reporting structures in practice. Our experiments have shown that our implementation of the asymptotically optimal structure, srr, of Brodal et al. [4] is beaten by both a simple method, ssrr, that simply sorts the contents of the query range each time, and by a data structure we introduce, bsrr, that presorts blocks of different sizes and merges these blocks at query time. This later method is consistently faster than the other methods. We remark that srr and bsrr complement each other and can be easily combined by leaving out the lower levels of the bsrr structure and simply resorting to sorting for short ranges.

In future work, we plan to sparsify our bsrr data structure so that it only includes some levels. This may require more merging to be done for some queries, but is an effective way to reduce data structure size. Moreover, we can reasonably expect a higher-degree merge to not adversely effect overall query time, as blocks are still few and processing of their contents will remain cache efficient. We aim to include an exploration of this approach in the full version of this paper.

Finally, one can further reduce the space usage of the bsrr data structure by storing sorted blocks in a compressed representation designed for sorted sequences, such as Elias-Fano [6,7] or by storing gaps between elements and using variable length codes. This should be particularly effective for large blocks.

References

1. P. Afshani, G. S. Brodal, and N. Zeh: *Ordered and unordered top-k range reporting in large data sets*, in Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2011, pp. 390–400.
2. W. Akram and S. Saxena: *Sorted range reporting and range minima queries*. arXiv preprint arXiv:2104.02461, 2021.
3. D. Belazzougui and S. J. Puglisi: *Range predecessor and lempel-ziv parsing*, in Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016, R. Krauthgamer, ed., SIAM, 2016, pp. 2053–2071.
4. G. S. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz: *Online sorted range reporting*, in Algorithms and Computation: 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings 20, Springer, 2009, pp. 173–182.
5. M. Cáceres, S. J. Puglisi, and B. Zhukova: *Fast indexes for gapped pattern matching*, in SOFSEM 2020: Theory and Practice of Computer Science: 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20–24, 2020, Proceedings 46, Springer, 2020, pp. 493–504.
6. P. Elias: *Efficient storage and retrieval by content and address of static files*. Journal of the ACM (JACM), 21(2) 1974, pp. 246–260.
7. R. M. Fano: *On the number of bits required to implement an associative memory*, Massachusetts Institute of Technology, Project MAC, 1971.
8. V. Mäkinen and G. Navarro: *Position-restricted substring searching*, in LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings, J. R. Correa, A. Hevia, and M. A. Kiwi, eds., vol. 3887 of Lecture Notes in Computer Science, Springer, 2006, pp. 703–714.
9. G. Navarro: *Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences*. ACM Computing Surveys, 46(4) 2014, p. article 52, 47 pages.
10. Y. Nekrich and G. Navarro: *Sorted range reporting*, in Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings, F. V. Fomin and P. Kaski, eds., vol. 7357 of Lecture Notes in Computer Science, Springer, 2012, pp. 271–282.
11. M. Skala: *Array Range Queries*, Springer Berlin Heidelberg, 2013, p. 333–350.