Elias-Fano and RMQ Implementation

Tobias Theuer
KIT, Germany

Abstract

This document gives an overview over my implementation of the *Elias-Fano* (EF) encoding [2] [3] [5] for answering predecessor queries as well as my implementation of several *RMQ* data structures. In particular, I focus on my implementations of various Bitvectors, which are used as part of the Elias-Fano encoding and a succinct RMQ implementation. My EF implementation achieves expected $\mathcal{O}(\log \mathcal{U})$ time for uniformly distributed numbers within an interval $I = [x_{min}, x_{max}] \in \mathcal{U}$ and $\mathcal{O}(\log n + \log \mathcal{U})$ worst-case complexity for a single predecessor query, where n is the number of stored values and $\mathcal{U}$ the universe of possible numbers.[1] My RMQ implementations achieve $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$ time, depending on the implementation.

2012 ACM Subject Classification None

Keywords and phrases Elias-Fano, RMQ

[1] In the implementation, $\mathcal{U} = \left(0, 2^{64}\right]$, so technically, there are at most $2^{64} \in \mathcal{O}(1)$ unique numbers in I and all queries could be answered in $O(1)$ time and space, but this isn't relevant in practice.

1 Algorithms and Data Structures

1.1 Bitvectors

I implemented several Bitvectors, which will be described in chronological order of implementation. Almost all of these Bitvectors take several hyper-parameters at compile time, which can be tweaked for different space-time trade-offs.

- The **Cache-Efficient Bitvector** answers `rank` queries with at most 1 cache miss; however, it is not very fast for `select` queries and uses twice as many bits as necessary to store the bit sequence because it only uses half of each cache line to store bits and the other half for meta data[2].
- The **Classical Rank Bitvector** is very close to the one from the slides but doesn't store `select` metadata. Instead, like the Cache-Efficient Bitvector, it relies on an optimized binary search over `rank` metadata for `select`. Blocks and superblocks have fixed sizes, which can be chosen at compile time.
- The **Trivial Bitvector** simply precomputes all possible `rank` and `select` queries. If the answer to one query can be represented using k bits, it needs $2kn$ bits in total: There are n $\texttt{rank}_1$ entries and a total of n $\texttt{select}_0$ and $\texttt{select}_1$ entries.[3] The original bit sequence can be reconstructed using $\texttt{rank}_1(i+1) - \texttt{rank}_1(i)$.
- The **Recursive Bitvector**[4] tries to (space) efficiently answer `select` queries: The bitvector is partitioned into *blocks* of size b (by default: $b = 256$) and for each block, the rank at the beginning of the block, modulo b, is stored in an array a. Also, a *nested* Bitvector of $\lceil n/b \rceil$ bits stores a bit for each block, which is 1 if this block contains a 1 whose $\texttt{rank}_1$ is a multiple of b. To efficiently support $\texttt{select}_0$ in addition to $\texttt{select}_1$, a second nested bitvector storing $\texttt{rank}_0$ is necessary.[5] Then, $\texttt{rank}(i)$ can be easily implemented using the nested BV's $\texttt{rank}(\lfloor i/b \rfloor) * b$ and $a[\lfloor i/b \rfloor]$ while $\texttt{select}(i)$ can be implemented using the nested BV's $\texttt{select}(\lfloor i/b \rfloor)$, $\texttt{select}(\lfloor i/b \rfloor + 1)$ and a binary search over the corresponding entries of a. If the values of bits are independently and identically distributed with probability p for a 1, then only $1/p$ entries of a must be looked at on average for $\texttt{select}_1$, making this a $\mathcal{O}(1)$ implementation in that case, provided the nested BV supports constant-time operations. By using the Recursive Bitvector as its own nested Bitvector type, this is easy to guarantee. For $b := \log n$, a contains $\lceil \frac{n}{\log n} \rceil \log\log n$ bits and the nested BV stores $m := \lceil \frac{n}{\log n} \rceil$ logical bits (using $s_{\text{nested}}(n) := m + \lceil \frac{m}{\log m} \rceil \log\log m + s_{\text{nested}}(m)$ bits), or $o(n)$ additional bits in total[6].
- The **Classical Bitvector** is based on the Classical Rank Bitvector but additionally stores `select` metadata in two levels: By storing indices of superblocks (or blocks within a superblock) which contain bits where the `rank` is a multiple of some constant c, binary search over `rank` metadata need only consider a constant interval on average if bits are 'randomly' distributed, similar to the Recursive Bitvector Strategy.

---

[2] It would have been possible to add `select` metadata to make those queries more efficient without any additional space overhead, but there are better implementations in any case

[3] I. e. , $\texttt{select}_0$ and $\texttt{select}_1$ entries are stored in the same array, a trick which also applies to other Bitvectors.

[4] I don't know if this idea already exists because I avoided looking up existing Bitvector implementation, apart from some low-level bit-fiddling tricks.

[5] Nested Bitvectors only need to answer $\texttt{select}_1$ queries, never $\texttt{select}_0$.

[6] For $b = 256$, the nested's nested Bitvector contains only $\lceil \frac{n}{2^{16}} \rceil$ bits, so the Trivial Bitvector can be used there without wasting too much space

1.2 Elias-Fano

The smallest value is subtracted from each value and the universe is taken to be $\mathcal{U}' := [0, x_{\max} - x_{\min}]$ to reduce the size of the upper Bitvector. To find the predecessor in an interval of lower values, it is necessary to use binary search to avoid linear worst-case behavior.

1.3 RMQ

Compared to Elias-Fano, I spent far less time on the RMQ problem. I have implemented 5 different RMQ structures:

- The **Simple RMQ** is simply an unbounded array (*`std::vector`*) that uses binary search.
- The **Naive RMQ** precomputes all queries.
- The $n \log n$ **space RMQ** uses an additional array to store minima indices as shown on the slides.
- The **Linear space RMQ** partitions the list of values into blocks of b (default: $b = 256$) values and builds a $n \log n$ RMQ structure over the block minima. It also uses an array to store the index of the minimum within each block; blocks are partitioned in smaller subblocks, and prefix and suffix minima are stored over subblocks within in a block[7] [8].
- The **Succinct RMQ** is a very unoptimized implementation of the succinct RMQ from [4] using the range min-max tree from [1] to answer rm queries in $\mathcal{O}(\log n)$ time.

2 Implementation

The entire project was written in standard-conforming single-threaded[9] C++17 Test in C++17, test on windows but optionally uses compiler extensions and C++20 features (if available) for optional goals such as full compile-time evaluation of all algorithms and data structures[10], better error messages and warnings, and increased speed by using special instructions (e.g. BMI2 instructions), C++ attributes such as `[[(un)likely]]`, and by informing the compiler that addresses have a high alignment.[11]

3 Experimental Evaluation

I ran all benchmarks on my Laptop computer, a Thinkpad T14 with 32GB RAM and 16 logical cores. For the benchmarks, the laptop was put in a special benchmarking state, which includes disabling turbo boost, disabling dynamic frequency scaling (so all cores are always running at 1,7 Ghz), reserving a logical core for the exclusive use of the benchmarked program and completely disabling its SMT partner. The x86 BMI2 `pdep` instruction, which can be used for `select` on 64bit values, is implemented in microcode on my processor, resulting in

[7] This part could be implemented much more space-efficiently

[8] Because the block size is constant, it's technically not in $\mathcal{O}(n)$ space, but plugging in 2^{64} for n in the equation from the slides ([5]) would only give a block size b' of 16 values, resulting in an even larger $n \log n$ RMQ

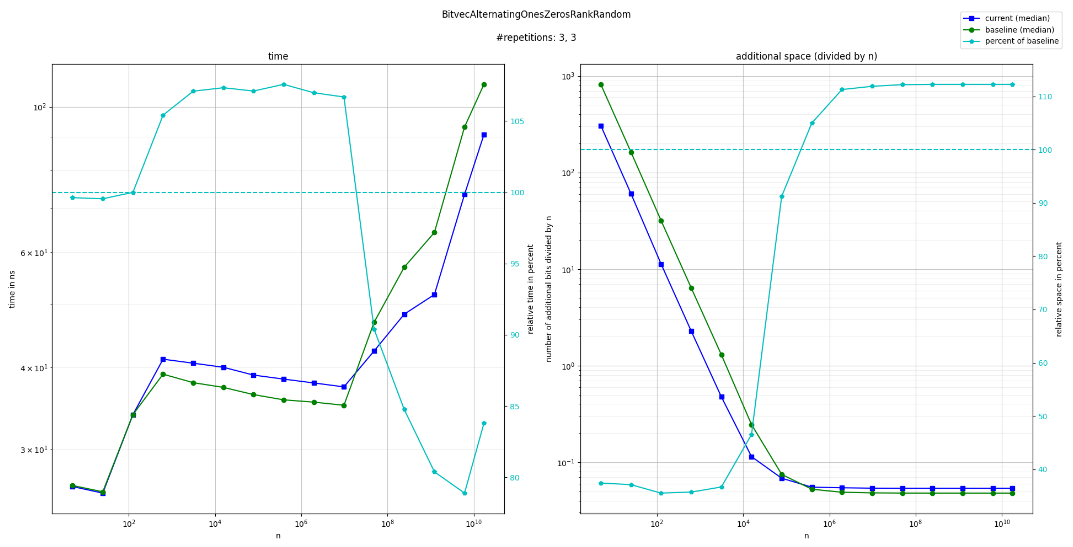
[9] It would have been straightforward to parallelize queries, but that seemed to violate the spirit of the task

[10] This is helpful in C++ because constant evaluation is required to produce an error when executing Undefined Behavior, which allows testing for the absence of UB

[11] To keep this document relatively short, I don't go into details of the individual implementation.

dramatically worse performance compared to the lookup table-based implementation, and therefore isn't used[12].

Figure 1 Comparison of the Classical Bitvector (current) vs. Recursive Bitvector (baseline) for Bitvectors consisting of alternating 0s and 1s, with random rank queries. The Bitvector hyper-parameters used are their default values, which have been chosen to use roughly the same amount of additional space. As with all plots, error bars are drawn when the coefficient of variance out of 3 repetition is greater than 5 percent, where a single repetition executes the benchmark multiple times, as controlled by the google benchmark framework. Cache effects are clearly visible.



References

1 Joshimar Cordova and Gonzalo Navarro. Simple and efficient fully-functional succinct trees, 2016. `arXiv:1601.06939`.

2 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, apr 1974. `doi:10.1145/321812.321820`.

3 Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

4 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. `arXiv:https://doi.org/10.1137/090779759`, `doi:10.1137/090779759`.

5 Florian Kurpicz. Advanced data structures lecture 04: Predecessor and range minimum query data structures, 2023. URL: `https://algo2.iti.kit.edu/download/kurpicz/2023_advanced_data_structures/04_predecessor_rmq_handout.pdf`.

[12] Which makes all `select` calls slightly slower and should be especially unfavorable for the Recursive Bitvector

Figure 2 Bitvector Rank of the same value for each size on a Bitvector with alternating 1s and 0s, Classical Bitvector (current) vs Recursive Bitvector (baseline). It is clear to see that both Bitvectors support this operation in O(1), although there is considerable variation in the time for different sizes (but very little variance within a given size). Based on profiling data, this appears to be due to the differing number of iterations in the final `popcount` loop, but not scaling with the number of these iterations. I am not certain what the reason for this behavior is, it may have been caused by incorrect branch predictions (although `rank` was called many times for each size, so the branch predictor should have been able to learn the number of loop iterations).

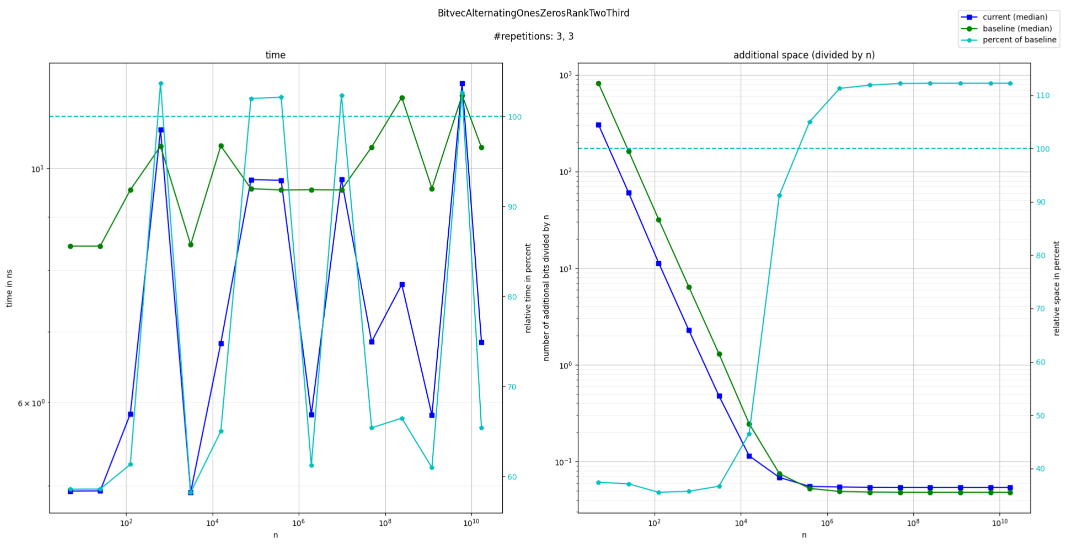


Figure 3 Bitvector `rank` of random indices for random Bitvectors, Classical Bitvector (current) vs Recursive Bitvector (baseline). The variation in rank times has been smoothened by the random queries, but query times are no longer constant as soon as the L3 cache size (4 MiB) is exceeded.

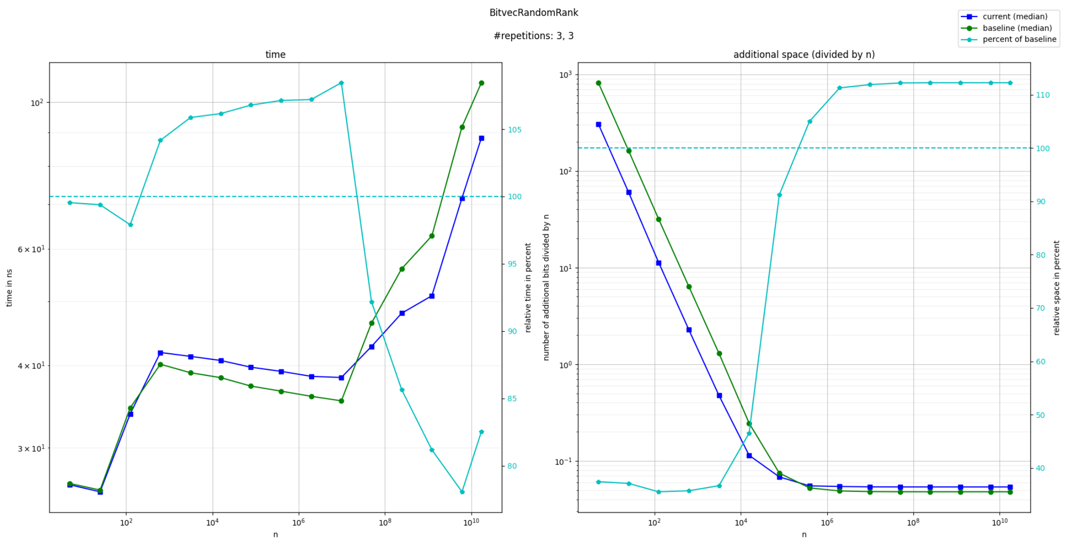


Figure 4 A very similar benchmark to figure 2, but this time with `select` instead of `rank`. Again, it's easy to see that query times are constant, with significant variation for different sizes (but very small measurement noise).

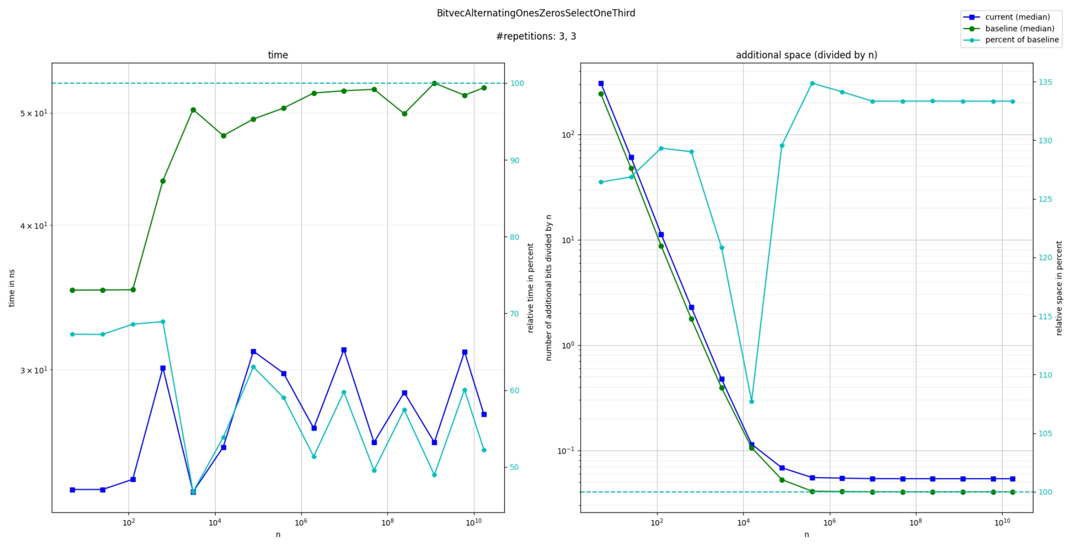


Figure 5 Bitvector `select` of random indices for random Bitvectors, Classical Bitvector (current) vs Recursive Bitvector (baseline). Again, the variation in `select` times has been smoothened by the random queries, but query times are no longer constant due to cache misses and branch mispredictions.

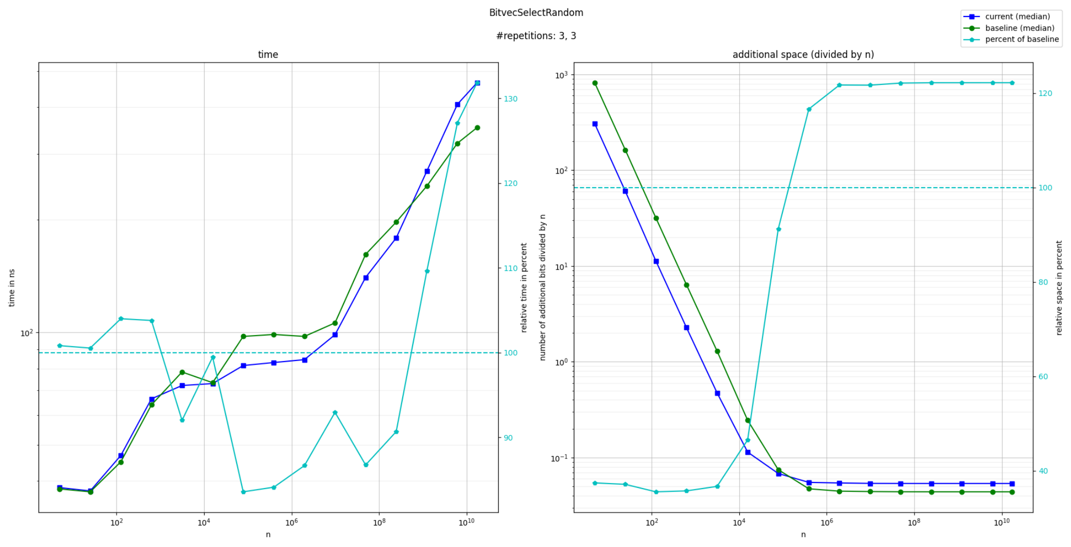


Figure 6 Bitvector `select` of random indices for Bitvectors with random bits; comparison between different Bitvector implementations. The variability in the Trivial Bitvector for large sizes may be explained by the fact that my computer was heavily swapping memory at that point, due to the way the benchmarks are set up internally this can occurs before an individual Bitvector's size exceeds the RAM capacity. Unfortunately, I didn't finish implementing an additional *SelectBitvector* in time, which should have had roughly the same performance characteristic as the Classical Bitvector while using only about 70 percent of its additional space.

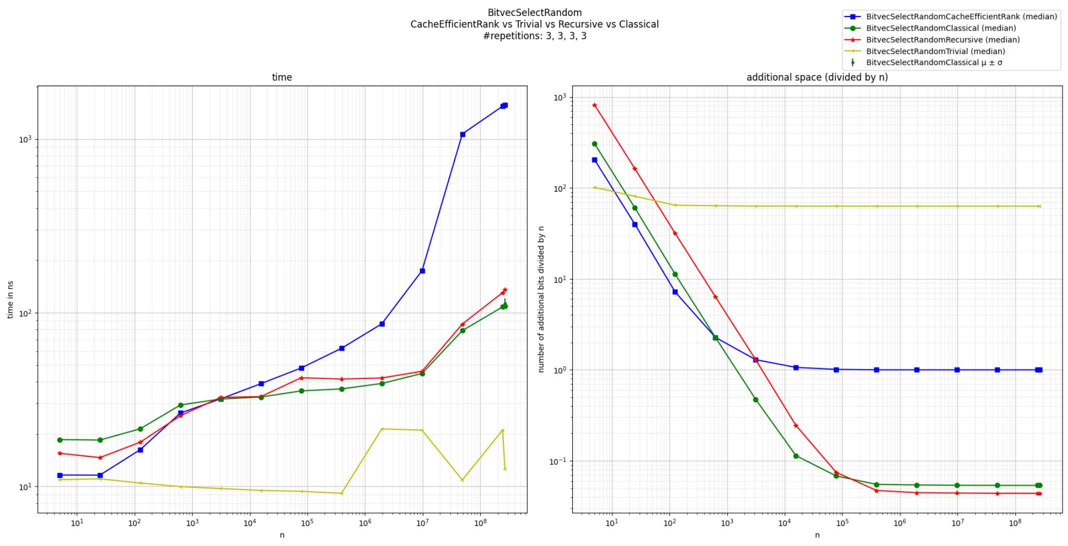


Figure 7 Elias-Fano using the Classical Bitvector containing n consecutive numbers. Queries are randomly chosen from all numbers contained in the Elias-Fano structure. The size of the L3 cache was 4 MiB, enough to store approx.$5 * 10^6$ 64bit values, which is also where the time per query starts to increase due to cache misses. The size of the L1 cache was 32 KiB, enough to store around $4 * 10^3$ values, an effect that can also be seen in the plot.

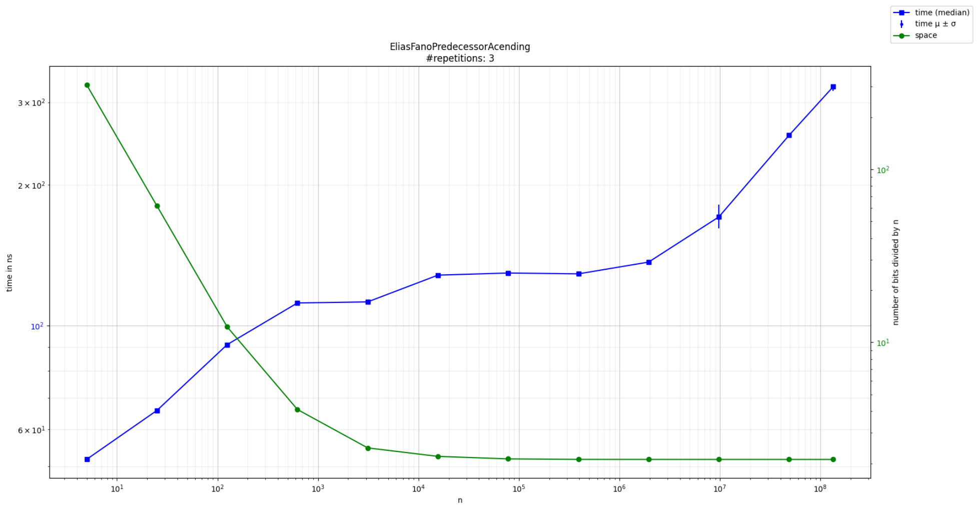
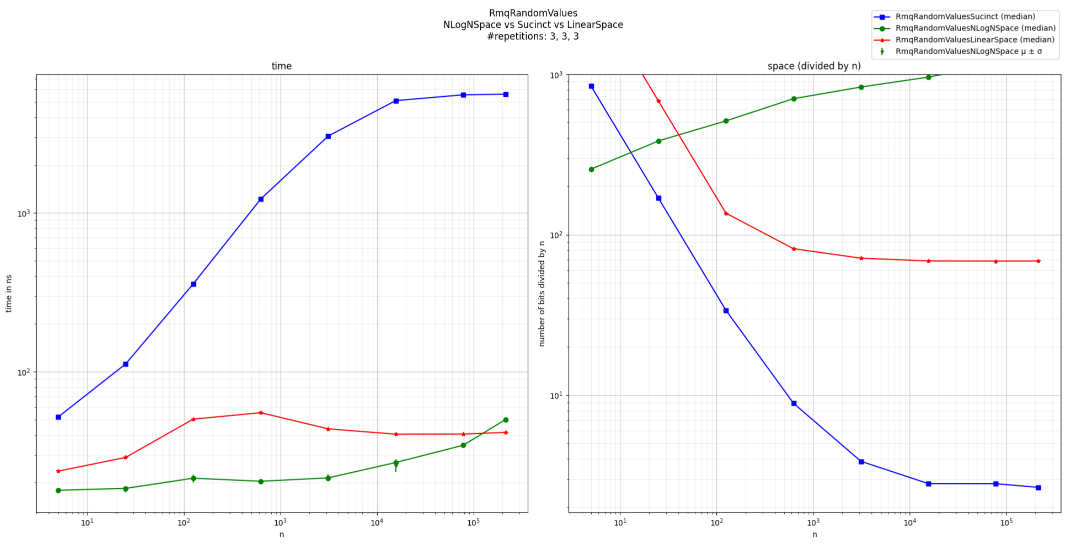


Figure 8 Three different RMQ implementation with random values and random query intervals. All implementations could be optimized further, the Succinct RMQ is slow because it needs $\mathcal{O}(\log n)$ time to answer a query and because the current implementation traverses each bit individually instead of using precomputed tables.


RmqRandomValues
NLogNSpace vs Sucinct vs LinearSpace
#repetitions: 3, 3, 3
RmqRandomValuesSucinct (median)
RmqRandomValuesNLogNSpace (median)
RmqRandomValuesLinearSpace (median)
RmqRandomValuesNLogNSpace μ ± σ
time
time in ns
n
space (divided by n)
number of bits divided by n
n