Indexing and Encoding Arrays for Element Distinctness Queries

Johannes Fischer
Department of Computer Science, Technical University of Dortmund, Germany

Filippo Lari[1]
Department of Computer Science, University of Pisa, Italy

Abstract

We introduce the data structure variant of the well-known element distinctness problem. Given an array of n elements, the goal is to preprocess the array into a data structure that supports queries asking whether all elements within a given query range are distinct. This has applications in text indexing and possibly also in other algorithmic domains.

In the indexing model (where access to the input array is allowed), we design a data structure using $O(\frac{n \log b}{b})$ bits and answering queries in the time needed to solve an online element distinctness instance of size $O(b)$, for any $b \geq 1$. As a concrete instantiation of this, there exists an index that answers queries in $O(\log \log \log n)$ time using $O(\frac{n \log^2(\log \log \log n)}{\log \log \log n})$ bits of additional space.

Moving to the encoding model (where access to the input array is not allowed), we begin by proving an information-theoretic lower bound for the space usage of $2n - O(\log n)$ bits, and then design a matching encoding with $O(1)$ time queries. We then consider the case in which the alphabet size σ is constant. In this setting, the lower bound can be refined to $n \log r_\sigma - 3 \log(\sigma + 2) + O(1)$ bits, where $r_\sigma = 4 \cos^2\left(\frac{\pi}{\sigma+2}\right)$. This lower bound is matched by an encoding with $O(1)$ time queries.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases element distinctness, range queries, lower bounds, succinct data structures

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.9

Funding *Filippo Lari*: Supported by the project "Enhancing Compression and Search Capabilities in the Software Heritage Archive", grant #G-2025-25193 (`https://sloan.org`), funded by the Alfred P. Sloan Foundation.

1 Introduction

Given an array A of n elements drawn from an alphabet Σ, let $A[l, r]$ denote the subarray of A from position l to r. A range query $q_A(l, r, \cdots)$, for some $0 \leq l \leq r < n$ (and possibly additional parameters "$\cdots$"), is any function whose value depends only on the subarray $A[l, r]$ (and on the additional parameters, if any). Prominent examples of range queries include: range minimum queries, defined as $\mathrm{RMQ}_A(l, r) = \mathrm{argmin}_{l \leq i \leq r}\{A[i]\}$, which return the position of the minimum element in $A[l, r]$ when Σ is totally ordered [8]; rank queries, defined as $\textsc{rank}_A(l, r, c) = |\{i \ : \ l \leq i \leq r \wedge A[i] = c\}|$ for some $c \in \Sigma$, which count the occurrences of the symbol c in $A[l, r]$ [14]; and range mode queries, which return the most frequent element in $A[l, r]$ [17]. Many other examples have been studied in the literature [24]. We are interested in the data structure perspective of range queries, i.e., given a static array A and a function q_A (or simply q in the following), the goal is to build a compact data structure $\mathcal{D}(A, q)$ that allows us to answer queries efficiently.

We introduce a new type of query which, despite its fundamental nature, to the best of our knowledge appears not to have been previously studied from the data structure perspective: *element distinctness queries*. Formally, a query $\textsc{All-Distinct}_A(l, r)$ returns TRUE if all elements in $A[l, r]$ are distinct, and FALSE otherwise.

[1] This work was carried out while visiting the Department of Computer Science at the Technical University of Dortmund.

In an algorithmic setting, where the task is to detect whether an array contains any duplicate elements, this is a well-studied problem; see [25] for recent progress in that context.

We argue that it is worth considering this problem in the array query setting. When A is the document array [20] of a text collection, an index returns the suffix array interval $[l, r]$ corresponding to a given pattern P. Asking whether P occurs exactly once in each text then corresponds precisely to a ALL-DISTINCT$_A(l, r)$ query, which could, for example, provide insight into the biological function of a queried DNA fragment.

There are two models of range query problems [6], depending on whether the input array A is available at query time (*indexing model*) or not (*encoding model*). In the indexing model, some space for the data structure $\mathcal{D}$ can potentially be saved, as the query algorithm may compensate for missing information by consulting A when answering the queries. However, since access to A is required at query time, the *total* space usage, which is $|A| + |\mathcal{D}|$ bits, can never be sublinear in the space needed to store A itself.

This contrasts with the *encoding model*, where the data structure $\mathcal{D}$ must be constructed so that queries can be answered *without* consulting A. Any encoding data structure $\mathcal{D}_\mathrm{E}$ is automatically also an indexing data structure. Conversely, an indexing data structure $\mathcal{D}_\mathrm{I}$ can always be converted to an encoding data structure by storing $\mathcal{D}_\mathrm{I}$ together with a copy of A. Consequently, distinguishing between the two models is meaningful only if there exist indexing data structures that use less space than the best encoding data structures, and if encoding data structures can achieve space usage sublinear in $|A|$.

As for range minimum queries [8], we show that this is indeed the case for ALL-DISTINCT-queries. Specifically, in Sect. 2 we prove the following result, which, for the sake of generality, is stated in terms of parameters $T_{dist}(b, \sigma)$ and $S_{dist}(b, \sigma)$ denoting the time and space complexity of online algorithms for the element distinctness problem:

▶ **Theorem 1.** *Let A be an array of n elements drawn from a totally ordered set of size $\sigma \leq n$ and $b \geq 1$. Assuming that the element distinctness problem on an $O(b)$-sized instance can be solved in $T_{dist}(b, \sigma)$ time using $S_{dist}(b, \sigma)$ bits of additional space, there exists an indexing data structure using $\frac{2n}{b}(2 + \log b) + o(n/b)$ bits of space and answering ALL-DISTINCT queries in $T_{dist}(b, \sigma)$ time and $S_{dist}(b, \sigma)$ bits of working space. The data structure can be constructed in $O(n)$ time and $O(\sigma \log n)$ bits of additional working space.*

Let us give two concrete instantiations of the theorem for the choice of $b = \Theta(\log n)$, which yields an index using $O(n \log\log n / \log n) = o(n)$ bits of space. First, assuming that $\sigma = O(n \log\log n / \log^2 n)$, we can solve the element distinctness problem using *counting sort* in $T_{dist}(b, \sigma) = O(\log n)$ time and $S_{dist}(b, \sigma) = O(n \log\log n / \log n)$ bits of working space, which does not exceed the space of the index. Second, without any restriction on σ, one can use comparison based sorting to solve element distinctness queries in $T_{dist}(b, \sigma) = O(\log n \log\log n)$ time using only a negligible amount $S_{dist}(b, \sigma) = O(\log^2 n)$ bits of working space. We refer to [25] and references therein for other space-time trade-offs for the element distinctness problem. Using comparison sort, we can actually set the block size to any $b = \omega(1)$ to achieve sublinear space and $O(b \log b)$ query time. For instance, with $b = \frac{\log^{[k]} n}{\log^{[k+1]} n}$ for a positive constant integer k, where $\log^{[k]} n$ denotes k iterations of the logarithm, the space of the index is $O(\frac{n(\log^{[k+1]} n)^2}{\log^{[k]} n}) = o(n)$ bits, and the query time is $O(\log^{[k]} n)$.

Moving to the encoding model, in Sect. 3, we begin by showing the following result:

▶ **Theorem 2.** *Let A be an array of n elements drawn from a totally ordered set of size $\sigma \leq n$. There exists an encoding data structure answering ALL-DISTINCT queries in $O(1)$ time while using $2n + O(\frac{n \log\log n}{\log n})$ bits of space, which is asymptotically optimal. The data structure can be constructed in $O(n)$ time and $O(\sigma \log n)$ bits of additional working space.*

We then consider restricted alphabet sizes, where we refine Thm. 2 as follows:

▶ **Theorem 3.** *Let $r_\sigma = 4\cos^2(\frac{\pi}{\sigma+2})$, and A be an array of n elements drawn from a totally ordered set of constant size $\sigma < n$. There exists an encoding data structure using $n \log r_\sigma + O(\frac{n}{\log n})$ bits of space and answering* All-Distinct *queries in $O(1)$ time, which is asymptotically optimal. The data structure can be constructed in $O(n)$ time using $o(n)$ bits of working space.*

Interpreting the term $\log r_\sigma$ as a function of σ, we observe that $1/4 \leq \cos^2(\frac{\pi}{\sigma+2}) < 1$, therefore $0 \leq \log(4\cos^2(\frac{\pi}{\sigma+2})) = \log r_\sigma < 2$, and hence the leading term in the space usage of Thm. 3 is never larger than $2n$ bits. Some special cases for small alphabet sizes ($\sigma = 2, 3, 4$) can be computed explicitly as n, $2n\log((1+\sqrt{5})/2) = 1.388n$, and $n\log 3 = 1.585n$, respectively. Interestingly, these are the same values for range minimum queries on restricted alphabets [16].

Finally, we note that all our results hold in the *transdichotomous word-RAM model*, where the word size w satisfies $w = \Theta(\log n)$, and arithmetic as well as bitwise operations on w bits are performed in constant time. Additionally, our results also require the alphabet to be *linearly sortable* (e.g., characters drawn from a universe of size $n^{O(1)}$). This assumption is necessary to compactify the alphabet to $[1, \sigma]$ in linear time; otherwise, we would violate the natural $O(n \log n)$ lower bound for duplicate detection in *in the comparison model* [18].

2 Indexing

In the indexing version of the All-Distinct problem, the data structure is allowed to access the input array while answering queries. In this section, we design an index whose space usage and query time can be tuned based on an input parameter $b \geq 1$. To this end, we first introduce two sequences that form the basis of our solution:

▶ **Definition 4.** *Let A be an array of n elements drawn from a totally ordered set of size σ. For every $0 \leq i < n$, let $P_A[i] = \min\{\, j \leq i \mid$ All-Distinct$_A(j, i) =$ true$\}$ and $N_A[i] = \max\{\, j \geq i \mid$ All-Distinct$_A(i, j) =$ true$\}$.*

Both P_A and N_A are monotone, with P_A being non-decreasing and N_A being non-increasing. Moreover, by the pigeonhole principle, any range of length greater than σ must contain at least one duplicate. It follows that for each $0 \leq i < n$, we have $\max\{0, i-\sigma+1\} \leq P_A[i] \leq i$ and similarly $i \leq N_A[i] \leq \min\{n-1, i+\sigma-1\}$.

Constructing either of these two sequences can be easily done in $O(n)$ time and $O(\sigma \log n)$ bits of space. We describe the procedure for P_A, but the same idea applies to the construction of N_A. We compute $P_A[i]$ for all $0 \leq i < n$ in increasing order, starting with $P_A[0] = 0$. We maintain an auxiliary array $C[0, \sigma-1]$, initialized with negative values, where $C[k]$ stores the index of the last occurrence of symbol k in the current range $[i, j]$. When computing $P_A[i]$, we can inductively assume that $P_A[i-1]$ has already been computed. If $C[A[i]] < P_A[i-1]$, we set $P_A[i]$ to $P_A[i-1]$, as the new symbol $A[i]$ does not induce a duplicate in $A[P_A[i-1], i]$. Otherwise, we set $P_A[i]$ to $C[A[i]] + 1$, which is the minimal index j such that $A[i]$ is not repeated in $A[j, i]$. In all cases, we update $C[A[j]]$ to j.

Having access to either P_A or N_A is already sufficient to answer any All-Distinct(l, r) query in $O(1)$ time: because of Def. 4, the answer is false if $P_A[r] > l$, and true otherwise (equivalently, one may check whether $N_A[l] < r$). However, even though P_A and N_A can certainly be stored in fewer than $n\lceil \log n \rceil$ bits, it is impossible to use less than (roughly) a linear number of bits without having access to A (see Sect. 3 for details). Therefore, we proceed to show how this can be achieved in the indexing setting.

Our index is obtained by first partitioning the input array A into $\lceil n/b \rceil$ blocks of size $1 \leq b < n$. For each $0 \leq i < \lceil n/b \rceil$, we sample the array P_A at position $(i+1)b - 1$, i.e., the last element of the i-th block, and store the resulting value in a separate array S_{P_A}. Similarly, for the same values of i, we sample the array N_A at position ib, i.e., the first element of the i-th block, storing these values in another array S_{N_A}. After collecting these samples, the full arrays P_A and N_A are discarded, and only their sampled versions are used to answer queries.

S_{P_A} and S_{N_A} both require $\frac{n}{b}\lceil \log n \rceil$ bits each if stored plainly, but we can save space using ideas similar to Elias-Fano codes [21, §3.4.3]: Let us now focus on S_{P_A}, as the same considerations apply to S_{N_A}. For each $0 \leq i < \lceil n/b \rceil$, we compute the integer division $S_{P_A}[i]/b$, and store the quotient in $Q_{P_A}[i]$, and the remainder in $R_{P_A}[i]$. Q_{P_A} is again monotonically increasing, with each $0 \leq Q_{P_A}[i] \leq n/b$, hence by encoding the differences between consecutive elements in negated unary and augmenting the resulting binary sequence with $\textsc{select}_1$ support [15, 13] we can access each element of Q_{P_A} in $O(1)$ time while using $\frac{2n}{b} + o(\frac{n}{b})$ bits of space. R_{P_A} is stored plainly using $\frac{n}{b}\lceil \log b \rceil$ bits, and supports $O(1)$ time access. Each $S_{P_A}[i]$ can then be reconstructed in $O(1)$ time as $b \cdot Q_{P_A}[i] + R_{P_A}[i]$. Overall, this encoding for both S_{P_A} and S_{N_A} requires $\frac{2n}{b}(2 + \log b) + o(\frac{n}{b})$ bits of space.

Answering queries with our index is straightforward. Given a query $\textsc{All-Distinct}(l, r)$, we first split the range $[l, r]$ into three parts: $[l, l' - 1]$, $[l', r']$, and $[r' + 1, r]$, where $[l', r']$ is the maximal subrange of $[l, r]$ that perfectly aligns with full blocks of size b in A. Using the same idea as for answering queries with either P_A or N_A, we check in constant time whether the aligned portion $[l', r']$ contains any duplicates by verifying if $S_{P_A}[r'/b] > l$ or $S_{N_A}[l'/b] < r$. If this test fails, the answer to the query is immediately $\textsc{false}$. However, this is insufficient to guarantee uniqueness over the entire range $[l, r]$. Indeed, an element in $A[l, l' - 1]$ may also appear in $A[r' + 1, r]$, and such duplicates are not yet detected by simply verifying the previous condition. Therefore, $A[l, l' - 1] \cup A[r' + 1, r]$ must be checked for duplicates, which is an $O(b)$-sized instance of the element distinctness problem.

Construction. The data structure can be easily constructed in $O(n)$ time by running the previously sketched algorithm only on the sampled positions of S_{P_A}, and directly storing the values in Q_{P_A} (via bit vector appends) and R_{P_A} (via simple array writes). Augmenting Q_{P_A} with support for $\textsc{select}_1$ operations can be done in-place in time linear in the size of the bit vector. Symmetrically, the same procedure applies to all arrays based on N_A, which completes the proof of Thm. 1.

3 Encoding

We now consider $\textsc{All-Distinct}$ queries in the encoding model, in which the underlying array A cannot be accessed at query time. We first consider the general case, developing upper and lower bounds that are independent of the alphabet size σ (Thm. 2). We then focus on restricted alphabets, where tighter bounds can be obtained (Thm. 3).

3.1 Unbounded Alphabet

The following easy lemma forms the basis of the lower bound in Thm. 2.

▶ **Lemma 5.** *Let A and B be two arrays of length n, each drawn from a totally ordered set of size σ. For all $0 \leq l \leq r < n$, we have $\textsc{All-Distinct}_A(l, r) = \textsc{All-Distinct}_B(l, r)$ if and only if $P_A = P_B$.*

Proof. Let us assume that $A \neq B$, otherwise the lemma immediately follows. We prove the two implications by contradiction. ($\Longrightarrow$) Suppose that $\text{ALL-DISTINCT}_A(l,r) = \text{ALL-DISTINCT}_B(l,r)$ for all $0 \leq l \leq r < n$, but $P_A \neq P_B$. Let j be the smallest index such that $P_A[j] \neq P_B[j]$, and assume without loss of generality that $P_A[j] > P_B[j]$ (the other case is symmetric). Then $\text{ALL-DISTINCT}_B(P_B[j], j) = \text{TRUE}$ while $\text{ALL-DISTINCT}_A(P_B[j], j) = \text{FALSE}$, which is a contradiction. ($\Longleftarrow$) Suppose that $P_A = P_B$, but there exists a range $[l,r]$ such that $\text{ALL-DISTINCT}_A(l,r) \neq \text{ALL-DISTINCT}_B(l,r)$. Without loss of generality, assume $\text{ALL-DISTINCT}_A(l,r) = \text{FALSE}$. This implies that there exists a $j > l$ such that $P_A[r] = j$. Since $\text{ALL-DISTINCT}_B(l,r) = \text{TRUE}$, it must be that $P_B[r] \leq l$, which in turn means $P_A[r] \neq P_B[r]$, a contradiction. ◀

Lemma 5 implies that the number of equivalence classes of the ALL-DISTINCT problem corresponds to the number of possible P-arrays as in Def. 4. From a simple information-theoretic argument, taking the logarithm of such a quantity yields the lower bound of Thm. 2. To perform such a counting, we begin by introducing the following definition:

▶ **Definition 6.** *The delta sequence Δ_{P_A} of P_A is defined by setting $\Delta_{P_A}[0] = 0$ and, for each $1 \leq i < n$, $\Delta_{P_A}[i] = P_A[i] - P_A[i-1]$.*

Notice that, because of Def. 4, $0 \leq \Delta_{P_A}[i] \leq \sigma$ for each $0 \leq i < n$. The following result is already formulated more generally to make it also applicable to restricted alphabet sizes (in this section, it should be interpreted with $\sigma = n$):

▶ **Lemma 7.** *The set of P_A arrays computed from any array A of n elements drawn from a totally ordered set of size $\sigma \leq n$ is in bijection with the set of ordered trees with $n+1$ nodes and height at most $\sigma + 1$.*

Proof. We prove this by giving a bijection with balanced parenthesis sequences of length $2(n+1)$ having maximum excess $\sigma + 1$, which in turn gives the desired bijection. Given P, we construct the sequence B as follows: Initially, B is empty. Then, for all $0 \leq i < n$ in increasing order, append $\Delta_{P_A}[i]$ closed parentheses and a single open parenthesis to B. Finish by appending $(n-1) - P[n-1] + 1$ closed parentheses and enclose everything in another pair of open and closed parentheses.

It is easy to see that B has length $2(n+1)$ and is balanced. The latter follows because, for any position $1 \leq i < 2(n+1)$, if we denote by k the step in which its parenthesis is generated, then $\text{EXCESS}(i) \geq k + 1 - P[k+1] \geq 0$ (there are $k+1$ open parentheses before i), and $\text{EXCESS}(2(n+1)) = 0$. Using a similar argument, we also obtain that, for any position i, $\text{EXCESS}(i) \leq k + 2 - P[k]$, which is at most $k + 2 - \max\{0, k - \sigma + 1\}$ by Def. 4. If $k \leq \sigma$, then $\text{EXCESS}(i) \leq k + 1 \leq \sigma + 1$. Otherwise, if $k > \sigma$, then $\text{EXCESS}(i) \leq k + 2 - k + \sigma - 1 = \sigma + 1$.

It remains to prove that our mapping is bijective. Surjectivity follows easily, since any sequence of balanced parentheses allows us to construct Δ_{P_A} and hence P_A. Injectivity can be proved by contradiction, in a way similar to the proof of Lemma 5. ◀

From Lemma 7, the number of different P_A-arrays for $\sigma = n$ is given by the n-th Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$, which implies the following:

▶ **Lemma 8.** *Let A be an array of n elements drawn from a totally ordered set of size n. Any encoding data structure for A supporting ALL-DISTINCT queries must use at least $2n - O(\log n)$ bits of space.*

We are now left with designing a data structure that uses space close to the lower bound of Lemma 8 while still providing efficient queries. As already mentioned in Sect. 2, if we have access to P_A, answering any $\text{ALL-DISTINCT}_A(l,r)$ query reduces to checking whether

$P_A[r] > l$. Since P_A is a monotone sequence satisfying $P_A[i] \leq i$ for all $0 \leq i < n$, we can store it succinctly by encoding the Δ_{P_A}-values in negated unary encoding in a bit vector G, i.e., $G = 0^{\Delta_{P_A}[0]}10^{\Delta_{P_A}[1]}1 \dots 0^{\Delta_{P_A}[n-1]}1$. G has n ones and $\sum_{i=0}^{n-1} \Delta_{P_A}[i] = P_A[n-1]$ zeros, and since by Def. 4, $P_A[n-1] \leq n-1$, the overall size of G is at most $2n$ bits. We conclude by noticing that $P_A[i] = \text{SELECT}_1(G, i+1) - i$, and that the SELECT_1 operation can be supported in $O(1)$ time by only adding $O(\frac{n \log\log n}{\log n})$ bits on top of G [15, 13].

Construction. G can be constructed directly using the algorithm from Sect. 2, and augmenting it with SELECT_1-support can be done in-place. Overall, we proved Thm. 2.

3.2 Restricted Alphabet

In this section, we consider the All-Distinct problem under a restricted alphabet (i.e., $\sigma < n$). We first prove a lower bound that explicitly depends on σ, providing a refinement of Lemma 8 for this setting. We then complement this result by designing an encoding whose space usage approaches the lower bound while still supporting $O(1)$ query time.

For the special cases $\sigma = 2$ or $\sigma = 3$, the lower bound of Lemma 8 can be surpassed. It suffices to store the input array A directly (after compactifying the alphabet), using n bits for $\sigma = 2$ or $n \log 3 \approx 1.585n$ bits for $\sigma = 3$. Queries of length at most 2 (or 3, respectively) can then be answered by scanning the corresponding elements of A, while longer queries must necessarily contain a duplicate. However, already for $\sigma = 4$, this simple approach cannot use fewer than $2n$ bits. Nevertheless, a tighter bound can still be achieved.

3.2.1 Lower Bound

▶ **Lemma 9.** *Let A be an array of n elements drawn from a totally ordered set of constant size $\sigma < n$. For large enough n, any encoding data structure for A supporting All-Distinct queries must use at least $n \log r_\sigma - 3 \log(\sigma+2) + O(1)$ bits of space, where $r_\sigma = 4\cos^2\left(\frac{\pi}{\sigma+2}\right)$.*

Proof. Exploiting Lemmas 7 and 5, the number of equivalence classes for the All-Distinct problem on a restricted alphabet $\sigma < n$ is given by the number of ordered trees having $n+1$ nodes and maximum height at most $\sigma+1$. Let us denote with $C_{n+1}^{\sigma+1}$ this number. De Bruijn et al. [3] already studied this combinatorial problem, giving the following asymptotic equivalence for any constant σ:

$$C_{n+1}^{\sigma+1} \sim \frac{4^{n+1}}{\sigma+2} \cos^{2(n+1)}\left(\frac{\pi}{\sigma+2}\right) \tan^2\left(\frac{\pi}{\sigma+2}\right)$$

The asymptotic equivalence $f \sim g$ means $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$. In our case, this implies that, for large enough n, $C_{n+1}^{\sigma+1} = \Theta\left(\frac{4^{n+1}}{\sigma+2} \cos^{2(n+1)}\left(\frac{\pi}{\sigma+2}\right) \tan^2\left(\frac{\pi}{\sigma+2}\right)\right)$ with an hidden constant c arbitrarily close to 1. Observing that $\frac{\pi}{\sigma+2} \in (0, \frac{\pi}{2})$ for all $\sigma \geq 1$, and recalling that $\tan x > x$ on $(0, \frac{\pi}{2})$, we have $\tan\left(\frac{\pi}{\sigma+2}\right) > \frac{\pi}{\sigma+2}$. Using this inequality, and letting $r_\sigma = 4\cos^2\left(\frac{\pi}{\sigma+2}\right)$, we obtain the following lower bound on $\log C_{n+1}^{\sigma+1}$, from which our claim follows:

$$\begin{aligned}
\log C_{n+1}^{\sigma+1} &= n \log r_\sigma + 2 \log\left(\tan\left(\frac{\pi}{\sigma+2}\right)\right) - \log(\sigma+2) + O(1) \\
&> n \log r_\sigma + 2 \log\left(\frac{\pi}{\sigma+2}\right) - \log(\sigma+2) + O(1) \\
&= n \log r_\sigma - 3 \log(\sigma+2) + O(1)
\end{aligned}$$

◀

3.2.2 Upper Bound

We show how to represent P_A in space as close as possible to the lower bound of Lemma 9, while still allowing arbitrary entries to be retrieved in constant time. All-Distinct queries are then answered as in Sect. 2, hence checking whether $P_A[r] > l$. Our approach follows the idea of the $2n$-bit *indexing* data structure for $O(1)$ range minimum queries [8]. Specifically, we partition the array Δ_{P_A} into sufficiently small blocks, assign a *type* to each block that uniquely identifies it among all possible blocks, and use a universal lookup table that stores precomputed answers to queries within a block. A running example is provided in Fig. 1.

Notice that, due to the bijection with ordered trees of bounded height of Lemma 7, a natural approach to proving Thm. 3 would be based on tree covering [4] or, possibly, on hypersuccinct trees [19], in the spirit of [16]. In Appendix B, we sketch such an alternative. However, its space has a larger second-order term than Thm. 3, and, more importantly, we are not aware of a space-efficient construction algorithm analogous to that in Thm. 3.

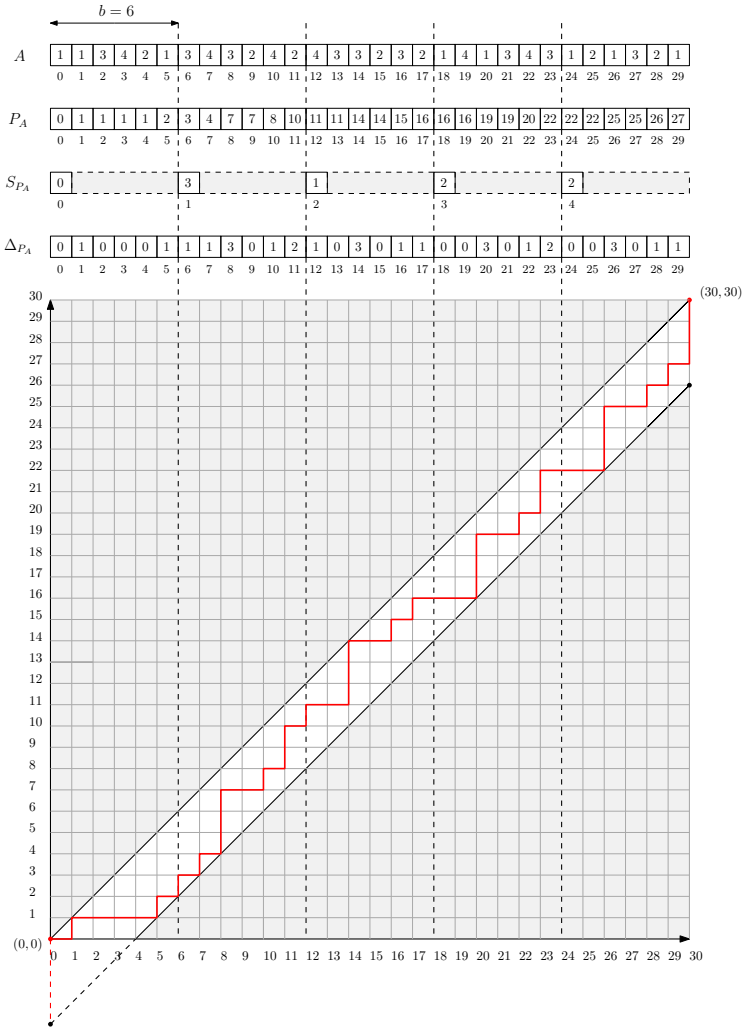


Figure 1 Example of the main components of our encoding data structure for an instance of size $n = 30$ over an alphabet of size $\sigma = 4$.

Given an array A, we partition P_A into consecutive blocks of $b \geq 1$ elements each. For every such block, we sample its first value and store it relative to its position in a separate array S_{P_A} (i.e., for the j-th block $S_{P_A}[j] = j \cdot b - P_A[j \cdot b]$, which is always bounded by $\sigma - 1$). The key idea is that, once the first value of a block is known, all the P_A values within that block can be reconstructed from the corresponding portion of the delta sequence Δ_{P_A}, without using P_A. Indeed, for each position $0 \leq i < n$, if we let $j = \lfloor i/b \rfloor$ be its corresponding block and $k = j \cdot b$ its starting position in P_A, then:

$$P_A[i] = j \cdot b - S_{P_A}[j] + \sum_{z=k+1}^{i} \Delta_{P_A}[z] \tag{1}$$

Notice that, because of the samples, the first Δ_{P_A} value of each block is not used when reconstructing an entry of P_A. In Fig. 1, to recover the value of $P_A[20]$ in position $i = 20$, we access the sample array S_{P_A} in position $j = \lfloor 20/6 \rfloor = 3$, then, using Δ_{P_A}, we compute $P_A[20] = 3 \cdot 6 - S_{P_A}[2] + \Delta_{P_A}[19] + \Delta_{P_A}[20] = 18 - 2 + 0 + 3 = 19$.

Therefore, to efficiently compute Eq. 1, we use the following precomputed tables:

- T, which stores, for each block of size b, an integer identifying its *type*. For all $0 \leq i, j < \lceil n/b \rceil$, we have $T[i] = T[j]$ if and only if $\Delta_{P_A}[i \cdot b + k] = \Delta_{P_A}[j \cdot b + k]$ for all $1 \leq k < b$.
- L, which stores, for each possible block type t and every position j within such a block, in $L[t][j]$ the sum of deltas up to that position starting from the second element of the block, with the first entries $L[t][0]$ always set to zero.

Having access to those tables, Eq. 1 can be computed in $O(1)$ time as follows:

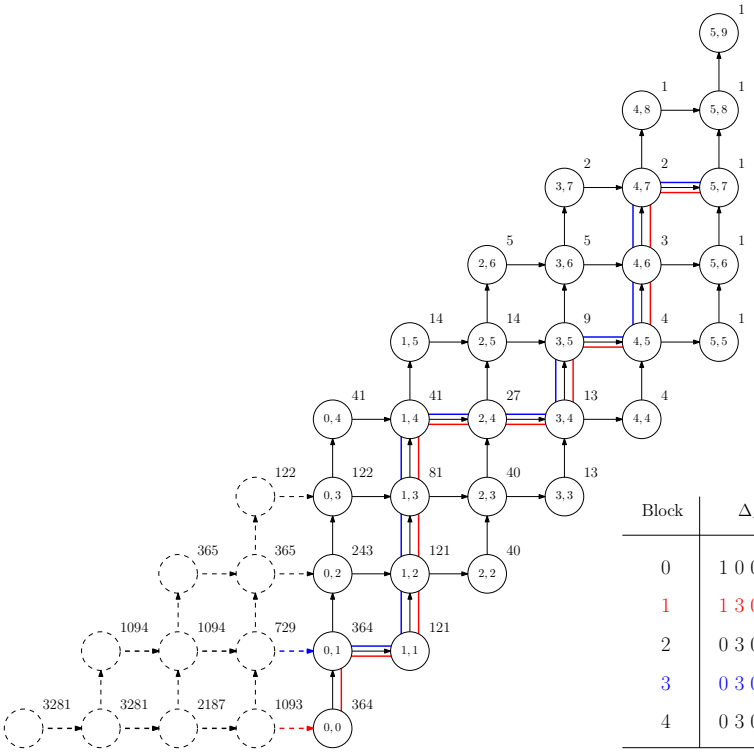
$$P_A[i] = (j \cdot b - S_{P_A}[j]) + L[T[j]][i \bmod b] \ , \tag{2}$$

with $j = \lfloor i/b \rfloor$ being the block number of i. It remains to show how to assign types to blocks based on their Δ_{P_A} sequence, and that the combined space usage of T, L, and S_{P_A} approaches that of Lemma 9.

3.2.2.1 Computing the Block Types

We begin by describing an algorithm for assigning types to blocks. To this end, it is convenient to interpret the delta sequence Δ_{P_A} as a two-dimensional lattice path in the Cartesian plane going from $(0, 0)$ to (n, n) and consisting only of *East* and *North* steps (i.e., $(1, 0)$ and $(0, 1)$). Such a path can be obtained using the same idea we exploited to prove the bijection with balanced parenthesis sequences (see Lemma 7). For $0 \leq i < n$ in increasing order, perform $\Delta_{P_A}[i]$ North steps followed by a single East step. Terminate by appending $(n - 1) - P_A[n - 1] + 1$ North steps. Notice that, because of Def. 4, the number of North steps is never more than the number of East steps. Consequently, the path never goes above the main diagonal and therefore corresponds to a *Dyck path* [10, p.76]. Moreover, the vertical distance between the path and the main diagonal (i.e., the difference between the number of East and North steps) is bounded by σ. See Fig. 1 as an example.

Each block of size b in Δ_{P_A} naturally corresponds to a Dyck path fragment confined within a parallelogram of base b and height $\sigma + 1$. As an example, see the region delimited by the points $(6, 2)$, $(6, 6)$, $(12, 8)$, and $(12, 12)$ in Fig. 1, corresponding to the second block of Δ_{P_A} in the example. This holds for every block except the first one, but we can conceptually regard it as having a parallelogram extending below the x-axis, with its path starting at its leftmost top vertex and always beginning with σ North steps. Notice, however, that due to Eq. 1, we are only concerned with paths of length $b - 1$, since the first Δ_{P_A} value in each block is not used.



Block	Δ_{P_A}	Entry	Type
0	1 0 0 0 1	3	$(3)_2 \cdot 0$
1	1 3 0 1 2	0	$(0)_2 \cdot 86$
2	0 3 0 1 1	1	$(1)_2 \cdot 113$
3	0 3 0 1 2	1	$(1)_2 \cdot 86$
4	0 3 0 1 1	1	$(1)_2 \cdot 113$

Figure 2 Left: the extended graph G for $\sigma = 4$ and $b - 1 = 5$. Right: the types of the blocks from the example in Fig. 1, with two example blocks (red/blue) whose normalized paths are also shown in G. Notice that, despite having different Δ_{P_A}-sequences, without appending the binary encoding of their entry height, they would end up with the same code.

Exploiting this reduction to Dyck paths, we are now left with the problem of assigning the same identifier to blocks corresponding to the same path. To this end, we begin by defining the following (conceptual) graph G. See Fig. 2 for an example with $b = 6$ and $\sigma = 4$; ignore the dotted part in the lower left corner.

▶ **Definition 10.** *Graph G has nodes (p, q) for all $0 \leq p \leq b-1$ and all $p \leq q \leq p+\sigma$. Each node (p, q) with $p \leq q$ has an outgoing edge to $(p, q+1)$ if $q+1-p \leq \sigma$, and an outgoing edge to $(p+1, q)$ if $p+1 \leq q$.*

We then assign to each node (p, q) a value representing the number of distinct paths starting at (p, q) and reaching the terminal node $(b-1, b+\sigma-1)$ (i.e., the only node without outgoing edges, shown as the top rightmost vertex in Fig. 2). These values are stored in a table $N[0, b-1][0, b+\sigma-1]$ of $b \cdot (b+\sigma)$ elements, such that $N[p][q]$ stores the number of paths from node (p, q) to $(b-1, b+\sigma-1)$, with 0's in cells not corresponding to any node. By exploiting the fact that there are at most two options for the next step from (p, q) (either first going North or first going East), the following formula allows us to precompute the N-array in $O(b\,(b+\sigma))$ time:

$$N[p][q] = \begin{cases} 1 & \text{if } p = b-1 \text{ and } p \leq q \leq b+\sigma-1 \\ 0 & \text{if } p > q \text{ or } q > \sigma + p \\ N[p+1][q] + N[p][q+1] & \text{otherwise} \end{cases} \tag{3}$$

Algorithm 1 Computing all block types.

Input: The sequences P_A and Δ_{P_A}, table N, block size b, alphabet size σ
Output: The type table T

```
s ← σ ;                                   // Initial entry height of the first block
for j ← 0 to ⌈n/b⌉ − 1 do
    h ← min_{jb<i<(j+1)b} {P_A[i] − i + σ − 1};          // distance to lower diagonal
    s ← (s + Δ_{P_A}[j · b] − 1) − h;     // skip first element and normalize path
    code ← Σ_{q=1}^{s−1} N[1][q];
    (p, q) ← (1, s);
    for k ← 1 to b − 1 do
        for h ← 1 to Δ_{P_A}[j · b + k] do
            if p + 1 ≤ q then
                code ← code + N[p + 1][q] ;                  // Add missed paths
            end
            q ← q + 1 ;                                              // North step
        end
        p ← p + 1 ;                                                   // East step
    end
    T[j] ← concatenation of s (using ⌈log σ⌉ bits) and code;
    s ← s + q − p;
end
return T;
```

In Fig. 2, the non-zero N-values are shown next to each node in G. Note that the numbers $C_{\circ}^{\sigma+1}$ from Sect. 3.2.1 appear on the upper diagonal of G.

We proceed by describing an algorithm for computing the type of the j-th block, that is, for the sequence of deltas $\Delta_{P_A}[j \cdot b + 1], \Delta_{P_A}[j \cdot b + 2], \ldots, \Delta_{P_A}[(j+1) \cdot b - 1]$ (remember that the first delta of each block is ignored). As before, this sequence of deltas can be seen as a path in G, starting at $(0, s)$ for some height $0 \leq s < \sigma$, and ending at $(b-1, t)$ for some t. We begin by *normalizing* the path by shifting it downward as much as possible, by simply computing the minimum vertical distance between the path and the lower boundary, subtracting this value h from every point along the path. This prevents assigning different codes to paths having the same shape but different starting heights.

To ensure each path starts at $(0, 0)$, we modify it by prepending s North steps. We then assign the codes as follows: initialize a variable *code* to 0, and each time the path moves from (p, q) to $(p, q+1)$ via a North step, increment *code* by the number of paths that would be skipped if the path were to take an East step to $(p+1, q)$; that is, by $N[p+1][q]$ if such a node exists, or 0 otherwise. Lastly, we note that, following this numbering scheme, paths starting from different nodes may be assigned the same code. As an example, see the paths corresponding to the second and fourth blocks in Fig. 2, which would both get code 86, although the delta sequence of the blocks is different. To still ensure a unique code for different blocks, we prepend the binary encoding of the starting height s to *code* using $\lceil \log \sigma \rceil$ bits to obtain the final block type. See Alg. 1 for a complete, formal description of our block type assignment procedure, which runs in $O(n)$ time, since all paths together contain at most $2n$ North and East steps.

It remains to prove that the algorithm we just described is a valid numbering scheme, that is, it never assigns the same code to two blocks having the same starting height and representing different normalized Dyck path fragments. In the following, we sketch the main idea of the proof and defer the complete formal argument to Appendix A.

▶ **Lemma 11.** *Let T be the table produced by Alg. 1. For all indices $0 \leq i \leq j < \lceil n/b \rceil$, if $T[i] = T[j]$ then for all $1 \leq k < b$, $\Delta_{P_A}[i \cdot b + k] = \Delta_{P_A}[j \cdot b + k]$.*

Proof Sketch. We proceed by contradiction. Suppose there exist $0 \leq i < j < \lceil n/b \rceil$ such that $T[i] = T[j]$, but for some $1 \leq k < b$, $\Delta_{P_A}[i \cdot b + k] \neq \Delta_{P_A}[j \cdot b + k]$. This difference then translates to a difference in the corresponding Dyck path fragments of the two blocks.

Let P_1 and P_2 be the paths corresponding to blocks i and j, and let (p, q) be the first node at which they differ. Moreover, without loss of generality, assume that at (p, q) the path P_1 takes a North step while P_2 takes an East step.

The key idea of our proof is that, starting from (p, q), the minimum possible code that can be assigned to the remaining suffix of P_1 is strictly larger than the maximum possible code that can be assigned to the remaining suffix of P_2. This can be proved by induction after characterizing such quantities. Overall, this implies that the code assigned to P_1 differs from the code assigned to P_2, contradicting the assumption that $T[i] = T[j]$. ◀

3.2.2.2 Space Analysis

To show that the space usage of our encoding approaches that of Lemma 9, we introduce the following definition:

▶ **Definition 12.** *Let b and σ be positive integers with $b > \sigma$. We denote by M_b^σ the number of distinct normalized paths in G from any $(0, s)$ to $(b-1, b+\sigma-1)$ for all $0 \leq s < \sigma$.*

Ideally, we would like to analyze the sizes of L and T in terms of M_{b-1}^σ, but unfortunately, we are not aware of a closed formula for M_{b-1}^σ, and neither for the numbers $N[q][q]$.

However, we can still derive an upper bound as follows. Consider again the conceptual graph G from Def. 10: As depicted by the dotted part in Fig. 2, we can extend G to the left by adding σ levels according to the same construction rule, and only keeping the nodes lying at the same height and above node $(0, 0)$. If we consider all paths in the extended graph using only North and East steps and running from its leftmost node to its top rightmost node, these paths are in bijection with Dyck paths of length $b+\sigma-1$ and maximum height σ, which are counted by $C_{b+\sigma}^{\sigma+1}$. Moreover, for every path and every entry height in the original graph, there exists at least one such path in the extended graph that starts at its leftmost node and reaches that height (see the example in Fig. 2). Consequently, $M_b^\sigma \leq C_{b+\sigma}^{\sigma+1}$, and the space usage in bits of the type table $T[0, \lceil n/b \rceil - 1]$ can be bounded as follows.

$$\begin{aligned}
|T| &= \left\lceil \frac{n}{b} \right\rceil \left\lceil \log M_{b-1}^\sigma \right\rceil \leq \left\lceil \frac{n}{b} \right\rceil \left\lceil \log C_{b+\sigma}^{\sigma+1} \right\rceil \\
&\leq \left(\frac{n}{b} + 1 \right) \left((b+\sigma) \log r_\sigma - \log(\sigma+2) + O(1) \right) \\
&= n \log r_\sigma + O\left(\frac{n}{b}\right)
\end{aligned}$$

The space usage in bits for the table L storing the prefix sums can be bounded as follows:

$$\begin{aligned}
|L| &= M_{b-1}^\sigma (b-1) \lceil \log(b\,\sigma) \rceil \leq C_{b+\sigma}^{\sigma+1} (b-1)(\log(b\,\sigma) + 1) \\
&= \Theta\left(r_\sigma^{(b+\sigma)} b \log b \right)
\end{aligned}$$

Choosing the block size as $b = \left\lceil \frac{(1-\epsilon)\log n}{\log r_\sigma} - \sigma \right\rceil$ for an arbitrarily small constant $\epsilon > 0$ makes $|L| = O(n^{1-\epsilon}\log n \log\log n) = o(n/\log n)$ bits. The sample array S_{P_A} requires $\frac{n}{b}\lceil \log \sigma \rceil = O(n/b)$ bits; hence, the overall size of the encoding is $n \log r_\sigma + O(\frac{n}{\log n})$ bits. Queries are answered in $O(1)$ time as shown in Eq. 2 using only T and L.

Construction. Table N can be computed optimally in $O(b\,(b+\sigma)) = O(\log^2 n)$ time while using $O(b\,(b+\sigma)\log n) = O(\log^3 n)$ bits of space, and is only needed at construction time. The type table T is filled using Alg. 1 in $O(n)$ time. Moreover, the same algorithm can be modified easily to operate block-wise (i.e., computing $P_A[j \cdot b, (j+1)\cdot b - 1]$ and the resulting Δ_{P_A}-sequence only when the j-th block is processed), it never uses more than $O(b\,\sigma \log n) = O(\log^2 n)$ bits of working space. Lastly, the relevant entries of L can be computed on-the-fly in no more than $o(n)$ bits of space and time. Overall, we proved Thm. 3.

4 Conclusions

We introduced a new type of range query on arrays, with potential applications in text indexing as well as other algorithmic realms. Our algorithms build on techniques that are inspired by range minimum queries [7, 8], but needed substantial new ideas and adaptations to work for All-Distinct. We note that in cases where an All-Distinct query returns a negative answer, our algorithms can be easily modified to report the position of a duplicate in the query range, if desired. Future research directions include: proving a space-time trade-off lower bound in the indexing model [1]; extending the techniques to higher dimensional arrays [2]; considering upper and lower bounds for random instances [19]; compressing the data structures [9, 12]; learning from the distribution of the data [5]; or exploring relaxed variants, such as parameterized queries that return TRUE if the query range contains at most k repeated elements, with k specified either at preprocessing or at query time. So far, we have considered only the simplest case of queries testing the distinctness of values in a range. More challenging problems include deciding whether a unique element exists in the query range, as well as counting or reporting distinct or repeated elements, similarly to [11].

References

1 Gerth Stølting Brodal, Pooya Davoodi, Moshe Lewenstein, Rajeev Raman, and Srinivasa Rao Satti. Two dimensional range minimum queries and fibonacci lattices. *Theor. Comput. Sci.*, 638:33–43, 2016. doi:10.1016/J.TCS.2016.02.016.

2 Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012. doi:10.1007/S00453-011-9499-0.

3 N.G. de Bruijn, D.E. Knuth, and S.O. Rice. The average height of planted plane trees. In Ronald C. Read, editor, *Graph Theory and Computing*, pages 15–22. Academic Press, 1972. doi:10.1016/B978-1-4832-3187-7.50007-6.

4 Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014. doi:10.1007/S00453-012-9664-0.

5 Paolo Ferragina and Filippo Lari. FL-RMQ: A learned approach to range minimum queries. In Paola Bonizzoni and Veli Mäkinen, editors, *36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, June 17-19, 2025, Milan, Italy*, volume 331 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.CPM.2025.7.

6 Johannes Fischer. Compressed range minimum queries. In *Encyclopedia of Algorithms*, pages 379–382. Springer, 2016. doi:10.1007/978-1-4939-2864-4_640.

7 Johannes Fischer and Volker Heun. Finding range minima in the middle: Approximations and applications. *Math. Comput. Sci.*, 3(1):17–30, 2010. doi:10.1007/s11786-009-0007-8.

8 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

9 Johannes Fischer, Volker Heun, and Horst Martin Stühler. Practical entropy-bounded schemes for o(1)-range minimum queries. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA*, pages 272–281. IEEE Computer Society, 2008. doi:10.1109/DCC.2008.45.

10 Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. URL: http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521898065.

11 Travis Gagie, Juha Kärkkäinen, Gonzalo Navarro, and Simon J. Puglisi. Colored range queries and document retrieval. *Theor. Comput. Sci.*, 483:36–50, 2013. doi:10.1016/J.TCS.2012.08.004.

12 Pawel Gawrychowski, Seungbum Jo, Shay Mozes, and Oren Weimann. Compressed range minimum queries. *Theor. Comput. Sci.*, 812:39–48, 2020. doi:10.1016/J.TCS.2019.07.002.

13 Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comput. Sci.*, 387(3):348–359, 2007. doi:10.1016/J.TCS.2007.07.041.

14 Roberto Grossi. Wavelet trees. In *Encyclopedia of Algorithms*, pages 2355–2359. Springer, 2016. doi:10.1007/978-1-4939-2864-4_642.

15 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

16 Seungbum Jo and Srinivasa Rao Satti. Encodings for range minimum queries over bounded alphabets. *Theor. Comput. Sci.*, 1070:115824, 2026. doi:10.1016/J.TCS.2026.115824.

17 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

18 Anna Lubiw and András Rácz. A lower bound for the integer element distinctness problem. *Inf. Comput.*, 94(1):83–92, 1991. doi:10.1016/0890-5401(91)90034-Y.

19 J. Ian Munro, Patrick K. Nicholson, Louisa Seelbach Benkner, and Sebastian Wild. Hypersuccinct trees - new universal tree source codes for optimal compressed tree data structures and range minima. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 70:1–70:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.70.

20 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2013. doi:10.1145/2535933.

21 Gonzalo Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016. URL: http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB.

22 Mihai Pătraşcu. Succincter. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, Philadelphia, PA, USA, October 25-28, 2008*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

23 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

24 Matthew Skala. Array range queries. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms – Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 333–350. Springer, 2013. doi:10.1007/978-3-642-40273-9_21.

25 Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Tight better-than-worst-case bounds for element distinctness and set intersection, 2025. arXiv:2511.02954.

A Proof of Lemma 11

To simplify the proof of Lemma 11, we introduce two auxiliary lemmas. Their proofs proceed by induction on the cells of $N[0, b-1][0, b+\sigma-1]$, following the reverse sum order: for any two pairs of coordinates (p', q') and (p, q), $(p', q') \prec (p, q)$ if and only if $p' + q' > p + q$. Under this ordering, the smallest coordinate is $(b-1, b+\sigma-1)$, while the largest is $(0, 0)$. Intuitively, this ordering is more suited for the indexing of the N-array (see Figure 2). Finally, for ease of presentation, throughout the following proofs, we assume that $N[p][q] = 0$ for every $p > b-1$ or $q > p+\sigma$.

▶ **Lemma 13.** *For every* $0 \le p \le b-2$ *and* $p \le q \le p+\sigma$, $N[p][q] = \sum_{i=0}^{p+\sigma-q} N[p+1][q+i]$.

Proof. The base case immediately follows from Def. 3. For the inductive case, consider a generic coordinate (p, q), and assume the predicate holds for all $(p', q') \prec (p, q)$. Then:

$$\begin{aligned} N[p][q] &= N[p][q+1] + N[p+1][q] \\ &= N[p+1][q] + \sum_{i=0}^{p+\sigma-q-1} N[p+1][q+1+i] \quad \text{(Inductive hypothesis)} \\ &= \sum_{i=0}^{p+\sigma-q} N[p+1][q+i] \end{aligned}$$

◀

▶ **Lemma 14.** *For every* $0 \le p < b-1$ *and* $p \le q < p+\sigma$, $N[p][q] > \sum_{i=0}^{b-(p+2)} N[p+1+i][q+i]$.

Proof. The base case holds since by Def. 3, $N[b-2][b+\sigma-3] = N[b-1][b-1+\sigma] + N[b-1][b+\sigma-2] = 2$ and $N[b-1][b+\sigma-3] = 1$ (see Figure 2). For the inductive case, consider a generic coordinate (p, q), and assume the predicate holds for all $(p', q') \prec (p, q)$. Then:

$$\begin{aligned} N[p][q] &= N[p+1][q] + N[p][q+1] \\ &\ge N[p+1][q] + N[p+1][q+1] \quad \text{(Expanding } N[p][q+1], N[p][q+2] \ge 0) \\ &> N[p+1][q] + \sum_{i=0}^{b-(p+3)} N[p+2+i][q+1+i] \quad \text{(Inductive hypothesis)} \\ &= \sum_{i=0}^{b-(p+2)} N[p+1+i][q+i] \end{aligned}$$

◀

For ease of reference, we restate Lemma 11 here, and proceed with a formal proof based on the argument sketched in Sect. 3.2.2.

▶ **Lemma 15.** *Let* T *be the table produced by Algorithm 1. For all indices* $0 \le i \le j < \lceil n/b \rceil$, *if* $T[i] = T[j]$ *then for all* $1 \le k < b$, $\Delta_{P_A}[i \cdot b + k] = \Delta_{P_A}[j \cdot b + k]$.

Proof. We proceed by contradiction. Suppose there are two indices $0 \le i < j < \lceil n/b \rceil$ such that $T[i] = T[j]$, but for some $1 \le k < b$, $\Delta_{P_A}[i \cdot b + k] \ne \Delta_{P_A}[j \cdot b + k]$. This difference then translates to a difference in the corresponding Dyck path fragments of the two blocks.

Let P_1 and P_2 be the paths corresponding to blocks i and j, and let (p, q) be the first node at which they differ. For $k \in \{1, 2\}$, let the portion of P_k up to and including (p, q) be called the prefix of P_k, denoted as $\text{pref}(P_k)$, and let the remaining part be its suffix, denoted as $\text{suff}(P_k)$.

Without loss of generality, assume that at (p, q) the path P_1 takes a North step while P_2 takes an East step. Note that $0 \leq p \leq b-2$ and $p < q < p+\sigma$, otherwise these steps cannot be performed.

Since the paths coincide up to (p, q), we have $\text{pref}(P_1) = \text{pref}(P_2)$. The key idea of our proof is then to show that, after the mismatched step, the largest possible code that can be assigned to $\text{suff}(P_2)$ is strictly smaller than the minimum possible code that can be assigned to $\text{suff}(P_1)$. This gives a contradiction, because then the overall codes of the two blocks cannot be the same, and thus $T[i] \neq T[j]$.

Let c_1 denote the smallest possible code for $\text{suff}(P_1)$. After the initial North step from (p, q), which contributes $N[p+1][q]$ to c_1, this minimum is achieved by taking $q-p+1$ East steps and then following the lower diagonal of the graph until reaching node $(b-1, b-1)$. It is immediate that, besides the initial North step, all the remaining steps do not contribute to c_1, hence $c_1 = N[p+1][q]$.

Similarly, let c_2 denote the largest possible code for $\text{suff}(P_2)$. After the initial East step from (p, q), which does not contribute to c_2, this maximum is achieved by first taking $p+\sigma-q$ consecutive North steps. Each of these North steps contributes $N[p+2][q+i]$ to c_2 for $0 \leq i \leq p+\sigma-q$. Once these North steps are completed, the path reaches the upper diagonal of the graph and then follows that diagonal until it arrives at the node $(b-1,\ b+\sigma-1)$. Each North step performed while following the upper diagonal contributes $N[p+3+i][p+\sigma+1+i]$ to c_2 for $0 \leq i \leq b-(p+4)$. Overall, the value of c_2 is given by the following expression:

$$c_2 = \sum_{i=0}^{p+\sigma-q} N[p+2][q+i] + \sum_{i=0}^{b-(p+4)} N[p+3+i][p+\sigma+1+i] \tag{4}$$

Finally, we obtain the following chain of inequalities relating c_1 to c_2, from which our claim follows:

$$\begin{aligned}
c_1 &= N[p+1][q] \\
&= \sum_{i=0}^{p+\sigma-q+1} N[p+2][q+i] \quad \text{(Lemma 13)} \\
&= \left(\sum_{i=0}^{p+\sigma-q} N[p+2][q+i] \right) + N[p+2][p+\sigma+1] \\
&> \sum_{i=0}^{p+\sigma-q} N[p+2][q+i] + \sum_{i=0}^{b-(p+4)} N[p+3+i][p+\sigma+1+i] \quad \text{(Lemma 14)} \\
&= c_2 \quad \text{(Eq. 4)}.
\end{aligned}$$

◀

B An Alternative Encoding Based on Tree Covering

As mentioned in Sect. 1, it is possible to design an alternative encoding in the restricted alphabet setting of Sect. 3.2. This encoding is based on Lemma 7, which gives a bijection between the set of P-arrays of length n over an alphabet of size σ, and ordered trees with $n+1$ nodes and height at most $\sigma+1$. In particular, if we denote by T_A the ordered tree corresponding to P_A under this bijection for some input array A, then each $P_A[i]$ is related to the depth of the $(i+1)$-th node visited in a pre-order traversal of T_A.

▶ **Lemma 16.** *Let T_A be the ordered tree corresponding to P_A under the bijection of Lemma 7 for some input array A. For every $0 \leq i < n$ and every node u with pre-order rank $i+1$ in T_A, $P_A[i] = i + 2 - \textsc{depth}(u)$.*

Proof. Let B be the balanced parentheses sequence of T_A obtained from P_A through the bijection of Lemma 7. Because of the fictitious node, for every $0 \leq i < n$, a node u with pre-order rank $i+1$ has its opening parenthesis in B at position $i+1+\sum_{k=0}^{i} \Delta_{P_A}[k] = i+1+P_A[i]$. In particular, there are $i+2$ opening parentheses and $P_A[i]$ closing parentheses. The difference between these two quantities gives the depth of u in T_A. Therefore, $\textsc{depth}(u) = i+2-P_A[i]$, which implies $P_A[i] = i + 2 - \textsc{depth}(u)$. ◀

The encoding is then based on the following observation. By Lemma 7, the height of T_A, and hence the depth of each node, is bounded by $\sigma + 1$. It follows that this bound also holds locally for any subtree in any partition of the nodes of T_A into subtrees. The key idea of this second encoding is therefore to decompose T_A into a certain number of subtrees and, for each possible subtree shape, store a small precomputed table that maps nodes to their corresponding depth within the subtree. By Lemma 16, this allows us to access the values of P_A and ultimately answer All-Distinct queries on A. To this end, we employ the following tree decomposition technique by Farzan & Munro [4].

▶ **Lemma 17** (Tree Covering [4])**.** *For any parameter $b \geq 1$, an ordinal tree with n nodes can be decomposed, in linear time, into $\Theta(n/b)$ connected subtrees (so-called micro trees) having the following properties:*

- *Each micro tree contains at most $2b$ nodes.*
- *Micro trees are pairwise disjoint aside from their roots.*
- *Apart from edges leaving the micro tree root, at most one other edge leads to a node outside of this micro tree.*

We apply Lemma 17 to T_A using a parameter $b \geq 1$, to be fixed later, obtaining $m = \Theta(n/b)$ micro trees, each having a height bounded by $\sigma + 1$. For each micro tree μ, we store the depth of its root, its size, and the pre-order rank of its root in T_A.

The depths of the roots and their sizes can be stored respectively using $\Theta((n/b)\log\sigma)$ and $\Theta((n/b)\log b)$ bits. As for the pre-order rank of the roots, we observe that these values form a set of $\Theta(n/b)$ elements from a universe of size n; therefore, we encode them using a Fully Indexable Dictionary (FID), e.g., [23, 22] in $O((n/b)\log b)$ bits.

For each micro tree size $1 \leq j < 2b$, we store a precomputed table D_j mapping the local pre-order rank of nodes inside the micro tree to their depth (local to the micro tree). Overall, these precomputed tables use the following space in bits:

$$\sum_{j=1}^{2b-1} |D_j| = \sum_{j=1}^{2b-1} C_{j+1}^{\sigma+1} j \lceil \log(\sigma+1) \rceil \leq C_{2b}^{\sigma+1} (2b-1)^2 \lceil \log(\sigma+1) \rceil$$

This last quantity is $\Theta((r_\sigma^b b)^2 \log\sigma)$. By setting $b = \left\lceil \frac{(1-\varepsilon)}{2\log r_\sigma} \log n \right\rceil$ for an arbitrarily small constant $\varepsilon > 0$, the space usage of the tables becomes $O(n^{1-\varepsilon}\log^2 n) = o(n/\log n)$ bits.

We then proceed by assigning a code to each micro tree of size $1 \leq j < 2b$ using the same algorithm based on lattice paths of Sect. 3.2.2. In this setting, we apply Lemma 7 to convert each micro tree into its corresponding P-array, which is then converted into a Dyck path. Notice that here, each such path starts from $(0,0)$ and reaches (j,j). Therefore, contrary to Sect. 3.2.2, we can avoid prepending $\lceil \log\sigma \rceil$ bits to the obtained code, and we can directly use $C_\circ^{\sigma+1}$ to analyze the resulting type table T. Therefore, let s_k denote the size of the k-th micro tree, the space usage in bits of T can be bounded as follows:

$$|T| = \sum_{k=1}^{m} \lceil \log C_{s_k+1}^{\sigma+1} \rceil = \sum_{k=1}^{m} (s_k \log r_\sigma - 3\log(\sigma+2) + O(1))$$
$$\leq n \log r_\sigma + m(\log r_\sigma - 3\log(\sigma+2) + O(1)) \ ,$$

where the last inequality follows since each micro tree in the worst case can contain a duplicated root (i.e., $\sum_{k=1}^{m} s_k \leq n+m$, see Lemma 17). Hence $|T| = n \log r_\sigma + O(\frac{n}{\log n})$.

Finally, it is possible to prove that by using $O(n \log\log n / \log n)$ bits data structures, indices of P_A can be mapped to their corresponding micro tree in $O(1)$ time [4].

Therefore, given an ALL-DISTINCT$_A(l, r)$ query, we first map r to its corresponding micro tree μ_r. We then subtract from r the pre-order rank of the root of μ_r to obtain the local pre-order rank of r within μ_r. Using this value, together with the size and the type of μ_r, we access the corresponding precomputed table to retrieve the local depth of the node associated with r. We add this local depth to the depth of the root of μ_r and, by Lemma 16, derive the value of $P_A[r]$. The query is then answered as usual by checking whether $P_A[r] < l$.

All the above operations are performed in $O(1)$ time, and the overall space usage of the encoding is $n \log r_\sigma + O\left(\frac{n \log\log n}{\log n}\right)$ bits, which matches the lower bound of Thm. 9 up to lower-order terms.

We conclude by highlighting that the encoding of Thm. 3 presented in Sect. 3.2.2 has the same query time, but an asymptotically smaller lower-order term in the space usage, namely $O(n/\log n)$ against $O(n \log\log n / \log n)$. Moreover, to the best of our knowledge, we are not aware of a space-efficient construction algorithm for the tree covering of Lemma 17 as in Thm. 3. Lastly, the alternative encoding we presented here is not a mere black-box application of the technique by Farzan & Munro. In particular, our method for assigning types to height-restricted Dyck paths, and hence to trees of bounded height, is still needed for construction.