EFFICIENT SUCCINCT DATA STRUCTURES ON DIRECTED ACYCLIC GRAPHS

LUCA LOMBARDO

Tesi Triennale

Dipartimento di Matematica
Università di Pisa

Supervisor: Roberto Grossi

ABSTRACT

In this thesis, we introduce a novel approach to constructing succinct data structures for Directed Acyclic Graphs (DAGs), with particular emphasis on optimizing query performance. Our research centers on redefining the conventional rank query, typically employed in bitvector contexts, for DAGs. We focus on a specialized class of DAGs where to each node is associated a string composed of characters drawn from a fixed alphabet, Σ. We begin by partitioning the original DAG into $|\Sigma|$ distinct DAGs, each corresponding to a character in the alphabet. Within these character-specific DAGs, nodes store the count of occurrences of the respective character in the string associated with each node. We then propose a succinct representation of these DAGs, specifically engineered to solve rank queries. Our proposed rank query takes a node and a character as input and efficiently returns a range $\langle l, r \rangle$. This range represents the minimum and maximum possible occurrences of the character along all paths originating from the root node to the specified node within the DAG.

CONTENTS

1 INTRODUCTION

2

COMPRESSION PRINCIPLES AND METHODS

Entropy, in essence, represents the minimal quantity of bits required to unequivocally distinguish an object within a set. Consequently, it serves as a foundational metric for the space utilization in compressed data representations. The ultimate aim of compressed data structures is to occupy space nearly equivalent to the entropy required for object identification, while simultaneously enabling efficient querying operations. This pursuit lies at the core of optimizing data compression techniques: achieving a balance between storage efficiency and query responsiveness.

There are plenty of compression techniques, yet they share certain fundamental steps. In Figure 1 is shown the typical processes employed for data compression. These procedures depend on the nature of the data, and the arrangement or fusion of the blocks in 1 may differ. Numerical manipulation, such as predictive coding and linear transformations, is commonly employed for waveform signals like images and audio. Logical manipulation involves altering the data into a format more feasible to compression, including techniques such as run-length encoding, zero-trees, set-partitioning information, and dictionary entries. Then, source modeling is used to predict the data's behavior and structure, which is crucial for entropy coding.

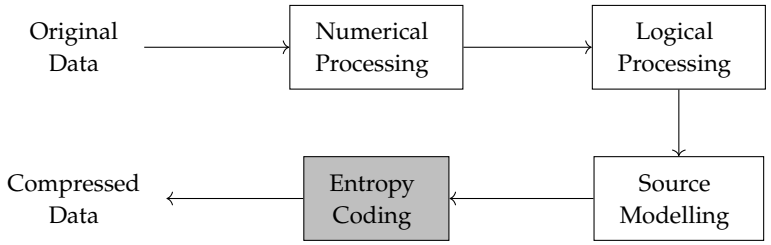


Figure 1: Typical processes in data compression

These initial numerical and logical processing stages typically aim to transform the data, exploiting specific properties like signal correlation or symbol repetition, to create a representation with enhanced statistical redundancy (e.g., more frequent symbols, predictable patterns). A common feature among most compression systems is the incorporation of *entropy coding* as the final process, wherein information is represented in the most compressed form possible. This stage may bear a significant impact on the overall compression ratio, as it is responsible for the final reduction in the data size. In this chapter we

will delve into the principles of entropy coding, exploring the fundamental concepts and methods that underpin this crucial stage of data compression.

2.1 WORST CASE ENTROPY

In its simplest form, entropy can be seen as the minimum number of bits required by identifiers (*codes*, see Section 2.3), when each element of a set U has a unique code of identical length. This is called the *worst case entropy* of U and it's denoted by $H_{wc}(U)$. The worst case entropy of a set U is given by the formula:

$$H_{wc}(U) = \log |U| \tag{1}$$

where $|U|$ is the number of elements in U.

Remark 2.1. *If we used codes of length $l < H_{wc}(U)$, we would have only $2^l \leqslant 2^{H_{wc}(U)} = |U|$ possible codes, which is not enough to uniquely identify all elements in U.*

The reason behind the attribute *worst case* is that if all codes are of the same length, then this length must be at least $\lceil \log |U| \rceil$ bits to be able to uniquely identify all elements in U. If they all have different lengths, the longest code must be at least $\lceil \log |U| \rceil$ bits long.

Example 2.2 (Worst-case entropy of $\mathcal{T}_n$)**.** *Let $\mathcal{T}_n$ denote the set of all general ordinal trees [5] with n nodes. In this scenario, each node can have an arbitrary number of children, and their order is distinguished. With n nodes, the number of possible ordinal trees is the $(n-1)$-th Catalan number, given by:*

$$|\mathcal{T}_n| = \frac{1}{n}\binom{2n-2}{n-1} \tag{2}$$

Using Stirling's approximation, we can estimate the worst-case entropy of $\mathcal{T}_n$ as:

$$|\mathcal{T}_n| = \frac{(2n-2)!}{n!(n-1)!} = \frac{(2n-2)^{2n-2} e^n e^{n-1}}{e^{2n-2} n^n (n-1)^{n-1} \sqrt{\pi n}} \left(1 + O\left(\frac{1}{n}\right)\right)$$

This simplifies to $\frac{4^n}{n^{3/2}} \cdot \Theta(1)$, hence

$$H_{wc}(\mathcal{T}_n) = \log |\mathcal{T}_n| = 2n - \Theta(\log n) \tag{3}$$

Thus, we have determined the minimum number of bits required to uniquely identify (encode) a general ordinal tree with n nodes.

2.2 ENTROPY

Let's introduce the concept of entropy as a measure of uncertainty of a random variable. While the worst-case entropy H_{wc}, discussed previously, provides a lower bound based solely on the set's cardinality (effectively assuming fixed-length codes or a uniform probability distribution over the elements), Shannon entropy offers a more refined measure. It accounts for the actual probability distribution of the elements, quantifying the *average* uncertainty or information content associated with the random variable. A deeper explanation can be found in [22, 33, 9]

Definition 2.3 (Entropy of a Random Variable)**.** *Let* X *be a random variable taking values in a finite alphabet* $\mathcal{X}$ *with the probabilistic distribution* $P_X(x) = Pr\{X = x\}$ $(x \in \mathcal{X})$*. Then, the entropy of* X *is defined as*

$$H(X) = H(P_X) \stackrel{\text{def}}{=} E_{P_x}\{-\log P_X(x)\} = -\sum_{x \in \mathcal{X}} P_X(x) \log P_X(x) \tag{1}$$

This is also known as Shannon entropy, named after Claude Shannon, who introduced it in his seminal work [45]

Where E_P denotes the expectation with respect to the probability distribution P. The log is taken to the base 2 and the entropy is expressed in bits. It is then clear that the entropy of a discrete random variable will always be nonnegative[1].

Example 2.4 (Toss of a fair coin)**.** *Let* X *be a random variable representing the outcome of a toss of a fair coin. The probability distribution of* X *is* $P_X(0) = P_X(1) = \frac{1}{2}$*. The entropy of* X *is*

$$H(X) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\log\frac{1}{2} = 1 \tag{2}$$

This means that the toss of a fair coin has an entropy of 1 bit.

Remark 2.5. *Due to historical reasons, we are abusing the notation and using* $H(X)$ *to denote the entropy of the random variable* X*. It's important to note that this is not a function of the random variable: it's a functional of the distribution of* X*. It does not depend on the actual values taken by the random variable, but only on the probabilities of these values.*

The concept of entropy, introduced in definition 2.3, helps us quantify the randomness or uncertainty associated with a random variable. It essentially reflects the average amount of information needed to identify a specific value drawn from that variable. Intuitively, we can think of entropy as the average number of digits required to express a sampled value.

1 The entropy is null if and only if $X = c$, where c is a costant with probability one

2.2.1 *Properties*

In the previous section 2.2, we have introduced the entropy of a single random variable X. What if we have two random variables X and Y? How can we measure the uncertainty of the pair (X, Y)? This is where the concept of joint entropy comes into play. The idea is to consider (X, Y) as a single vector-valued random variable and compute its entropy. This is the joint entropy of X and Y. To quantify the total uncertainty associated with a pair of variables considered together, we define the joint entropy:

Definition 2.6 (Joint Entropy)**.** *Let (X, Y) be a pair of discrete random variables (X, Y) with a joint distribution $P_{XY}(x, y) = Pr\{X = x, Y = y\}$. The joint entropy of (X, Y) is defined as*

$$H(X, Y) = H(P_{XY}) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P_{XY}(x, y) \log P_{XY}(x, y) \tag{3}$$

Which we can be extended to the joint entropy of n random variables $(X_1, X_2, \ldots, X_n)$ as $H(X_1, \ldots, X_n)$.

We also define the conditional entropy of a random variable given another as the expected value of the entropies of the conditional distributions, averaged over the conditioning random variable. Often, it's helpful to conceptualize the relationship between Y and X in terms of information transmission. Given X, we can determine the probability of observing Y through the conditional probability $W(y|x) = \Pr\{Y = y|X = x\}$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. The collection W of these conditional probabilities effectively describes how information about X influences the outcome of Y, and is often referred to as a *channel* with *input alphabet* $\mathcal{X}$ and *output alphabet* $\mathcal{Y}$.

Definition 2.7 (Conditional Entropy)**.** *Let (X, Y) be a pair of discrete random variables with a joint distribution $P_{XY}(x, y) = Pr\{X = x, Y = y\}$. The conditional entropy of Y given X is defined as*

$$\begin{aligned} H(Y|X) = H(W|P_X) &\stackrel{\text{def}}{=} \sum_x P_X(x) H(Y|x) && (4)\\ &= \sum_{x \in \mathcal{X}} P_X(x) \Big\{ -\sum_{y \in \mathcal{Y}} W(y|x) \log W(y|x) \Big\} && (5)\\ &= -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P_{XY}(x, y) \log W(y|x) && (6)\\ &= E_{P_{XY}}\{-\log W(Y|X)\} && (7) \end{aligned}$$

Since entropy is always nonnegative, conditional entropy is likewise nonnegative; it has value zero if and only if Y can be entirely determined from X with certainty, meaning there exists a function $f(X)$ such that $Y = f(X)$ with probability one.

The connection between joint entropy and conditional is more evident when considering that the entropy of two random variables equals the entropy of one of them plus the conditional entropy of the other. This connection is formally proven in the following theorem.

Theorem 2.8 (Chain Rule). *Let* (X, Y) *be a pair of discrete random variables with a joint distribution* $P_{XY}(x, y)$. *Then, the joint entropy of* (X, Y) *can be expressed as*

This is also known as additivity of entropy.

$$H(X, Y) = H(X) + H(Y|X) \tag{8}$$

Proof. From the definition of conditional entropy (2.7), we have

$$\begin{aligned} H(X, Y) &= -\sum_{x,y} P_{XY}(x, y) \log W(y|x) \\ &= -\sum_{x,y} P_{XY}(x, y) \log \frac{P_{XY}(x, y)}{P_X(x)} \\ &= -\sum_{x,y} P_{XY}(x, y) \log P_{XY}(x, y) + \sum_{x,y} P_X(x) \log P_X(x) \\ &= H(XY) + H(X) \end{aligned}$$

Where we used the relation

$$W(y|x) = \frac{P_{XY}(x, y)}{P_X(x)} \tag{9}$$

When $P_X(x) \neq 0$. □

Corollary 2.9.

$$H(X, Y|Z) = H(X|Z) + H(Y|X, Z) \tag{10}$$

Proof. The proof is analogous to the proof of the chain rule. □

Corollary 2.10.

$$\begin{aligned} H(X_1, X_2, \ldots, X_n) = H(X_1) + H(X_2|X_1) + H(X_3|X_1, X_2) \\ + \ldots + H(X_n|X_1, X_2, \ldots, X_{n-1}) \end{aligned} \tag{11}$$

Proof. We can apply the two-variable chain rule in repetition obtain the result. □

2.2.2 *Mutual Information*

Given two random variables X and Y, the mutual information between them quantifies the reduction in uncertainty about one variable due to the knowledge of the other. It is defined as the difference between the entropy and the conditional entropy. Figure 2 illustrates the concept of mutual information between two random variables. We've seen how to measure the uncertainty of individual variables and pairs. But how much does knowing one variable tell us about the other? In other words, how much uncertainty about X is removed by knowing Y? This is quantified by the mutual information:

Definition 2.11 (Mutual Information)**.** *Let (X, Y) be a pair of discrete random variables with a joint distribution $P_{XY}(x, y)$. The mutual information between X and Y is defined as*

$$I(X;Y) = H(X) - H(X|Y) \tag{12}$$

Using the chain rule (2.8), we can rewrite it as

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(X) + H(Y) - H(X,Y) && (13) \\ &= -\sum_x P_X(x) \log P_X(x) - \sum_y P_Y(y) \log P_Y(y) \\ &\quad + \sum_{x,y} P_{XY}(x,y) \log P_{XY}(x,y) && (14) \\ &= \sum_{x,y} P_{XY}(x,y) \log \frac{P_{XY}(x,y)}{P_X(x)P_Y(y)} && (15) \\ &= E_{P_{XY}} \left\{ \log \frac{P_{XY}(x,y)}{P_X(x)P_Y(y)} \right\} && (16) \end{aligned}$$

It follows immediately that the mutual information is symmetric, $I(X;Y) = I(Y;X)$.

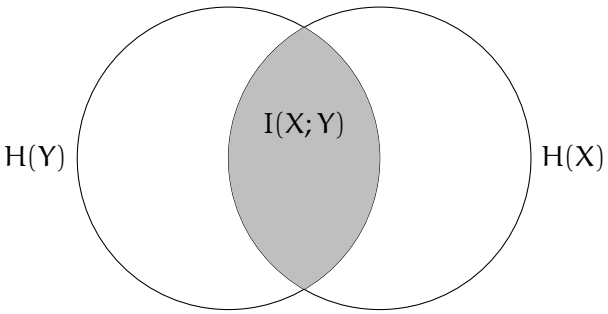


Figure 2: Mutual information between two random variables X and Y.

2.2.3 *Fano's inequality*

Information theory serves as a cornerstone for understanding fundamental limits in data compression. It not only allows us to prove the existence of encoders (Section 2.3) achieving demonstrably good performance, but also establishes a theoretical barrier against surpassing this performance. The following theorem, known as Fano's inequality, provides a lower bound on the probability of error in guessing a random variable X to it's conditional entropy $H(X|Y)$, where Y is another random variable[2].

Theorem 2.12 (Fano's Inequality). *Let X and Y be two discrete random variables with X taking values in some discrete alphabet $\mathcal{X}$, we have*

$$H(X|Y) \leqslant Pr\{X \neq Y\} \log(|\mathcal{X}| - 1) + h(Pr\{X \neq Y\}) \tag{17}$$

where $h(p) = -p \log p - (1-p) \log(1-p)$ is the binary entropy function.

Proof. Let Z be a random variable defined as follows:

$$Z = \begin{cases} 1 & \text{if } X \neq Y \\ 0 & \text{if } X = Y \end{cases} \tag{18}$$

We can then write

$$\begin{aligned} H(X|Y) &= H(X|Y) + H(Z|XY) = H(XZ|Y) \\ &= H(X|YZ) + H(Z|Y) \\ &\leqslant H(X|YZ) + H(Z) \end{aligned} \tag{19}$$

The last inequality follows from the fact that conditioning reduces entropy. We can then write

$$H(Z) = h(\Pr\{X \neq Y\}) \tag{20}$$

Since $\forall y \in \mathcal{Y}$, we can write

$$H(X|Y = y, Z = 0) = 0 \tag{21}$$

and

$$H(X|Y = y, Z = 1) \leqslant \log(|\mathcal{X}| - 1) \tag{22}$$

Combining these results, we have

$$H(X|YZ) \leqslant \Pr\{X \neq Y\} \log(|\mathcal{X}| - 1) \tag{23}$$

From equations 19, 20 and 23, we have Fano's inequality. □

2 We have seen in 2.7 that the conditional entropy of X given Y is zero if and only if X is a deterministic function of Y. Hence, we can estimate X from Y with zero error if and only if $H(X|Y) = 0$.

Fano's inequality thus provides a tangible link between the conditional entropy $H(X|Y)$, which quantifies the remaining uncertainty about X when Y is known, and the minimum probability of error achievable in any attempt to estimate X from Y. This inequality, along with the foundational concepts of entropy, joint entropy, conditional entropy, and mutual information introduced throughout this section, establishes a robust theoretical framework. These tools are not merely abstract measures; they allow us to quantify information, understand dependencies between data sources, and ultimately, to delineate the fundamental limits governing how efficiently data can be represented and compressed. Understanding these limits is essential as we delve deeper into specific encoding techniques.

2.3 SOURCE AND CODE

In the previous section, we established information-theoretic limits based on the probabilistic nature of data sources. Now, we turn our attention to the practical mechanisms for achieving data compression: the interplay between a *source* of information and the *code* used to represent it. A source, in this context, can be thought of as any process generating a sequence of symbols drawn from a specific alphabet (e.g., letters of text, pixel values in an image, sensor readings). Source coding, or data compression, is the task of converting this sequence into a different, typically shorter, sequence of symbols from a target coding alphabet (often binary).

The core principle behind efficient coding is to exploit the statistical properties of the source. Symbols or patterns that occur frequently should ideally be assigned shorter representations (codewords), while less frequent ones can be assigned longer codewords. A classic, intuitive example is Morse code: the most common letter in English text, 'E', is represented by the shortest possible signal, a single dot ('.'), whereas infrequent letters like 'Q' ('–.-') receive much longer sequences.

2.3.1 *Codes*

A source characterized by a random process generates symbols from a specific alphabet at each time step. The objective is to transform this output sequence into a more concise representation. This data reduction technique, known as *source coding* or *data compression*, utilizes a code to represent the original symbols more efficiently. The device that performs this transformation is termed an *encoder*, and the process itself is referred to as *encoding*. [22]

Definition 2.13 (Source Code)**.** *A source code for a random variable* X *is a mapping from the set of possible outcomes of* X*, called* $\mathcal{X}$*, to* $\mathcal{D}^*$*, the set of all finite-length strings of symbols from a* $\mathcal{D}$*-ary alphabet. Let* $C(X)$ *denote the codeword assigned to* x *and let* $l(x)$ *denote length of* $C(x)$

Definition 2.14 (Expected length)**.** *The expected length* $L(C)$ *of a source code* C *for a random variable* X *with probability mass function* $P_X(x)$ *is defined as*

$$L(C) = \sum_{x \in \mathcal{X}} P_X(x) l(x) \tag{1}$$

where $l(x)$ *is the length of the codeword assigned to* x*.*

Let's assume from now for simplicity that the $\mathcal{D}$-ary alphabet is $\mathcal{D} = \{0, 1, \ldots, D-1\}$.

Example 2.15. *Let's consider a source code for a random variable* X *with* $\mathcal{X} = \{a, b, c, d\}$ *and* $P_X(a) = 0.5$, $P_X(b) = 0.25$, $P_X(c) = 0.125$ *and* $P_X(d) = 0.125$. *The code is defined as*

$$C(a) = 0$$
$$C(b) = 10$$
$$C(c) = 110$$
$$C(d) = 111$$

The entropy of X *is*

$$H(X) = 0.5 \log 2 + 0.25 \log 4 + 0.125 \log 8 + 0.125 \log 8 = 1.75 \text{ bits}$$

The expected length of this code is also 1.75*:*

$$L(C) = 0.5 \cdot 1 + 0.25 \cdot 2 + 0.125 \cdot 3 + 0.125 \cdot 3 = 1.75 \text{ bits}$$

In this example we have seen a code that is optimal in the sense that the expected length of the code is equal to the entropy of the random variable.

Definition 2.16 (Nonsingular Code)**.** *A code is nonsingular if every element of the range of* X *maps to a different element of* $\mathcal{D}^*$. *Thus:*

$$x \neq y \Rightarrow C(x) \neq C(y) \tag{2}$$

While a single unique code can represent a single value from our source X without ambiguity, our real goal is often to transmit sequences of these values. In such scenarios, we could ensure the receiver can decode the sequence by inserting a special symbol, like a "comma," between each codeword. However, this approach wastes the special symbol's potential. To overcome this inefficiency, especially when dealing with sequences of symbols from X, we can leverage the concept of self-punctuating or instantaneous codes. These codes possess a special property: the structure of the code itself inherently indicates the end of each codeword, eliminating the need for a separate punctuation symbol. The following definitions formalize this concept. [9]

Definition 2.17 (Extension of a Code)**.** *The extension* C^* *of a code* C *is the mapping from finite-length sequences of symbols from* $\mathcal{X}$ *to finite-length strings of symbols from the* $\mathcal{D}$*-ary alphabet defined by*

$$C^*(x_1 x_2 \ldots x_n) = C(x_1)C(x_2)\ldots C(x_n) \tag{3}$$

where $C(x_1)C(x_2)\ldots C(x_n)$ *denotes the concatenation of the codewords assigned to* $x_1, x_2, \ldots, x_n$.

Example 2.18. *If* $C(x_1) = 0$ *and* $C(x_2) = 110$*, then* $C^*(x_1x_2) = 0110$.

Definition 2.19 (Unique Decodability)**.** *A code* C *is uniquely decodable if its extension is nonsingular*

Thus, any encoded string in a uniquely decodable code has only one possibile source string that could have generated it.

Definition 2.20 (Prefix Code)**.** *A code is a prefix code if no codeword is a prefix of any other codeword.*

Also called instantaneous code

Imagine receiving a string of coded symbols. An *instantaneous code* allows us to decode each symbol as soon as we reach the end of its corresponding codeword. We don't need to wait and see what comes next. Because the code itself tells us where each codeword ends, it's like the code "punctuates itself" with invisible commas separating the symbols. This let us decode the entire message by simply reading the string and adding commas between the codewords without needing to see any further symbols. Consider the example 2.15 seen a the beginning of this section, where the binary string `01011111010` is decoded as `0,10,111,110,10` because the code used naturally separates the symbols. [9]. Figure 3 shows the relationship between different types of codes.

Example 2.21 (Morse Code)**.** *Morse code serves as a classic illustration of these concepts. Historically used for telegraphy, it represents text characters using sequences from a ternary alphabet: a short signal (dot, '.'), a longer signal (dash, '-'), and a space (pause used as a delimiter). Frequent letters like 'E' receive short codes ('.'), while less common ones like 'Q' get longer codes ('–.-'). Here are a few examples:*

Character/Sequence	***Code***
E	.
T	-
A	. -
N	- .
S	. . .
O	- -
SOS	. . . - - . . .

Let's evaluate Morse code based on our definitions:

- ***Nonsingular:*** *The code is nonsingular because each letter corresponds to a unique sequence of dots and dashes. For instance,* $E \neq T$*, and their respective codes* $C(E) =$. *and* $C(T) =$ - *are distinct.*

- ***Prefix Code:*** *The code does not satisfy the prefix condition. Several codewords are prefixes of others. For example,* $C(E) = .$ *is a prefix of* $C(A) = .-$ *and* $C(S) = ...$. *Similarly,* $C(T) = -$ *is a prefix of* $C(N) = -.$ *and* $C(M) = --$. *This lack of the prefix property means that receiving a sequence like '.–' is ambiguous without further information; it could represent 'A' or the sequence 'ET'.*
- ***Uniquely Decodable:*** *The code achieves unique decodability, but this relies critically on the use of pauses (spaces) inserted between letters and words according to specific timing rules. These pauses function as explicit delimiters. Without them, the inherent ambiguity due to the lack of the prefix property would make decoding impossible. This contrasts with true prefix codes (like Example 2.15), which are uniquely decodable based solely on their structure, without needing external delimiters. For example, the sequence '.—' is unambiguously decoded as 'ET' only when the timing correctly separates the 'E' ('.') from the 'T' ('-').*

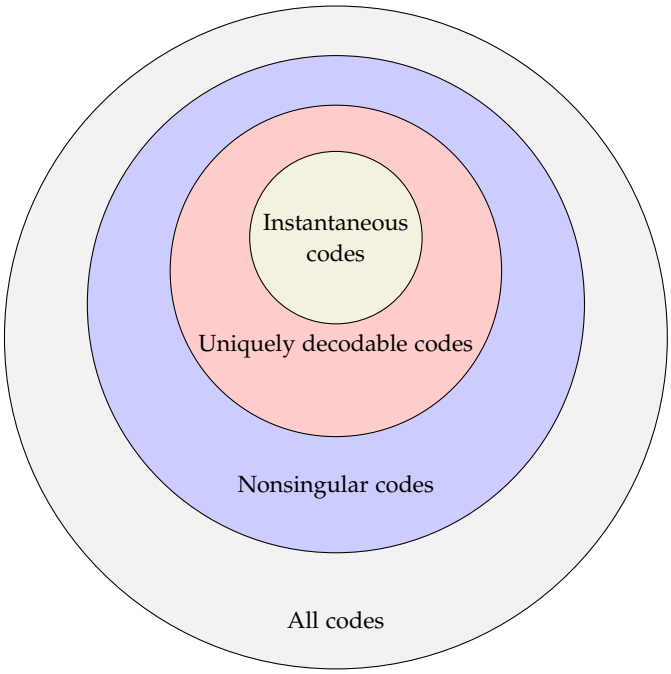


Figure 3: Relationship between different types of codes

2.3.2 *Kraft's Inequality*

We aim to construct efficient codes, ideally prefix codes (instantaneous codes), whose expected length approaches the source entropy. A fundamental constraint arises because we cannot arbitrarily assign short lengths to all symbols while maintaining the prefix property or even unique decodability. Kraft's inequality precisely quantifies this

limitation. It establishes a *necessary* condition that the chosen codeword lengths $l(x)$ must satisfy for *any uniquely decodable* code to exist. Crucially, the same inequality also serves as a *sufficient* condition guaranteeing that a *prefix* code with these exact lengths can indeed be constructed. We will first state and prove the necessity part for uniquely decodable codes.

Let's denote the size of the source and code alphabets with $J = |\mathcal{X}|$ and $K = |\mathcal{D}|$, respectively. Different proofs of the following theorem can be found in [9, 22], here we report the one from [22], however the one proposed in [9] is also very interesting, based on the concept of a source tree.

Theorem 2.22 (Kraft's Inequality)**.** *The codeword lengths* $l(x)$, $x \in \mathcal{X}$, *of any uniquely decodable code* C *over a* K*-ary alphabet must satisfy the inequality*

$$\sum_{x \in \mathcal{X}} K^{-l(x)} \leqslant 1 \tag{4}$$

Proof. Consider the left hand side of the inequality 4 and consider its n-th power

$$\begin{aligned}\left(\sum_{x \in \mathcal{X}} K^{-l(x)}\right)^n &= \sum_{x_1 \in \mathcal{X}} \sum_{x_2 \in \mathcal{X}} \dots \sum_{x_n \in \mathcal{X}} K^{-l(x_1)} K^{-l(x_2)} \dots K^{-l(x_n)} \\ &= \sum_{x^n \in \mathcal{X}^\setminus} K^{-l(x^n)}\end{aligned} \tag{5}$$

Where $l(x^n) = l(x_1) + l(x_2) + \dots + l(x_n)$ is the length of the concatenation of the codewords assigned to $x_1, x_2, \dots, x_n$. If we consider the all the extended codewords of length m we have

$$\sum_{x^n \in \mathcal{X}^\setminus} K^{-l(x^n)} = \sum_{m=1}^{n l_{max}} A(m) K^{-m} \tag{6}$$

where $A(m)$ is the number source sequences of length n whose codewords have length m and l_{max} is the maximum length of the codewords in the code. Since the code is separable, we have that $A(m) \leqslant K^m$ and therefore each term of the sum is less than or equal to 1. Hence

$$\left(\sum_{x \in \mathcal{X}} K^{-l(x)}\right)^n \leqslant n l_{max} \tag{7}$$

That is

$$\sum_{x \in \mathcal{X}} K^{-l(x)} \leqslant (n l_{max})^{1/n} \tag{8}$$

Taking the limit as n goes to infinity and using the fact that $(nl_{max})^{1/n} = e^{1/n \log(nl_{max})} \to 1$ we have that

$$\sum_{x \in \mathcal{X}} K^{-l(x)} \leqslant 1 \tag{9}$$

That concludes the proof. □

2.3.3 *Source Coding Theorem*

Add some introduction from [9, 45, 1, 22]

Theorem 2.23 (Source Coding Theorem). *TODO from [9, 22]*

Proof. TODO from [9, 22] □

2.4 EMPIRICAL ENTROPY

Before digging into the concept of empirical entropy, let's begin with the notion of binary entropy. Consider an alphabet $\mathcal{U}$, where $\mathcal{U} = \{0, 1\}$. Let's assume it emits symbols with probabilities p_0 and $p_1 = 1 - p_0$. The entropy of this source can be calculated using the formula:

$$H(p_0) = -p_0 \log_2 p_0 - (1 - p_0) \log_2 (1 - p_0)$$

We can extend this concept to scenarios where the elements are no longer individual bits, but sequences of these bits emitted by the source. Initially, let's assume the source is *memoryless* (or *zero-order*), meaning the probability of emitting a symbol doesn't depend on previously emitted symbols. In this case, we can consider chunks of n bits as our elements. Our alphabet becomes $\Sigma = \{0, 1\}^n$, and the Shannon Entropy of two independent symbols $x, y \in \Sigma$ will be the sum of their entropies. Thus, if the source emits symbols from a general alphabet Σ of size $|\Sigma| = \sigma$, where each symbol $s \in \Sigma$ has a probability p_s (with $\sum_{s \in \Sigma} p_s = 1$), the Shannon entropy of the source is given by:

$$H(P) = H(p_1, \ldots, p_\sigma) = -\sum_{s \in \Sigma} p_s \log p_s = \sum_{s \in \Sigma} p_s \log \frac{1}{p_s}$$

Remark 2.24. *If all symbols have a probability of* $p_s = 1$*, then the entropy is* 0*, and all other probabilities are* 0*. If all symbols have the same probability* $\frac{1}{\sigma}$*, then the entropy is* $\log \sigma$*. So given a sequence of* n *elements from an alphabet* Σ*, belonging to* $\mathcal{U} = \Sigma^n$*, its entropy is straightforwardly* $n\mathcal{H}(p_1, \ldots, p_\sigma)$

2.4.1 *Bit Sequences*

In many practical scenarios, however, we do not know the true probabilities p_s of the underlying source. Instead, we might only have access to a sequence generated by the source. The concept of empirical entropy allows us to estimate the information content based directly on the observed frequencies within that sequence. Let's first examine this for binary sequences.

Let's consider a bit sequence, $B[1, n]$, which we aim to compress without access to an explicit model of a known bit source. Instead, we only have access to B. Although lacking a precise model, we may reasonably anticipate that B exhibits a bias towards either more 0s or more 1s. Hence, we might attempt to compress B based on this characteristic. Specifically, we say that B is generated by a zero-order source emitting 0s and 1s. Assuming m represents the count of 1s

in B, it's reasonable to posit that the source emits 1s with a probability of $p = m/n$. This leads us to the concept of zero-order empirical entropy:

Definition 2.25 (Zero-order empirical entropy)**.** *Given a bit sequence* $B[1, n]$ *with* m 1s *and* $n - m$ 0s*, the zero-order empirical entropy of* B *is defined as:*

$$\mathcal{H}_0(B) = \mathcal{H}\left(\frac{m}{n}\right) = \frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m} \tag{1}$$

The concept of zero-order empirical entropy carries significant weight: it indicates that if we attempt to compress B using a fixed code $C(1)$ for 1s and $C(0)$ for 0s, then it's impossible to compress B to fewer than $\mathcal{H}_0(B)$ bits per symbol. Otherwise, we would have $m|C(1)| + (n-m)|C(0)| < n\mathcal{H}_0(B)$, which violates the lower bound established by Shannon entropy.

CONNECTION WITH WORST CASE ENTROPY It is interesting to note a connection between the zero-order empirical entropy $\mathcal{H}_0(B)$ and the worst-case entropy H_{wc} previously introduced (Section 2.1). Consider the specific set $\mathcal{B}_{n,m}$ comprising all possible binary sequences of length n that contain exactly m ones, like our sequence B. The worst-case entropy necessary to assign a unique identifier to each sequence *within this set* is $H_{wc}(\mathcal{B}_{n,m}) = \log|\mathcal{B}_{n,m}| = \log\binom{n}{m}$. Using Stirling's approximation for the binomial coefficient, it can be demonstrated that this quantity is closely related to the total empirical entropy: $H_{wc}(\mathcal{B}_{n,m}) \approx n\mathcal{H}_0(B) - O(\log n)$. Thus, $n\mathcal{H}_0(B)$ approximates the minimum number of bits required, on average per sequence, to distinguish among all sequences sharing the same number of 0s and 1s, providing another perspective on the meaning of empirical entropy [33].

2.4.2 *Entropy of a Text*

The zero-order empirical entropy of a string $S[1, n]$, where each symbol s occurs n_s times in S, is similarly determined by the Shannon entropy of its observed probabilities:

Definition 2.26 (Zero-order empirical entropy of a text)**.** *Given a text* $S[1, n]$ *with* n_s *occurrences of symbol* s*, the zero-order empirical entropy of* S *is defined as:*

$$\mathcal{H}_0(S) = \mathcal{H}\left(\frac{n_1}{n}, \ldots, \frac{n_\sigma}{n}\right) = \sum_{s=1}^{\sigma} \frac{n_s}{n}\log\frac{n}{n_s} \tag{2}$$

Example 2.27. *Let* $S =$ *"abracadabra". We have that* $n = 11$, $n_a = 5$, $n_b = 2$, $n_c = 1$, $n_d = 1$, $n_r = 2$. *The zero-order empirical entropy of* S *is:*

$$\mathcal{H}_0(S) = \frac{5}{11}\log\frac{11}{5} + 2 \cdot \frac{2}{11}\log\frac{11}{2} + 2 \cdot \frac{1}{11}\log\frac{11}{1} \approx 2.04$$

Thus, we could expect to compress S *to* $nH_0(S) \approx 22.44$ *bits, which is lower than the* $n \log \sigma = 11 \cdot \log 5 \approx 25.54$ *bits of the worst-case entropy of a general string of length* n *over an alphabet of size* $\sigma = 5$.

However, this definition falls short because in most natural languages, symbol choices aren't independent. For example, in English text, the sequence "*don*'" is almost always followed by "*t*". Higher-order entropy (Section 2.5) is a more accurate measure of the entropy of a text, as it considers the probability of a symbol given the preceding symbols. This principle was at the base of the development of the famous Morse Code and then the Huffman code (Section 2.7).

2.5 HIGHER ORDER ENTROPY

The zero-order empirical entropy $\mathcal{H}_0(S)$, discussed in the previous section, provides a useful baseline for compression by considering the frequency of individual symbols. However, it operates under the implicit assumption that symbols are generated independently, a condition seldom met in practice, especially for data like natural language text. For instance, the probability of encountering the letter 'u' in English text dramatically increases if the preceding letter is 'q'. To capture such dependencies and obtain a more accurate measure of the information content considering local context, we introduce the concept of *higher-order empirical entropy*. This approach conditions the probability of a symbol's occurrence on the sequence of k symbols that immediately precede it.

Definition 2.28 (Redundancy)**.** *For an information source* X *generating symbols from an alphabet* Σ, *the redundancy* R *is the difference between the maximum possible entropy per symbol and the actual entropy* $H(X)$ *of the source:*

$$R = \log_2 |\Sigma| - H(X) \tag{1}$$

This redundancy value, R, quantifies the degree of predictability or statistical structure inherent in the source. A high redundancy signifies that the source is far from random, exhibiting patterns (like non-uniform symbol probabilities or inter-symbol dependencies) that can potentially be exploited for compression. Conversely, a source with low redundancy behaves more randomly, leaving less room for compression beyond the theoretical minimum dictated by $H(X)$.

However, evaluating redundancy directly using Definition 2.28 often proves impractical, as determining the true source entropy $H(X)$ for the process generating a given string S is typically unfeasible. This limitation necessitates alternative, empirical approaches. To address this issue, we introduce the concept of the *k-th order empirical entropy* of a string S, denoted as $\mathcal{H}_k(S)$. In statistical coding (Section 2.7), we will see a scenario where $k = 0$, relying on symbol frequencies within the string. Now, with $\mathcal{H}_k(S)$, our objective is to extend the entropy concept by examining the frequencies of k-grams in string S. This requires analyzing subsequences of symbols with a length of k, thereby capturing the *compositional structure* of S [12].

Let S be a string of length $n = |S|$ over an alphabet Σ of size $|\Sigma| = \sigma$. Let ω denote a k-gram (a sequence of k symbols from Σ), and let n_ω be the number of occurrences of ω in S. Let $n_{\omega\sigma_i}$ be the number of times the k-gram ω is followed by the symbol $\sigma_i \in \Sigma$ in S. [3]

Definition 2.29 (k-th Order Empirical Entropy)**.** *The* k-th order empirical entropy *of a string* S *is defined as:*

$$\mathcal{H}_k(S) = \frac{1}{n} \sum_{\omega \in \Sigma^k} \left(\sum_{\sigma_i \in \Sigma} n_{\omega\sigma_i} \log_2 \left(\frac{n_\omega}{n_{\omega\sigma_i}} \right) \right) \tag{2}$$

where terms with $n_{\omega\sigma_i} = 0$ *contribute zero to the sum.*

This definition calculates the average conditional entropy based on the preceding k symbols. An equivalent and often more intuitive way to express this is by averaging the zero-order empirical entropies of the sequences formed by the symbols following each distinct k-gram context:

$$\mathcal{H}_k(S) = \sum_{\omega \in \Sigma^k, n_\omega > 0} \frac{n_\omega}{n} \cdot \mathcal{H}_0(S_\omega) \tag{3}$$

where S_ω is the string formed by concatenating all symbols that immediately follow an occurrence of the k-gram ω in S (its length is $|S_\omega| = n_\omega$). The sum is taken over all k-grams ω that actually appear in S (i.e., $n_\omega > 0$).

Example 2.30. *Consider the example 2.27, where* $S =$ *"abracadabra"* ($n = 11$) *and* $\Sigma = \{a, b, c, d, r\}$ ($\sigma = 5$)*. The zero-order empirical entropy is* $\mathcal{H}_0(S) \approx 2.04$*. Now, let's calculate the first-order* ($k = 1$) *empirical entropy using Equation 3. The contexts are the single characters:*

- *Context 'a'* ($n_a = 5$)*: Following symbols are 'b', 'c', 'd', 'b', '$' (assuming end-of-string marker).* $S_a =$ *"bcdb$".* $\mathcal{H}_0(S_a) \approx 1.922$ *bits/symbol (assuming* $ *is a unique symbol).*

3 We use the notation $\omega \in \Sigma^k$ for a k-gram.

- *Context 'b'* ($n_b = 2$)*: Following symbols are 'r', 'r'.* S_b *= "rr".* $\mathcal{H}_0(S_b) = 0$ *bits/symbol.*
- *Context 'c'* ($n_c = 1$)*: Following symbol is 'a'.* S_c *= "a".* $\mathcal{H}_0(S_c) = 0$ *bits/symbol.*
- *Context 'd'* ($n_d = 1$)*: Following symbol is 'a'.* S_d *= "a".* $\mathcal{H}_0(S_d) = 0$ *bits/symbol.*
- *Context 'r'* ($n_r = 2$)*: Following symbols are 'a', 'a'.* S_r *= "aa".* $\mathcal{H}_0(S_r) = 0$ *bits/symbol.*

Therefore, the first-order empirical entropy of S *is:*

$$\mathcal{H}_1(S) = \frac{n_a}{n}\mathcal{H}_0(S_a) + \frac{n_b}{n}\mathcal{H}_0(S_b) + \frac{n_c}{n}\mathcal{H}_0(S_c) + \frac{n_d}{n}\mathcal{H}_0(S_d) + \frac{n_r}{n}\mathcal{H}_0(S_r)$$

$$\mathcal{H}_1(S) = \frac{5}{11}\cdot(1.922) + \frac{2}{11}\cdot 0 + \frac{1}{11}\cdot 0 + \frac{1}{11}\cdot 0 + \frac{2}{11}\cdot 0 \approx 0.874 \text{ bits/symbol}$$

This value is significantly lower than the zero-order empirical entropy $\mathcal{H}_0(S)$*, reflecting the predictability introduced by considering the preceding character.*

The quantity $n\mathcal{H}_k(S)$ serves as a lower bound for the minimum number of bits attainable by any encoding of S, under the condition that the encoding of each symbol may rely only on the k symbols preceding it in S. Consistently, any compressor achieving fewer than $n\mathcal{H}_k(S)$ bits would imply the ability to compress symbols originating from the related k-th order Markov source to a level below its Shannon entropy.

Remark 2.31. *As* k *grows large (up to* $k = n - 1$*, and often sooner), the* k*-th order empirical entropy* $\mathcal{H}_k(S)$ *tends towards zero, given that most long* k*-grams appear only once, making their subsequent symbol perfectly predictable within the sequence* S*. This renders the model ineffective as a lower bound for practical compressors when* k *is very large relative to* n*. Even before reaching* $\mathcal{H}_k(S) = 0$*, achieving compression close to* $n\mathcal{H}_k(S)$ *bits becomes practically challenging for high* k *values. This is due to the necessity of storing or implicitly representing the conditional probabilities (or equivalent coding information) for all* σ^k *possible contexts, which requires significant space overhead (*$\approx \sigma^{k+1}\log n$ *bits in simple models). In theory, it is commonly assumed that* S *can be compressed up to* $n\mathcal{H}_k(S) + o(n)$ *bits for any* k *such that* $k + 1 \leqslant \alpha \log_\sigma n$ *for some constant* $0 < \alpha < 1$*. Under this condition, the overhead for storing the model (*$\sigma^{k+1}\log n \leqslant n^\alpha \log n$*) becomes asymptotically negligible compared to the compressed data size (*$o(n)$ *bits) [33].*

Definition 2.32 (Coarsely Optimal Compression Algorithm)**.** *A compression algorithm is* coarsely optimal *if, for every fixed value of* $k \geqslant 0$*, there exists a function* $f_k(n)$ *such that* $\lim_{n\to\infty} f_k(n) = 0$*, and for all sequences* S *of length* n*, the compression size achieved by the algorithm is bounded by* $n(\mathcal{H}_k(S) + f_k(n))$ *bits.*

The *Lempel-Ziv* algorithm family, particularly LZ78, serves as a prominent example of coarsely optimal compression techniques, as demonstrated by Plotnik et al. [40]. These algorithms typically rely on dictionary-based compression. However, as highlighted by Kosaraju and Manzini [27], the notion of coarse optimality does not inherently guarantee practical effectiveness across all scenarios. The additive term $n \cdot f_k(n)$ might still lead to poor performance on some sequences, especially if $f_k(n)$ converges slowly or if the sequence length n is not sufficiently large for the asymptotic behavior to dominate.

2.6 INTEGER CODING

This chapter examines methods for representing a sequence of positive integers, $S = \{x_1, x_2, \ldots, x_n\}$, potentially containing repetitions, as a compact sequence of bits [12]. The primary objective is to minimize the total bits used. A key requirement is that the resulting binary sequence must be *self-delimiting*: the concatenation of individual integer codes must be unambiguously decodable, allowing a decoder to identify the boundaries between consecutive codes.

The practical importance of efficient integer coding affects both storage space and processing speed in numerous applications. For example, *search engines* maintain large indexes mapping terms to lists of document identifiers (IDs). These *posting lists* can contain billions of integer IDs. Efficient storage is vital. A common approach involves sorting the IDs and encoding the differences (gaps) between consecutive IDs using variable-length integer codes, assigning shorter codes to smaller, more frequent gaps [12, 50]. The engineering considerations for building practical data structures based on these principles, such as providing random access capabilities, are discussed in detail in Appendix A, which describes a library developed as part of this work.

Another significant application occurs in the final stage of many *data compression algorithms*. Techniques such as LZ77, Move-to-Front (MTF), Run-Length Encoding (RLE), or Burrows-Wheeler Transform (BWT) often generate intermediate outputs as sequences of integers, where smaller values typically appear more frequently. An effective integer coding scheme is then needed to convert this intermediate sequence into a compact final bitstream [12]. Similarly, compressing natural language text might involve mapping words or characters to integer token IDs and subsequently compressing the resulting ID sequence using integer codes [12].

This chapter explores techniques for designing such variable-length, prefix-free binary representations for integer sequences, aiming for maximum space efficiency.

The central concern in this section revolves around formulating an efficient binary representation method for an indefinite sequence of integers. Our objective is to minimize bit usage while ensuring that the encoding remains prefix-free. In simpler terms, we aim to devise a binary format where the codes for individual integers can be concatenated without ambiguity, allowing the decoder to reliably identify the start and end of each integer's representation within the bit stream and thus restore it to its original uncompressed state.

2.6.1 *Unary Code*

We begin by examining the unary code, a straightforward encoding method that represents a positive integer $x \geqslant 1$[4] using x bits. It represents x as a sequence of $x-1$ zeros followed by a single one, denoted as $U(x)$. The correctness of this encoding is straightforward: the decoder identifies the end of the code upon encountering the first '1', and the value x is simply the total number of bits read.

This coding method requires x bits to represent x. While simple, this is exponentially longer than the $\lceil \log_2 x \rceil$ bits needed by its standard binary representation $B(x)$. Consequently, unary coding is efficient only for very small values of x and becomes rapidly impractical as x increases. This behavior aligns with the principles of Shannon's source coding theorem (Theorem 2.23), which suggests an ideal code length of $-\log_2 P(x)$ bits for a symbol x with probability $P(x)$. The unary code's length of x bits corresponds precisely to this ideal length if the integers follow the specific probability distribution $P(x) = 2^{-x}$ [12].

Theorem 2.33. *The unary code $U(x)$ of a positive integer x requires x bits, and it is optimal for the geometric distribution $P(x) = 2^{-x}$.*

Despite its theoretical optimality for the $P(x) = 2^{-x}$ distribution, the unary code faces practical challenges. Its implementation often involves numerous bit shifts or bit-level operations during decoding, which can be relatively slow on modern processors, especially for large x.

0	0	0	0	1

Figure 4: Unary code $U(5) = \texttt{00001}$. It uses $x = 5$ bits, consisting of $x - 1 = 4$ zeros followed by a one.

2.6.2 *Elias Codes*

While unary code is simple, its inefficiency for larger integers motivates the development of *universal codes* like those proposed by Elias [10], building upon earlier work by Levenstein. The term universal signifies that the length of the codeword for an integer x grows proportionally to its minimal binary representation, specifically as $O(\log x)$, rather than, for instance, $O(x)$ as in the unary code. Compared to the standard binary code $B(x)$ (which requires $\lceil \log_2 x \rceil$ bits but is not prefix-free), the γ and δ codes are only a constant factor longer but possess the crucial property of being prefix-free.

4 This is not a strict condition, but we will assume it for clarity.

GAMMA (γ) CODE The γ code represents a positive integer x by combining information about its magnitude (specifically, the length of its binary representation) with its actual bits. First, determine the length of the standard binary representation of x, denoted as $l = \lfloor \log_2 x \rfloor + 1$. The γ code, $\gamma(x)$, is formed by concatenating the unary code of this length, $U(l)$, with the $l-1$ least significant bits of x (i.e., $B(x)$ excluding its leading '1' bit, which is implicitly represented by the '1' in $U(l)$). The decoding process mirrors this structure: read bits until the terminating '1' of the unary part is found to determine l, then read the subsequent $l-1$ bits and prepend a '1' to reconstruct x. The total length is $|U(l)| + (l-1) = l + (l-1) = 2l-1 = 2(\lfloor \log_2 x \rfloor + 1) - 1$ bits. From Shannon's condition, it follows that this code is optimal for sources where integer probabilities decay approximately as $P(x) \approx 1/x^2$ [12].

Theorem 2.34. *The γ code of a positive integer x takes $2(\lfloor \log_2 x \rfloor + 1) - 1$ bits. It is optimal for distributions where $P(x) \propto 1/x^2$ and its length is within a factor of two of the length of the standard binary code $B(x)$.*

0	0	1	1	0

Figure 5: Elias γ code for $x = 6$. Binary $B(6) = \texttt{110}$, length $l = 3$. The code consists of $U(3) = \texttt{001}$ followed by the $l-1 = 2$ trailing bits (10). Result: $\gamma(6) = \texttt{00110}$ (5 bits).

The inefficiency in the γ code resides in the unary encoding of the length l, which can become long for large x. The δ code addresses this.

DELTA (δ) CODE The δ code improves upon γ by encoding the length $l = \lfloor \log_2 x \rfloor + 1$ more efficiently using the γ code itself. The δ code, $\delta(x)$, is constructed by first computing $\gamma(l)$, the gamma code of the length l. Then, it appends the same $l-1$ least significant bits of x used in $\gamma(x)$ (i.e., $B(x)$ without its leading '1'). Decoding involves first decoding $\gamma(l)$ to find the length l, and then reading the next $l-1$ bits to reconstruct x. The total number of bits is $|\gamma(l)| + (l-1) = (2\lfloor \log_2 l \rfloor + 1) + (l-1) = 2\lfloor \log_2 l \rfloor + l$. Asymptotically, this is approximately $\log_2 x + 2\log_2 \log_2 x + O(1)$ bits, which is only marginally longer ($1 + o(1)$ factor) than the raw binary representation $B(x)$. This code achieves optimality for distributions where $P(x) \approx 1/(x(\log_2 x)^2)$ [12].

Theorem 2.35. *The δ code of a positive integer x takes $2\lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor + \lfloor \log_2 x \rfloor + 1$ bits, approximately $\log_2 x + 2\log_2 \log_2 x$. It is optimal for distributions $P(x) \propto 1/(x(\log_2 x)^2)$ and is within a factor $1 + o(1)$ of the length of $B(x)$.*

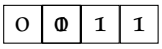


Figure 6: Elias δ code for $x = 6$. $B(6) = \texttt{110}$, length $l = 3$. First, encode $l = 3$ using γ: $\gamma(3) = \texttt{011}$. Then, append the $l - 1 = 2$ trailing bits (`10`). Result: $\delta(6) = \texttt{01110}$ (5 bits).

As with the unary code, decoding Elias codes often involves bit shifts, potentially impacting performance for very large integers compared to byte-aligned or word-aligned codes.

2.6.3 *Rice Code*

Elias codes offer universality but can be suboptimal if integers cluster around values far from powers of two. Rice codes [42] (a special case of Golomb codes) address this by introducing a parameter $k > 0$, chosen based on the expected distribution of integers. For an integer $x \geqslant 1$, the Rice code $R_k(x)$ is determined by calculating the quotient $q = \lfloor (x-1)/2^k \rfloor$ and the remainder $r = (x-1) \pmod{2^k}$. The code is then formed by concatenating the unary code of the quotient plus one, $U(q+1)$, followed by the remainder r encoded using exactly k bits (padding with leading zeros if necessary), denoted $B_k(r)$. This structure is efficient when integers often yield small quotients q, meaning they are close to (specifically, just above) multiples of 2^k.

The total number of bits required for $R_k(x)$ is $(q+1)+k$. Rice codes are optimal for geometric distributions $P(x) = p(1-p)^{x-1}$, provided the parameter k is chosen such that 2^k is close to the mean or median of the distribution (specifically, optimal when $2^k \approx -\frac{\ln 2}{\ln(1-p)} \approx 0.69 \times \text{mean}(S)$) [12, 50]. The fixed length of the remainder part facilitates faster decoding compared to Elias codes in certain implementations.

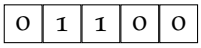


Figure 7: Rice code for $x = 13$ with parameter $k = 3$. Calculate $q = \lfloor (13-1)/2^3 \rfloor = 1$ and $r = (13-1) \pmod 8 = 4$. The code is $U(q+1) = U(2) = \texttt{01}$ followed by $r = 4$ in $k = 3$ bits, $B_3(4) = \texttt{100}$. Result: $R_3(13) = \texttt{01100}$ (5 bits).

2.6.4 *Elias-Fano Code*

The Elias-Fano representation, conceived independently by Peter Elias [10] and Robert M. Fano [11], provides an elegant and practically effective method for compressing monotonically increasing sequences of integers. A key advantage of this technique is its ability to achieve

near-optimal space occupancy, often requiring only a small overhead above the information-theoretic minimum, while simultaneously supporting efficient random access and search operations directly on the compressed form [12, 39]. This makes it highly suitable for applications such as inverted index compression in modern search engines [49, 36].

REPRESENTATION STRUCTURE Let's consider a sequence of n non-negative increasing integers:

$$S = \{s_0, s_1, \ldots, s_{n-1}\}$$

where $0 \leqslant s_0 < s_1 < \ldots < s_{n-1} < u$. The universe size is u, and we typically assume $u > n$. Each integer s_i can be represented using $b = \lceil \log_2 u \rceil$ bits. The Elias-Fano encoding strategy involves partitioning these b bits into two segments, based on a parameter l. The choice $l = \lfloor \log_2(u/n) \rfloor$ minimizes the total space requirement [10, 12] (if $u \leqslant n$, we set $l = 0$). The two parts are:

- The *lower bits*, $L(s_i)$, consisting of the l least significant bits of s_i.
- The *upper bits*, $H(s_i)$, consisting of the remaining $h = b - l$ most significant bits.

The representation then comprises two main components:

1. *Lower Bits Array (L):* This array is formed by concatenating the l-bit lower parts of all integers in the sequence: $L = L(s_0)L(s_1)\ldots L(s_{n-1})$. The total size of this array is exactly $n \cdot l$ bits.
2. *Upper Bits Bitvector (H):* This bitvector encodes the distribution of the upper bits. For each possible value j (from 0 to $2^h - 1$) that the upper bits can assume, let c_j be the number of elements s_i in S for which $H(s_i) = j$. The bitvector H is constructed by concatenating, for $j = 0, 1, \ldots, 2^h - 1$, a sequence of c_j ones followed by a single zero ($1^{c_j}0$). This structure results in a bitvector of length exactly $n + 2^h$, containing n ones (one for each element in S) and 2^h zeros (one acting as a delimiter for each possible upper bit value).

The space bound $n + 2^h \leqslant 2n$ holds if $2^h \leqslant n$, which corresponds to $u/n \leqslant n$. When u/n is small (dense sequences), l is small and h is large; when u/n is large (sparse sequences), l is large and h is small.

Theorem 2.36 (Elias-Fano Space Complexity [12, Thm 11.5], [39])**.** *The Elias-Fano encoding of a strictly increasing sequence S of n integers in the range $[0, u)$ requires $n\lfloor \log_2(u/n) \rfloor + n + 2^h$ bits, where $h = \lceil \log_2 u \rceil -$*

$\lfloor \log_2(u/n) \rfloor$. *This is upper bounded by* $n \log_2(u/n) + 2n$ *bits, which is provably less than* 2 *bits per integer above the information-theoretic lower bound. The representation can be constructed in* $O(n)$ *time.*

Figure 8 illustrates the Elias-Fano encoding for the sequence $S = \{1, 4, 7, 18, 24, 26, 30, 31\}$. In this example, we have $n = 8$ integers in the universe $u = 32$. The total number of bits needed to represent any number in the universe is $b = \lceil \log_2 32 \rceil = 5$. Since $u/n = 32/8 = 4$, the number of lower bits is $l = \lfloor \log_2 4 \rfloor = 2$, and the number of upper bits is $h = b - l = 5 - 2 = 3$. The lower bits array L is formed by concatenating the $l = 2$ least significant bits of each s_i. The upper bits bitvector H is formed by counting the occurrences c_j of each upper bit value $j \in [0, 2^h - 1)$ and concatenating $1^{c_j}0$ for each j. For instance, $H(s_0) = 0$ occurs once ($c_0 = 1$), $H(s_1) = H(s_2) = 1$ occurs twice ($c_1 = 2$), $H = 2$ and $H = 3$ never occur ($c_2 = 0, c_3 = 0$), $H(s_3) = 4$ occurs once ($c_4 = 1$), $H = 5$ never occurs ($c_5 = 0$), $H(s_4) = H(s_5) = 6$ occurs twice ($c_6 = 2$), and $H(s_6) = H(s_7) = 7$ occurs twice ($c_7 = 2$). Concatenating 1^10 (for $H = 0$), 1^20 (for $H = 1$), 1^00 (for $H = 2$), 1^00 (for $H = 3$), 1^10 (for $H = 4$), 1^00 (for $H = 5$), 1^20 (for $H = 6$), and 1^20 (for $H = 7$) yields the final bitvector H.

i	s_i	$H(s_i)$ (val, 3b)	$L(s_i)$ (val, 2b)
0	1	0 (000)	1 (01)
1	4	1 (001)	0 (00)
2	7	1 (001)	3 (11)
3	18	4 (100)	2 (10)
4	24	6 (110)	0 (00)
5	26	6 (110)	2 (10)
6	30	7 (111)	2 (10)
7	31	7 (111)	3 (11)

$L = 0100111000101011$ ($n \cdot l = 8 \times 2 = 16$ bits)
$H = 1011000100110110$ ($n + 2^h = 8 + 2^3 = 16$ bits)

Figure 8: Elias-Fano encoding example for the sequence $S = \{1, 4, 7, 18, 24, 26, 30, 31\}$ with parameters $n = 8$, $u = 32$, $l = 2$, $h = 3$. The table shows the decomposition of each s_i into its upper $H(s_i)$ and lower $L(s_i)$ bits. Below the table are the resulting concatenated lower bits array L and the upper bits bitvector H.

QUERY OPERATIONS A significant advantage of the Elias-Fano representation is its support for direct queries on the compressed data. This requires augmenting the upper bits bitvector H with auxiliary data structures that enable constant-time calculation of *rank* and *select* queries (see Section 3.1 for details). These structures typically add a $o(n)$ bits to the overall space complexity. With these in place, the core operations are:

Access(i): This operation retrieves the i-th element s_i (using 0-based indexing for i, $0 \leqslant i < n$).

1. The lower l bits, $L(s_i)$, are directly read from the array L starting at bit position $i \cdot l$.
2. The position p in H corresponding to the end of the unary code for s_i is found using $p = \texttt{select}_1(H, i+1)$. The $\texttt{select}_1$ operation finds the position of the $(i+1)$-th bit set to 1.
3. The value of the upper h bits, $H(s_i)$, is determined by counting the number of preceding zeros in H up to position p. This count is precisely $H(s_i) = p - (i+1)$, as there are $i+1$ ones and $H(s_i)$ zeros up to that point. Alternatively, $H(s_i) = \texttt{rank}_0(H, p)$.
4. The original integer is reconstructed by combining the upper and lower parts: $s_i = (H(s_i) \ll l) \vee L(s_i)$, where $\ll$ denotes the bitwise left shift and $\vee$ denotes the bitwise OR.

Since reading from L and performing rank/select on H take constant time, *Access(i)* operates in $O(1)$ time [39].

Successor(x) (or *NextGEQ(x)*): This operation finds the smallest element s_i in S such that $s_i \geqslant x$, given a query value $x \in [0, u)$.

1. Determine the upper h bits $H(x)$ and lower l bits $L(x)$ of the query value x.
2. Identify the range of indices $[p_1, p_2)$ in S corresponding to elements whose upper bits are equal to $H(x)$. The starting index p_1 is the number of elements in S with upper bits strictly less than $H(x)$. This can be found by locating the $H(x)$-th zero in H using $pos_0 = \texttt{select}_0(H, H(x)+1)$ (using 1-based index for select); then $p_1 = pos_0 - H(x)$. The ending index p_2 (exclusive) is similarly found using the $(H(x)+1)$-th zero: $pos'_0 = \texttt{select}_0(H, H(x)+1+1)$; then $p_2 = pos'_0 - (H(x)+1)$.
3. Perform a search (e.g., binary search, or linear scan if $p_2 - p_1$ is small) over the lower bits $L[p_1 \cdot l \ldots p_2 \cdot l - 1]$. The goal is to find the smallest index $k \in [p_1, p_2)$ such that the reconstructed value $(H(x) \ll l) \vee L(s_k)$ is greater than or equal to x.
4. If such a k is found within the range $[p_1, p_2)$, then s_k is the successor.
5. If no such element exists in the range (i.e., all elements with upper bits $H(x)$ are smaller than x), the successor must be the first element with upper bits greater than $H(x)$. This element is simply s_{p_2}, which can be retrieved using *Access*(p_2) (if $p_2 < n$).

The dominant cost is the search over the lower bits. Since there can be up to roughly u/n elements sharing the same upper bits in the worst case, the search step takes $O(\log(u/n))$ time using binary search. The select operations take $O(1)$ time. Thus, *Successor(x)* takes $O(1 + \log(u/n))$ time [39, 12].

Predecessor(x): Finding the largest element $s_i \leqslant x$ follows a symmetric logic, searching within the same index range $[p_1, p_2)$ identified using $H(x)$. If the search within the lower bits $L[p_1 \cdot l \dots p_2 \cdot l - 1]$ yields a suitable candidate $s_k \leqslant x$, that is the answer (specifically, the largest such s_k). If all elements in the range $[p_1, p_2)$ are greater than x, or if the range is empty ($p_1 = p_2$), the predecessor must be the last element with upper bits less than $H(x)$, which is s_{p_1-1} (if $p_1 > 0$). This can be retrieved using *Access*$(p_1 - 1)$. The time complexity is also $O(1 + \log(u/n))$.

2.7 STATISTICAL CODING

This section explores a technique called *statistical coding*: a method for compressing a sequence of symbols (*texts*) drawn from a finite alphabet Σ. The idea is to divide the process in two key stages: modeling and coding. During the modeling phase, statistical characteristics of the input sequence are analyzed to construct a model, typically estimating the probability $P(\sigma)$ for each symbol $\sigma \in \Sigma$. In the coding phase, this model is utilized to generate codewords for the symbols, which are then employed to compress the input sequence. We will focus on two popular statistical coding methods: Huffman coding and Arithmetic coding.

2.7.1 *Huffman Coding*

Compared to the methods seen in Section 2.6, Huffman Codes, introduced by David A. Huffman in his landmark 1952 paper [23], offer broader applicability. They construct optimal prefix-free codes for a given set of symbol probabilities, without requiring specific assumptions about the underlying distribution itself (beyond non-zero probabilities). This versatility makes them suitable for diverse data types, including text where symbol frequencies often lack a simple mathematical pattern.

For instance, in English text, the letter *e* is far more frequent than *z*, and simple integer codes based on alphabetical order would be highly inefficient. Huffman coding directly addresses this by assigning shorter codewords to more frequent symbols.

CONSTRUCTION OF HUFFMAN CODES The construction algorithm is greedy and builds a binary tree bottom-up. Each symbol $\sigma \in \Sigma$ initially forms a leaf node, typically weighted by its probability $P(\sigma)$ or its frequency count n_σ. The algorithm repeatedly selects the two nodes (initially leaves, later internal nodes representing merged subtrees) with the smallest current weights, merges them into a new internal node whose weight is the sum of the two merged weights, and places the two selected nodes as its children. This process continues until only one node, the root, remains.

The prefix-free code for each symbol σ is then determined by the path from the root to the leaf corresponding to σ. Conventionally, a 0 is assigned to traversing a left branch and a 1 to a right branch (or vice versa). The concatenation of these bits along the path forms the Huffman code for the symbol. More formal descriptions and variations can be found in [12, 44, 22, 9].

Example 2.37 (Huffman Coding Construction). *Let* $\Sigma = \{a, b, c, d, e\}$ *with probabilities* $P(a) = 0.25$, $P(b) = 0.25$, $P(c) = 0.2$, $P(d) = 0.15$, $P(e) = 0.15$.

1. *Initial nodes: (a: 0.25), (b: 0.25), (c: 0.2), (d: 0.15), (e: 0.15).*
2. *Merge smallest: d and e. Create node (de: 0.30). Current nodes: (a: 0.25), (b: 0.25), (c: 0.2), (de: 0.30).*
3. *Merge smallest: c and a. Create node (ca: 0.45). Current nodes: (b: 0.25), (de: 0.30), (ca: 0.45). (Note: Choosing c and a over c and b is arbitrary here, another valid tree exists).*
4. *Merge smallest: b and de. Create node (bde: 0.55). Current nodes: (ca: 0.45), (bde: 0.55).*
5. *Merge last two: ca and bde. Create root (root: 1.00).*

The resulting tree and codes (assigning 0 to left, 1 to right) are shown in Figure 9.

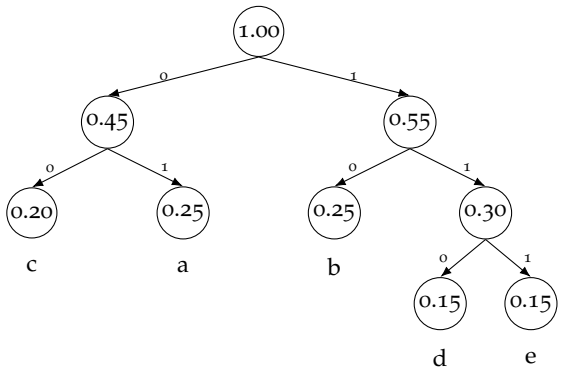


Figure 9: Huffman tree for the example probabilities ($P(a) = 0.25, P(b) = 0.25, P(c) = 0.2, P(d) = 0.15, P(e) = 0.15$). The resulting codes (0 for left, 1 for right) are: $C(a) = \texttt{01}$, $C(b) = \texttt{10}$, $C(c) = \texttt{00}$, $C(d) = \texttt{110}$, $C(e) = \texttt{111}$.

Let $L_C = \sum_{\sigma \in \Sigma} P(\sigma) \cdot l(\sigma)$ be the average codeword length for a prefix-free code C, where $l(\sigma)$ is the length of the codeword assigned to symbol σ. The Huffman coding algorithm produces a code C_H that is optimal among all possible prefix-free codes for the given probability distribution.

Theorem 2.38 (Optimality of Huffman Codes). *Let* C_H *be a Huffman code generated for a given probability distribution* P *over alphabet* Σ*. For any other prefix-free code* C' *for the same distribution, the average codeword length satisfies* $L_{C_H} \leqslant L_{C'}$.

This optimality signifies that no other uniquely decodable code assigning fixed codewords to symbols can achieve a shorter average

length. The proof typically relies on induction or an exchange argument, demonstrating that any deviation from the greedy merging strategy cannot improve the average length [12, 44, 22, 9].

The length of individual Huffman codewords can vary. In the worst case, the longest codeword might approach $|\Sigma| - 1$ bits (in a highly skewed distribution). However, a tighter bound related to the minimum probability $p_{\min}$ exists: the maximum length is $O(\log(1/p_{\min}))$ [33]. If probabilities derive from empirical frequencies in a text of length n, then $p_{\min} \geqslant 1/n$, bounding the maximum codeword length by $O(\log n)$. The encoding process itself, once the tree (or equivalent structure) is built, is typically linear in the length of the input sequence S, i.e., $O(|S|)$.

Decoding uses the Huffman Tree (or an equivalent lookup structure). Bits are read sequentially from the compressed stream, traversing the tree from the root according to the bit values (e.g., 0 for left, 1 for right) until a leaf node is reached. The symbol associated with that leaf is output, and the process restarts from the root for the next symbol. The total decoding time is proportional to the total number of bits in the compressed sequence. Since individual codes have length $O(\log n)$ in the empirical case, decoding a single symbol takes at most $O(\log n)$ bit reads and tree traversals.

While optimal among prefix codes, Huffman coding still assigns an integer number of bits to each symbol. This leads to a slight inefficiency compared to the theoretical entropy limit, as quantified by the following theorem.

Theorem 2.39. *Let $\mathcal{H} = \sum_{\sigma \in \Sigma} P(\sigma) \log_2(1/P(\sigma))$ be the entropy of a source emitting symbols from Σ according to distribution P. The average length L_H of the corresponding Huffman code is bounded by $\mathcal{H} \leqslant L_H < \mathcal{H} + 1$.*

Proof. The lower bound $\mathcal{H} \leqslant L_H$ follows directly from Shannon's source coding theorem (Theorem 2.23), which states that $\mathcal{H}$ is the minimum possible average length for any uniquely decodable code. For the upper bound, consider assigning an "ideal" codeword length $l'_\sigma = -\log_2 P(\sigma)$ to each symbol σ. As we must use an integer number of bits, let $l_\sigma = \lceil -\log_2 P(\sigma) \rceil$. Clearly, $l_\sigma < -\log_2 P(\sigma) + 1$. Summing these l_σ satisfies Kraft's inequality ($\sum 2^{-l_\sigma} \leqslant \sum 2^{-(-\log_2 P(\sigma))} = \sum P(\sigma) = 1$), guaranteeing the existence of a prefix code C' with these lengths l_σ. Its average length is $L_{C'} = \sum P(\sigma) l_\sigma < \sum P(\sigma)(-\log_2 P(\sigma) + 1) = \mathcal{H} + 1$. Since Huffman coding produces the optimal prefix code (Theorem 2.38), its average length L_H must be less than or equal to $L_{C'}$. Therefore, $L_H \leqslant L_{C'} < \mathcal{H} + 1$. Combining bounds gives $\mathcal{H} \leqslant L_H < \mathcal{H} + 1$. □

This theorem highlights that the average Huffman code length is always within one bit of the source entropy. The gap $(L_H - \mathcal{H})$ represents the inefficiency due to the constraint of using integer bit lengths for each symbol's codeword. This gap is significant only when some symbol probabilities are very high (close to 1).

2.7.2 *Arithmetic Coding*

Introduced conceptually by Peter Elias in the 1960s and later developed into practical algorithms by Rissanen [43] and Pasco [37] in the 1970s, Arithmetic Coding offers a more powerful approach to statistical compression than Huffman coding. Its key advantage lies in its ability to approach the theoretical entropy limit more closely, often achieving better compression ratios, especially when dealing with skewed probability distributions or when encoding sequences rather than individual symbols.

Unlike Huffman coding, which assigns a distinct, fixed-length (integer number of bits) prefix-free code to each symbol, Arithmetic coding represents an entire sequence of symbols as a single fraction within the unit interval $[0, 1)$. The length of the binary representation of this fraction effectively corresponds to the information content (entropy) of the entire sequence, allowing for an average representation that can use a fractional number of bits per symbol. This overcomes the inherent inefficiency of Huffman coding, which is bounded by $\mathcal{H} \leqslant L_H < \mathcal{H} + 1$. Arithmetic coding aims to achieve a compressed size very close to $n\mathcal{H}$ bits for a sequence of length n.

2.7.2.1 *Encoding and Decoding Process*

Let $S = S[1]S[2]\dots S[n]$ be the input sequence of symbols drawn from alphabet Σ, and let $P(\sigma)$ be the probability of symbol σ according to the chosen statistical model.

The core idea of the encoding process (Algorithm 1) is to progressively narrow down a sub-interval of $[0, 1)$. Initially, the interval is $[l_0, h_0) = [0, 1)$. For each symbol $S[i]$ in the sequence, the current interval $[l_{i-1}, h_{i-1})$ of size $s_{i-1} = h_{i-1} - l_{i-1}$ is partitioned into smaller sub-intervals, one for each symbol $\sigma \in \Sigma$. The size of the sub-interval for σ is proportional to its probability, $s_{i-1} \cdot P(\sigma)$. The algorithm then selects the sub-interval corresponding to the actual symbol $S[i]$ and makes it the new current interval $[l_i, h_i)$ for the next step. The cumulative probability function $C(\sigma) = \sum_{\sigma' \leqslant \sigma} P(\sigma')$ is used to efficiently calculate the start (l_i) of the correct sub-interval. After processing all n symbols, the final interval is $[l_n, h_n) = [l_n, l_n + s_n)$, where $s_n = \prod_{i=1}^{n} P(S[i])$.

Algorithm 1 Arithmetic Coding (Conceptual)

Require: Sequence $S = S[1..n]$, Probabilities $P(\sigma)$ for $\sigma \in \Sigma$
Ensure: A sub-interval $[l_n, l_n + s_n)$ uniquely identifying S.

```
Compute cumulative probabilities C(σ) = Σ_{σ'<σ} P(σ') (Note: sum over σ' < σ)
l ← 0
s ← 1                                   ▷ Initial interval [0, 1), size 1
for i = 1 to n do
    l_new ← l + s · C(S[i])             ▷ Calculate start of sub-interval
    s_new ← s · P(S[i])                 ▷ Calculate size of sub-interval
    l ← l_new
    s ← s_new
end for
return [l, l + s)                       ▷ Final interval represents the sequence
```

The final output of the encoder is not the interval itself, but rather a binary fraction x that falls within this final interval $[l_n, l_n + s_n)$ and can be represented with the fewest possible bits. Practical implementations use techniques to incrementally output bits as soon as they are determined (i.e., when the interval lies entirely within $[0, 0.5)$ or $[0.5, 1)$) and rescale the interval to maintain precision using fixed-point arithmetic [32, 12].

The decoding process (Algorithm 2) essentially reverses the encoding. The decoder needs the compressed bitstream (representing the fraction x), the same probability model $P(\sigma)$, and the original sequence length n. It starts with the interval $[0, 1)$. In each step i, it determines which symbol σ's sub-interval $[l + s \cdot C(\sigma), l + s \cdot C(\sigma) + s \cdot P(\sigma))$ contains the encoded fraction x. That symbol σ must be $S[i]$. The decoder outputs σ and updates its current interval to be this sub-interval, just as the encoder did. This is repeated n times to reconstruct the original sequence S.

2.7.2.2 *Efficiency of Arithmetic Coding*

The final interval size $s_n = \prod_{i=1}^{n} P(S[i])$ is crucial. If we use empirical probabilities $P(\sigma) = n_\sigma / n$, where n_σ is the frequency of σ in S, then $s_n = \prod_{\sigma \in \Sigma} (n_\sigma / n)^{n_\sigma}$. As noted before, the number of bits required to uniquely specify a number within an interval of size s_n is approximately $-\log_2 s_n$.

Algorithm 2 Arithmetic Decoding (Conceptual)

Require: Encoded fraction x, Probabilities $P(\sigma)$, Sequence length n.
Ensure: Original sequence $S[1..n]$.

```
Compute cumulative probabilities C(σ) = Σ_{σ'<σ} P(σ')
l ← 0
s ← 1
S ← empty sequence
for i = 1 to n do
    Find symbol σ such that l + s · C(σ) ⩽ x < l + s · (C(σ) + P(σ))
    S.append(σ)
    l_new ← l + s · C(σ)
    s_new ← s · P(σ)
    l ← l_new
    s ← s_new
    ▷ Practical decoders also update x relative to the new interval
end for
return S
```

Calculating $-\log_2 s_n$ with empirical probabilities gives:

$$\begin{aligned} -\log_2 \left(\prod_{\sigma \in \Sigma} \left(\frac{n_\sigma}{n} \right)^{n_\sigma} \right) &= -\sum_{\sigma \in \Sigma} n_\sigma \log_2 \left(\frac{n_\sigma}{n} \right) \\ &= n \sum_{\sigma \in \Sigma} \frac{n_\sigma}{n} \log_2 \left(\frac{n}{n_\sigma} \right) \\ &= n\mathcal{H} \end{aligned}$$

where $\mathcal{H}$ is the empirical (0-th order) entropy of the sequence S. This demonstrates that the *ideal* number of bits needed by arithmetic coding matches the entropy of the sequence exactly.

The connection between the final interval size s_n and the actual number of output bits deserves clarification. The encoder needs to transmit a binary representation of *some* number x that lies within the final interval $[l_n, l_n + s_n)$. To ensure the decoder can uniquely identify this interval (and thus the sequence), the chosen number x must be distinguishable from any number lying in adjacent potential intervals. This requires a certain precision. The minimum number of bits k needed to represent such an x as a dyadic fraction (i.e., a number of the form $N/2^k$) must satisfy $2^{-k} \leqslant s_n$. This condition ensures that the precision 2^{-k} is fine enough to pinpoint a unique value within the target interval of size s_n. Taking logarithms, this implies $k \geqslant -\log_2 s_n$. To guarantee that such a fraction actually *exists* within the interval, and to handle the process of incrementally outputting bits, practical arithmetic coding requires slightly more bits than the theoretical minimum $-\log_2 s_n$. A careful analysis shows that at most 2 extra bits are needed beyond the ideal $n\mathcal{H}$ [12].

Theorem 2.40. *The number of bits emitted by arithmetic coding for a sequence S of n symbols, using probabilities $P(\sigma)$ derived from the empirical frequencies within S, is at most $2 + nH$, where H is the empirical entropy of the sequence S.*

Proof. Formal proofs can be found in standard texts on information theory and data compression [12, 44, 22, 9]. The core idea, as outlined above, relates the required number of bits k to the final interval size $s_n = 2^{-nH}$ via $k \approx -\log_2 s_n = nH$. The additive constant accounts for representing a specific point within the interval. □

Remark 2.41. *Practical arithmetic coders do not use floating-point numbers due to precision issues. They employ integer arithmetic, maintaining the interval bounds $[L, H)$ as large integers within a fixed range (e.g., 16 or 32 bits). As the conceptual interval shrinks, common leading bits of L and H are output, and the integer interval is rescaled (e.g., doubled) to occupy the full range again, effectively shifting the conceptual interval. Special handling ("underflow") is needed when the interval becomes very small but straddles the midpoint (e.g., 0.5), preventing immediate output of the next bit. These implementation details ensure correctness and efficiency with fixed-precision arithmetic [32].*

3

RANK AND SELECT

In the preceding chapters, we explored foundational concepts related to data compression and information theory. We now transition to the domain of *compressed data structures*, which are designed to store data in a compact format while still permitting efficient query operations directly on the compressed representation. This paradigm often leads to what is sometimes termed *pointer-less programming*, where traditional memory pointers are eschewed in favor of structures built upon bit sequences (bitvectors) augmented with operations that implicitly handle navigation and access [12].

This chapter introduces the *bitvector* as a fundamental building block in this area. We will formally define the core operations associated with bitvectors, namely `rank` and `select`, and investigate techniques to support these operations efficiently, often in constant time, while maintaining low space overhead (succinctness). Subsequently, we will delve into methods for compressing bitvectors themselves, particularly exploiting skewed distributions of bits, and discuss practical considerations for implementing these structures effectively. We will also briefly touch upon generalizations like wavelet trees for handling larger alphabets later in the thesis. The efficient implementation of rank and select on bitvectors is crucial, as they underpin numerous advanced compressed data structures used throughout computer science [33].

3.1 BITVECTORS

Consider the following problem [12]: imagine a dictionary $\mathcal{D}$ containing n strings from an alphabet Σ. We can merge all strings in $\mathcal{D}$ into a single string $T[1, m]$, without any separators between them, where m is the total length of the dictionary. The task is to handle the following queries:

- `Read(i)`: retrieve the i-th string in $\mathcal{D}$.
- `Which_string(x)`: find the starting position of the string in T, including the character $T[x]$.

The conventional solution involves employing an array of pointers $A[1, n]$ to the strings in $\mathcal{D}$, represented by their offsets in $T[1, m]$, requiring $\Theta(n \log n)$ bits. Consequently, `Read(i)` simply returns $A[i]$,

while `Which_string(x)` involves locating the predecessor of x in A. The first operation is instantaneous, whereas the second one necessitates $O(\log n)$ time using binary search.

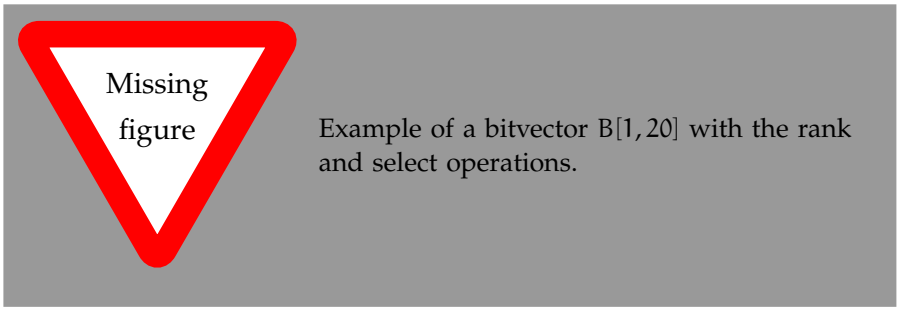
We can address the problem by employing a compressed representation of the offsets in A via a binary array $B[1, m]$ of m bits, where $B[i] = 1$ if and only if i is the starting position of a string in T. In this case then `Access_string(i)` searches for the i-th 1 in B, while `Which_string(x)` counts the number of 1s in the prefix $B[1, x]$.

In modern literature this two operations are well known as *rank* and *select* queries, respectively.

Definition 3.1 (Rank and Select)**.** *Given $B[1, n]$ a binary array of n bits (a bitvector), we define the following operations:*

- *The **rank** of an index i in B relative to a bit b is the number of occurrences of b in the prefix $B[1, i]$. We denote it as $\texttt{rank}_1(i) = \sum_{j=1}^{i} B[j]$. Similarly we can compute $\texttt{rank}_0(i) = i - \texttt{rank}_1(i)$ in constant time.*
- *The **select** of the i-th occurrence of a bit b in B is the index of the i-th occurrence of b in B. We denote it as $\texttt{select}_b(i)$. Opposite to rank, we can't derive select of 0 from select of 1 in constant time.*

Example 3.2 (Rank and Select on a plain bitvector)**.**



Example of a bitvector $B[1, 20]$ with the rank and select operations.

As stated before, bitvectors are the fundamental piece in the implementation of compressed data structures. Therefore, an efficient implementation is crucial. In the following sections, our aim is to built structures of size $o(n)$ bits that can be added on top either the bit array or the compressed representation of B to facilitate rank and select operations. We will see that will often encounter skewed distributions of 0s and 1s in B, and we will exploit this property to achieve higher order compression.

Remark 3.3. *If we try to compress bitvectors with the techniques seen in Chapter 2, we would need to encode each bit individually, requiring at least n bits.*

3.1.1 *Rank*

In their seminal paper [41] Raman et al. introduced a hierarchical succinct data structure that supports the rank operation in constant time, while only using only extra $o(n)$ bits of space. The structure is based on the idea of splitting the binary array $B[1, n]$ into big and small blocks of fixed length, and then encoding the number of bits set to 1 in each block.

More precisely, the structure is composed of three levels: in the first one we (logically) split $B[1, n]$ into blocks of size Z each, where at the beginning of each superblock we store the number (*class number*) of bits set to 1 in the corresponding block, i.e the output of the query $\mathtt{rank}_1(i)$ for i being the starting position of the block. In the second level, we split the superblocks into blocks of size z bits each[1] with the same meta-information stored at the beginning of each block. Finally the third level is a lookup table that is indexed by the small blocks and queried positions. In other words, for each possible small block and each possible position within that block, the lookup table stores the result of the $\mathtt{rank}_1$ operation. This pre-computed information allows for constant time retrieval of the $\mathtt{rank}_1$ operation results, as the result can be directly looked up in the table instead of having to be computed each time. This is the key to the efficiency of the data structure. In this way, the $i-\mathtt{th}$ block, of size Z, can be accessed as

$$B[i \cdot Z + 1, (i+1) \cdot Z]$$

while the small block j of size z in the $i-\mathtt{th}$ superblock is

$$B[i \cdot Z + j \cdot z + 1, i \cdot Z + (j+1) \cdot z] \qquad \forall j \in [0, Z/z), \forall i \in [0, n/Z)$$

We will denote with r_i and call it *absolute rank* the number of bits set to 1 in the $i-\mathtt{th}$ block, and with $r_{i,j}$ (*relative rank*) the number of bits set to 1 in the $j-\mathtt{th}$ small block of the $i-\mathtt{th}$ superblock. Figure 10 shows a visual representation of the RRR data structure.

Let's focus on the third level: the lookup table. Along with the value of the absolute and relative ranks, we also store an offset that serves as an index[2] into the table. To be precise, this table is a table of tables: one for each possible value of r_i and $r_{i,j}$. The table T is then indexed by the values of r_i and $r_{i,j}$. For every possibile value of r_i and $r_{i,j}$, the sub-table stores an array of prefix sums. Thus, since we have $\binom{Z}{z}$ possible values for r_i and $r_{i,j}$ (and consequently entries in the considered sub-table), the lookup table has a size of $\binom{Z}{z} \log Z$ bits. In Table 1 we show an example of a lookup table for the RRR data structure.

We can now state the following theorem [12]:

1 For simplicity, we assume that z divides Z

2 I we imagine that the blocks are sorted lexically, the offset is position of the block in that order

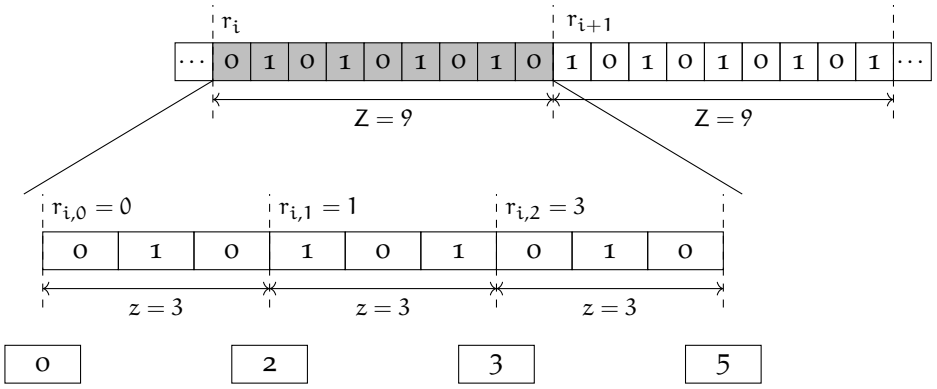


Figure 10: The RRR Rank data structure. The first level is composed of blocks of size Z, the second level of blocks of size z, and the third level is an entry of the lookup table.

block	$r_{i,0}$	$r_{i,1}$	$r_{i,2}$
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Table 1: Example of a lookup table T for the RRR data structure. The table stores the result of the rank operation for all possible small blocks with $z = 3$. The cell $T[b, r_{i,j}]$ stores the result of the rank operation for the block b inside the $i-th$ superblock and the $j-th$ small block.

Theorem 3.4. *The space occupancy of the Rank data structure is* $o(n)$ *bits, and thus it is asymptotically sublinear in the size of the binary array* $B[1, n]$*. The Rank algorithm takes constant time in the worst case, and accesses the array* B *only in read-mode*

Proof. The space occupancy of all the big blocks can be computed by multiplying the number of big blocks by the number of bits needed to store the *absolute rank* of each block. Thus, the space occupancy of the big blocks is $O(\frac{n}{Z} \log m)$ bits, since each block can store at most m bits. The same reasoning can be applied to the small blocks, which occupy $O(\frac{n}{z} \log Z)$ bits, since each block can store at most Z bits. So the space complexity is

$$O\left(\frac{n}{Z} \log m + \frac{n}{z} \log Z\right) \tag{1}$$

Let's set $Z = (\log n)^2$ and $z = 1/2 \log n$, then the space complexity becomes

$$= O\left(\frac{n}{(\log n)^2} \log m + \frac{n}{\frac{1}{2}\log n} \log(\log n)^2\right) \tag{2}$$

$$= O\left(\frac{n}{\log^2 n} \log m + \frac{n}{\log n} \log\log n\right) \tag{3}$$

$$= O\left(\frac{n \log\log n}{\log n}\right) = o(n) \tag{4}$$

□

The $o(n)$ space complexity highlighted in Theorem 3.4 signifies that the auxiliary structures consume asymptotically less space than the bitvector itself. Significant research effort has been dedicated to minimizing the constant factors hidden within this $o(n)$ term and understanding the inherent space-time tradeoffs. Works such as [21] explore techniques to further reduce this redundancy, striving for implementations that are both theoretically efficient and practically performant, often achieving space bounds closer to the information-theoretic minimum $B(n, m)$ (the space needed just to represent the bitvector) plus a smaller redundancy term, especially for certain ranges of n and m.

The current explanation of this data structure only clarifies how to respond to rank queries for indices located at the end of a block (or superblock). This can be achieved efficiently, taking constant time, either by directly accessing the value in the lookup table or by calculating the cumulative rank of preceding blocks along with the relative rank within the current block.

However, we also need to address the non-trivial case where the index i is located in the middle of a block[3]. Differently from the previous case, if we want to compute the $\texttt{rank}_1$ operation over an arbitrary position x, we would need to compute $r_i + r_{i,j} + \texttt{popcount}(B_{i,j}[1, x])$, where the last term is an operation that counts the number of bits set to 1 in the prefix $B_{i,j}[1, x]$. While the first two terms can be computed in constant time, the last term requires $O(\log n)$ time[4] in the worst case.

If the size of the small blocks doesn't fit in a single memory word, we can pre-process in our lookup table (the third level of the data structure) all the results of the `popcount` operation for all possible blocks and then use this table to answer rank queries in constant time (as shown in table 1). Let's denote this table as T and see how to use it to answer rank queries in constant time. In order to retrive the

3 For the sake of simplicity, we will assume that $B[x]$ is included in the $j-th$ small block of the $i-th$ superblock

4 It actually grows log-logarithmically with the size of the small blocks

result of $\texttt{popcount}(B_{i,j}[1, x])$ we can access the table T at the position $T[B_{i,j}, o]$. Where o is the offset of the bit $B[x]$ in $B_{i,j}$, and $B_{i,j}$. The offset o can be computed as $o = 1 + ((x - 1) \mod z)$. Thus we only need to perform three atomic operations, two memory accesses and one addition, to retrieve the result of the rank operation in constant time.

Storing this table requires $O(\sqrt{n} \log \log n)$ bits[5], which is asymptotically sub-linear in the size of the binary array $B[1, n]$ and allows the `popcount` operation in a block of $O(\log n)$ bits in constant time. Thus, if we consider the word length as $\log n$ and still maintain the $o(n)$ space occupancy stated in 3.4

3.1.2 *Select*

The select operation can be seen as the inverse of the rank operation, i.e given a binary array B and an integer i, the select operation returns the index of the i-th occurrence of a bit b in B. More formally, we have that:

$$\mathrm{rank}_c(B, \mathit{select}_c(B, i)) = i$$

Maybe talk about monoids and how rank and select are inverses of each other

The implementation of the select operation heavily relies on the three level data structure discussed before (3.1.1). The difference lies in the fact that, in this case, the bitmap B doesn't get split into blocks of fixed size, but rather into blocks of variable size that are determined by the rank of the block. We start by designing the first level of the select data structure: we split the bitmap B into blocks of size Z bitvectors each containing K bits set to 1.

Remark 3.5 (Notation and assumptions)**.** *In the following,* Z *will represent, as before, the size in bits of the big blocks containing* K *bits set to* 1*, where* $K = \log n$*. We will use always the same notation* Z *even if the size of the blocks is variable, clarifying the context in which it is used.*

Since $K \leqslant Z$, we can easily derive that space occupance of all the starting positions of the blocks $O(\frac{n}{K} \log n) = o(n)$ bits. The first step of our search in then clear: since each block contains K bits set to 1, we can find the block containing the i-th occurrence of 1 in B by computing i/K.

The second step is to find the i-th occurrence of 1 in the block. This could be done by scanning the block from the beginning and counting the number of bits set to 1 until we reach the i-th occurrence,

5 We have 2^z rows and z columns and each cell stores a value in $[0, z]$.

but this would require $O(K)$ time making it highly un-efficient for our purposes. To address this issue, we introduce the second level of the select data structure where we divide the big blocks into smaller blocks and categorize them into two types: *dense* and *sparse* blocks. A big block is considered *dense* if $Z \leqslant K^2$ and *sparse* otherwise. When dealing with a sparse block, we can store the positions of the bits set to 1 in the block in a separate array, allowing us to access the i-th occurrence of 1 in constant time. Due to it's small number of bits set to 1, we can store the positions in $O(\frac{n}{K^2} K \log n) = O(\frac{n}{\log^2 n} \log n) = o(n)$ bits.

Dealing with the dense blocks is not as straightforward as with the sparse ones. In this case, we can't afford to store the positions of the bits set to 1 in the block, as it would require too much space. We introduce then the third level of the select data structure, where we split the dense blocks into smaller blocks of length[6] z, each containing $k = (\log\log m)^2$ bits set to 1. Thus storing all the starting positions of the smalls blocks and relative beginning of the dense blocks requires $O(\frac{n}{k} \log K^2) = O(\frac{n}{(\log\log n)^2} \log\log^4 n) = o(n)$ bits[7].

The only remaining issue is to keep track of the positions of the bits set to 1 in the small blocks. We can follow the idea introduced for the big blocks and divide them into *dense* and *sparse* small blocks. The sparse small blocks are those with length less then $k^2 = (\log\log m)^4$, and we can store the positions of the bits set to 1 in the block relative to the beginning of its enclosing block in

$$O\left(\frac{n}{k^2} k \log K^2\right) = O\left(\frac{n}{(\log\log n)^2} \log\log^4 n\right) = o(n)$$

bits[8]. Following the idea of the third level of the rank data structure, we can store the positions of the bits set to 1 in the dense small blocks in a lookup table, allowing us to access the i-th occurrence of 1 in constant time. This table will store all the pre-computed results of the select operation for all possible small blocks and, since $z \leqslant k^2$, having 2^z columns and z rows, it will require $O(z2^z \log z) = o(n)$ bits[9].

Remark 3.6 (Pratical Considerations)**.** *The value* $(\log\log m)^4$ *can be very small for practical values of* m*, thus we could avoid dividing the small blocks into dense and sparse blocks and just scan the block from the beginning to find the* i*-th occurrence of* 1*.*

6 The same assumptions made before apply as well: z can vary but we will use the same notation for simplicity and clarify the context in which it is used idf necessary

7 We exploited the fact that each small block has at least length k and the length of its enclosing dense block is at most K^2.

8 We exploited the fact that each sparse small block has length $z > k^2$, thus their number is $O(\frac{n}{k^2})$. We also note that the length of the enclosing dense block is at most K^2.

9 Each cell of the table stores a value in $[0, z]$, thus the $\log z$ factor.

In algorithm 3 are outlined the steps of the $\texttt{select}_1$ (the $\texttt{select}_0$ works in the same way) algorithm, which takes as input the binary array B and an index i, and returns the index of the i-th occurrence of a bit b in B.

Algorithm 3 $\texttt{Select}_1$ Algorithm

```
function Select_1(B, i)
    j = 1 + ⌊(i−1)/K⌋                          ▷ index of big block
    B_j ← big block j
    if B_j is sparse then
        S ← array of positions of bits set to 1 in B_j
        return S[i mod K]
    else
        s_j ← starting position of B_j
        i' ← 1 + (i − 1 mod K)                 ▷ Relative select index in the block
        j' ← 1 + ⌊(i'−1)/k⌋                    ▷ index of small block
        B_{j,j'} ← small block j' in big block j
        s_{j,j'} ← starting position of B_{j,j'}
        if B_{j,j'} is sparse then
            S ← array of positions of bits set to 1 in B_{j,j'}
            return s_j + S[i' mod k]
        else
            o ← 1 + (i' − 1 mod k^2)           ▷ offset in the small block
            return s_j + s_{j,j'} + T[B_{j,j'}, o]
        end if
    end if
end function
```

As for the rank data structure, we can state the following theorem:

Theorem 3.7. *The space occupancy of the Select data structure is* $o(n)$ *bits, and thus it is asymptotically sublinear in the size of the binary array* $B[1, n]$*. The Select algorithm takes constant time in the worst case, and accesses the array* B *only in read-mode*

Proof. Follows from the previous discussion. □

For dense small blocks whose size z is very small (e.g., $z \leqslant k^2 = (\log\log m)^4$), direct scanning can indeed be faster than accessing the precomputed table T.

3.1.3 *Compressing Sparse Bitvectors with Elias-Fano*

The rank and select structures discussed before (3.1.1, 3.1.2) operate on the plain bitvector $B[1, n]$, achieving a total space occupancy of $n + o(n)$ bits. However, in many practical scenarios, the bitvector B exhibits a skewed distribution, containing significantly fewer 1s than

os (or vice-versa). Let $m = \mathrm{rank}_1(n)$ be the total number of set bits. When $m \ll n$, storing the full n-bit vector is inefficient.

In such sparse settings, we can leverage compression techniques that exploit the low density of set bits. The Elias-Fano representation, previously introduced in Section 2.6.4, provides a highly effective method for this task. Recall that Elias-Fano encodes a monotonically increasing sequence of m integers up to a maximum value n. We can represent the bitvector B by encoding the sequence of indices $\{i \mid B[i] = 1\}$.

As detailed by Vigna [49] in the context of quasi-succinct indices for information retrieval, the Elias-Fano representation achieves a space complexity of approximately $m \log_2(n/m) + O(m)$ bits. This is remarkably close to the information-theoretic lower bound for representing a subset of size m from a universe of size n, often expressed as $n\mathcal{H}_0(B) + O(m)$ bits, where $\mathcal{H}_0(B)$ is the empirical zero-order entropy of the bitvector B. The crucial advantage is that the space depends primarily on m, the number of set bits, rather than the full length n, leading to significant compression when m is small.

This compressed representation directly supports efficient operations. The $\mathrm{select}_1(i)$ operation, finding the position of the i-th set bit, can typically be implemented in constant time on average, often leveraging auxiliary pointers within the Elias-Fano structure as engineered in [49]. However, this space efficiency comes at the cost of potentially slower rank_1 and access (checking the value of $B[i]$) operations compared to the $n + o(n)$ structures. These operations usually involve decoding parts of the Elias-Fano structure and may take $O(\log(n/m))$ time or depend on the specific implementation details [33]. Therefore, Elias-Fano presents a compelling space-time trade-off, offering near-optimal compression for sparse bitvectors at the expense of rank and access time complexity. The choice between plain bitvector structures and Elias-Fano depends critically on the sparsity of the data and the required query performance profile.

3.1.4 *Practical Implementation Considerations*

While the asymptotic analysis guarantees $O(1)$ query time and $o(n)$ extra space for the rank and select structures presented earlier, achieving high performance in practice requires careful consideration of architectural factors and constant overheads hidden in the $o(n)$ term. Memory latency, cache efficiency, and instruction-level parallelism often dominate the actual running time on modern processors [12].

A particularly effective approach for optimizing rank and select implementations leverages *broadword programming* (also known as SWAR

- SIMD Within A Register). This technique treats machine registers as small parallel processors, performing operations on multiple data fields packed within a single word using standard arithmetic and logical instructions. Vigna [47] applied these techniques to rank and select queries, leading to highly efficient practical implementations.

The `rank9` structure proposed by Vigna [47] exemplifies this approach. It employs a two-level hierarchy, similar in concept to the structure in Section 3.1.1, but critically relies on broadword algorithms for the final rank computation within a machine word (specifically, sideways addition or population count). Instead of large precomputed lookup tables for small blocks, `rank9` uses carefully designed constants and bitwise operations (detailed in Algorithm 1 of [47]) to compute the rank within a 64-bit word quickly. This typically involves storing relative counts for sub-blocks (e.g., seven 9-bit counts within a 64-bit word) in the second level. The advantages include:

- Speed: Exploits fast register operations and avoids large table lookups, often outperforming other methods in practice.
- Space Efficiency: Requires relatively low space overhead, typically around 25% on top of the original bitvector B, mainly for storing the cumulative rank counts.
- Branch Avoidance: Broadword algorithms are generally branch-free, which benefits performance on modern pipelined processors by avoiding potential misprediction penalties.

Similarly, Vigna [47] developed broadword algorithms for selection within a word (Algorithm 2 in the paper). The companion `select9` structure integrates these intra-word selection capabilities with a multi-level inventory scheme. The objective of `select9` is to support high-performance selection queries, often achieving near constant-time execution, through hierarchical indexing combined with efficient broadword search for the final location. This capability involves an additional space cost, typically measured at approximately 37.5% relative to the `rank9` structure.

Furthermore, a major bottleneck in rank/select operations is often memory access latency. To mitigate this, *interleaving* the auxiliary data structures is highly recommended. For instance, storing a first-level (superblock) rank count immediately followed by its corresponding second-level (sub-block) counts increases the probability that all necessary auxiliary information for a query resides within the same cache line. This simple layout optimization can dramatically reduce cache misses compared to storing different levels of the hierarchy in separate arrays.

3.2 WAVELET TREES

Improve this introduction, just a draft

Wavelet trees, introduced in 2003 by Grossi, Gupta, and Vitter [17] are a self indexing data structure: meaning they can answer rank and select queries, while still allowing to access the text. This combination makes them particularly useful for compressed full-text indexes like the FM-index [14]. In such indexes, wavelet trees are employed to efficiently answer rank queries during the search process.

Upon closer examination, one can recognize that the wavelet tree is a slight extension of an older (1988) data structure by Chazelle [7], commonly used in Computational Geometry. This structure represents points on a two-dimensional grid, undergoing a reshuffling process to sort them by one coordinate and then by the other. Kärkkäinen (1999) [26] was the first to apply this structure to text indexing, although the concept and usage differed from Grossi et al.'s proposal four years later

Wavelet Trees can be seen in different ways: (i) as sequence representation, (ii) as a permutation of elements, and (iii) as grid point representation. Since 2003, these perspectives and their interconnections have proven valuable across diverse problem domains, extending beyond text indexing and computational geometry, where the structure originated [34, 20, 13].

An introduction to the problem

Consider a sequence $S[1, n]$ as a generalization of bitvectors whose elements $S[i]$ are drawn from an alphabet Σ[10]. We are interested in the following operations on the sequence S:

- `Access(i)`: return the i-th element of S.
- `Rank(c,i)`: return the number of occurrences of character `c` in the prefix $S[1, i]$.
- `Select(c,i)`: return the position of the i-th occurrence of character `c` in S.

However, dealing with sequences is much more complex than dealing with bitvectors (as we have seen in Section 3.1). In [33] shows how a naive approach to solve this problem would require $n\sigma + o(n\sigma)$ bits of space, which is not space-efficient. Consider σ bitvectors of length n, one for each symbol in the alphabet such that the i-th bit of the

10 The size of the alphabet varies depending on the application. For example, in DNA sequences, the alphabet is $\Sigma = \{A, C, G, T\}$ (in **??** we will focus more on this specific case), while in other case it could be of millions of characters, such as in natural language processing.

c-th bitvector is 1 if $S[i] = c$ and 0 otherwise. Then answering a rank and select query would be done by this simple transformation

$$\mathrm{rank}_c(S, i) = \mathrm{rank}_1(B_c, i)$$
$$\mathrm{select}_c(S, j) = \mathrm{select}_1(B_c, j)$$

If we try to use the techniques from Section 3.1 to compress the bitvectors, we would end up with a constant time complexity for the rank and select queries, but with the downside of a space occupancy of $n\sigma + o(n\sigma)$ bits. This is not space-efficient considering that the plain representation of the string requires $n \log \sigma + o(n)$ bits.[11]

Remark 3.8 (Notation). *From now on, let* $S[1, n] = s_1 s_2 \dots s_n$ *be a sequence of length* n *over an alphabet* Σ *that for simplicity we write as* $\Sigma = \{1, \dots, \sigma\}$. *In this way, the string can be represented using* $n\lceil \log \sigma \rceil = n \log \sigma + o(n)$ *bits in plain form.*

3.2.1 *Structure and construction*

In the beginning of this section we showed that storing one bitvector per symbol is not space-efficient. The wavelet tree is a data structure that solves this problem by using a recursive hierarchical partitioning of the alphabet. Consider the subset $[a, b] \subset [1, \dots, \sigma]$, then a wavelet tree over $[a, b]$ is a balanced binary tree with $b - a + 1$ leaves[12]. The root node v_{root} is associated with the whole sequence $S[1, n]$, and stores a bitmap $B_{v_{root}}[1, n]$ defined as follows: $B_{v_{root}}[i] = 0$ if $S[i] \leqslant (a + b)/2$ and $B_{v_{root}}[i] = 1$ otherwise. The tree is then recursively built by associating the subsequence $S_0[1, n_0]$ of elements in $[a, \dots, \lfloor (a + b)/2 \rfloor]$ to the left child of v, and the subsequence $S_1[1, n_1]$ of elements in $[\lfloor (a + b)/2 \rfloor + 1, \dots, b]$ to the right child of v. This process is repeated until the leaves are reached. In this way the left child of the root node, is a wavelet tree for $S_0[1, n_0]$ over the alphabet $[a, \dots, \lfloor (a + b)/2 \rfloor]$, and the right child is a wavelet tree for $S_1[1, n_1]$ over the alphabet $[\lfloor (a + b)/2 \rfloor + 1, \dots, b]$. [34]

Building a wavelet tree is a recursive process that takes $O(n \log \sigma)$ time by processing each node of the tree in linear time. The steps are outlined in Algorithm 4. Excluding the sequence S and the final wavelet tree T, the algorithm uses $n \log \sigma$ bits of space [13].

11 Even if we use a compressed representation of the bitvectors, the space occupancy would still have the dominant term $n\sigma$, that is at least $\Omega(n\sigma \log \log n / \log n)$ bits if we still want to support constant time rank and select queries.

12 if $a = b$ then the tree is just a leaf

13 While building the wavelet tree, we can store the sequence S on disk to free memory.

Algorithm 4 Building a wavelet tree

```
function BUILD_WT(S, n)
    T ← build(S, n, 1, σ)
    return T
end function
function BUILD(S, n, a, b)                    ▷ Takes a string S[1, n] over [a, b]
    if a = b then
        Free S
        return null
    end if
    v ← new node
    m ← ⌊(a + b)/2⌋
    z ← 0                                     ▷ number of elements in S that are ⩽ m
    for i ← 1 to n do
        if S[i] ⩽ m then
            z ← z + 1
        end if
    end for
    Allocate strings S_left[1, z] and S_right[1, n − z]
    Allocate bitmap v.B[1, n]
    z ← 0
    for i ← 1 to n do
        if S[i] ⩽ m then
            bitclear(v.B, i)                  ▷ set i-th bit of v.B to 0
            z ← z + 1
            S_left[z] ← S[i]
        else
            bitset(v.B, i)                    ▷ set i-th bit of v.B to 1
            S_right[i − z] ← S[i]
        end if
    end for
    Free S
    v.left ← build(S_left, z, a, m)
    v.right ← build(S_right, n − z, m + 1, b)
    Pre-process v.B for rank and select queries
    return v
end function
```

Remark 3.9. *The wavelet tree described has σ leaves and $\sigma - 1$ internal nodes, and the height of the tree is $\lceil \log \sigma \rceil$. The space occupancy of each level it's exactly n bits, while we have at most n bits for the last level. The total number of bits stored by the wavelet tree in then upper bounded by $n\lceil \log \sigma \rceil$ bits. [34]. However, if we also interested in storing the topology of the wavelet tree, then another $O(\sigma \log n)$, that can be critical for large alphabets. In [8, 46] are presented some techniques to build wavelet tree in a space-efficient way.*

Example 3.10 (Building a wavelet tree)**.** *Consider the sentence*

`wookies_wield_wicked_weapons_with_wisdom$`

where spaces are replaced by underscores and the sentence ends with a special character. The sorted alphabet for this example is

$$\Sigma = \{\$, \_, a, c, d, e, h, i, k, l, m, n, o, p, s, t, w\}$$

where we assume that in the lexicon the special character comes before the underscore. We now assign a bit to each symbol in the alphabet, where 0 is assigned to the first half of the alphabet and 1 to the second half.

$	_	a	c	d	e	h	i	k	l	m	n	o	p	s	t	w
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

We can now build the wavelet tree for this sequence, recursively partitioning the alphabet and assigning a bit to each symbol. The resulting wavelet tree is shown in Figure 11

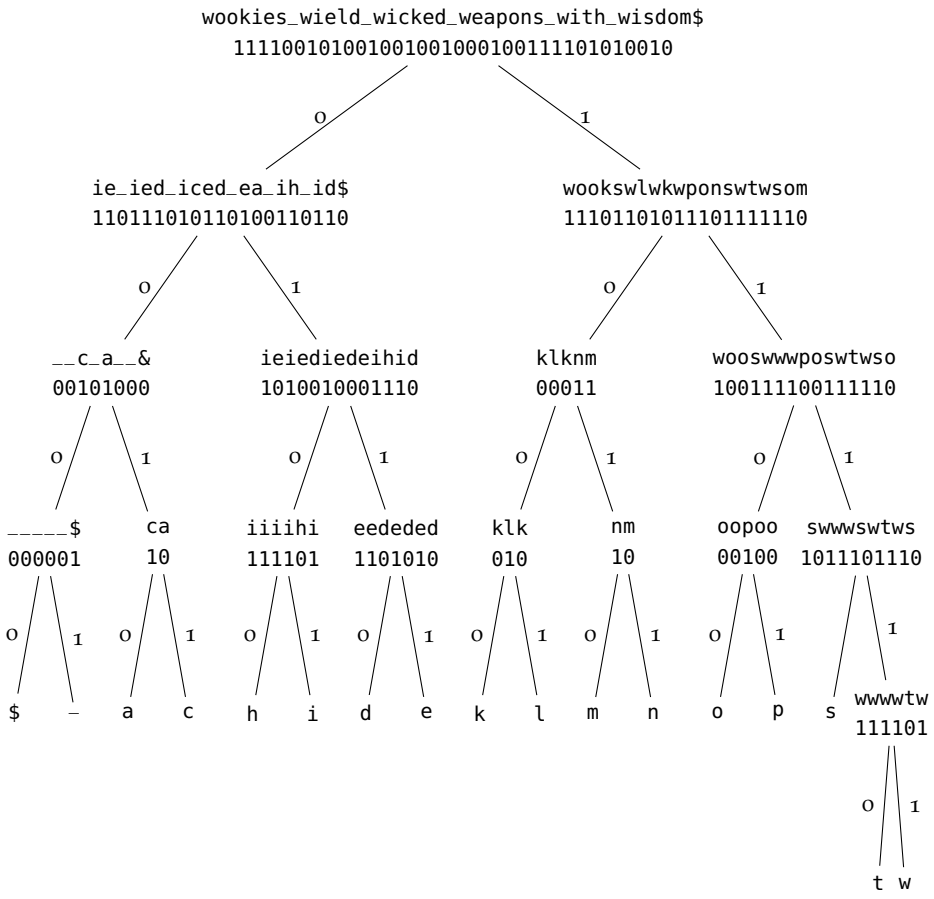


Figure 11: Wavelet tree for the sequence `wookies_wield_...`

Tracking symbols

We have seen how the wavelet tree serves as a representation for a string S, but more than that it is a succinct data structure for the string.

Thus, it takes space asymptotically close to the plain representation of the string and allows us to access the i-th symbol of the string in $O(\log \sigma)$ time.

3.2.1.1 *Access*

In algorithm 5 we show how extract the i-th symbol of the string S using a wavelet tree T, this operation is called `Access`. In order to find $S[i]$, we first look at the bitmap associated with the root node of the wavelet tree, and depending on the value of the i-th bit of the bitmap, we move to the left or right child of the root node and continue recursively. However, the problem is to determine where our i has been mapped to: if we move to the left child, then we need to find the i-th 0 in the bitmap of the left child, and if we move to the right child, then we need to find the i-th 1 in the bitmap of the right child. This is done by the $\mathtt{rank}_0$ and $\mathtt{rank}_1$ functions, respectively. We continue this process until we reach a leaf node, and then we return the value of the leaf node.

Algorithm 5 `Access` queries on a wavelet tree

```
function ACCESS(T, i)                    ▷ T is the sequence S seen as a wavelet tree
    v ← T_root                                                  ▷ start at the root node
    [a, b] ← [1, σ]
    while a ≠ b do
        if access(v.B, i) = 0 then                          ▷ i-th bit of the bitmap of v
            i ← rank_0(v.B, i)
            v ← v.left                                      ▷ move to the left child of node v
            b ← ⌊(a + b)/2⌋
        else
            i ← rank_1(v.B, i)
            v ← v.right                                     ▷ move to the right child of node v
            a ← ⌊(a + b)/2⌋ + 1
        end if
    end while
    return a
end function
```

3.2.1.2 *Select*

In addition to retrieving the i-th symbol of the string, we might also need to perform the inverse operation. That is, given a symbol's position at a leaf node, we aim to determine the position of the symbol in the string. This operation is referred to as `Select` and is outlined in Algorithm 6. Assume we start at a given leaf node v and want to find the position of the j-th occurrence of symbol c in the string. We recursively move to the left or right child of the node v: if the leaf is the right child of its parent, then we need to find the j-th 1 in the bitmap

of the parent node, and if the leaf is the left child of its parent, then we need to find the j-th 0 in the bitmap of the parent node. This is done by the $select_0$ and $select_1$ functions, respectively. We continue this process until we reach the root node, and then we return the position of the symbol in the string. As we have seen in Section 3.1, this two single operations can be solved in constant time if we use the RRR data structure [41] on each bitmap. Thus, the time complexity to perform a `Select` query on a wavelet tree is $O(\log \sigma)$.

Algorithm 6 `Select` queries on a wavelet tree

```
function SELECT_c(S, j)
    return select(T.root, 1, σ, c, j)
end function
function SELECT(v, a, b, c, j)
    if a = b then
        return j
    end if
    if c ⩽ ⌊(a + b)/2⌋ then
        j ← select(v.left, a, ⌊(a + b)/2⌋, c, j) return select_0(v.B, j)
    else
        j ← select(v.right, ⌊(a + b)/2⌋ + 1, b, c, j)
        return select_1(v.B, j)
    end if
end function
```

3.2.1.3 *Rank*

During the select algorithm, we track upwards the path from the leaf to the root. The process for solving a rank query is similar, but instead of moving from the leaf to the root, we move from the root to the leaf. Algorithm 5 also gives us the number of occurrences of a symbol S[i] in the prefix S[1, i], i.e $rank_{S[i]}(S, i)$. We now want to generalize this operation to solve any rank query $rank_c(S, i)$, where c is a symbol in the alphabet. This procedure is shown in 7.

TBD if to add: In 3.9 we mentioned that storing the topology of the wavelet tree requires $O(\sigma \log n)$ bits. This may be critical for large alphabets, and in this section we will show that this term can be removed by slightly altering the balanced wavelet tree shape. [31, 30].
Look for section 2.3 of [34]

Algorithm 7 Rank queries on a wavelet tree

```
function RANK_c(S, i)
    v ← T_root                                  ▷ start at the root node
    [a, b] ← [1, σ]
    while a ≠ b do
        if c ⩽ ⌊(a + b)/2⌋ then
            i ← rank_0(v.B, i)
            v ← v.left                          ▷ move to the left child of node v
            b ← ⌊(a + b)/2⌋
        else
            i ← rank_1(v.B, i)
            v ← v.right                         ▷ move to the right child of node v
            a ← ⌊(a + b)/2⌋ + 1
        end if
    end while
    return i
end function
```

3.2.2 *Compressed Wavelet Trees*

In order to make the wavelet tree more space efficient, we ask ourselves if we can compress this data structure. The answer is yes, and in this section we will see how a wavelet tree can be compressed to the zero-order entropy of the input string, while still being able to answer rank and select queries in $O(\log \sigma)$ time. The literature on this topic mainly focus one two different approaches: compressing the bitvectors and altering the shape of the wavelet tree itself.

3.2.2.1 *Compressing the bitvectors*

In [17] Grossi et. al showed that if the bitvectors of each single node are compressed to their zero-order entropy, then their overall space occupance is $nH_0(S)$. So if we suppose that the bitmap associated to the root node has a skewed distribution of 0s and 1s, then the zero-order compressing it yields a space of

$$n_0 \log \frac{n}{n_0} + n_1 \log \frac{n}{n_1} \tag{1}$$

where n_0 and n_1 are the number of 0s and 1s in the bitmap, respectively. This is the same as the zero-order entropy of the bitmap. The same reasoning can be applied to the bitmaps of the children of the root node, and so on. This way, one can easy prove by induction [33] that the overall space of the wavelet tree is

$$\sum_{c \in \Sigma} n_c \log(\frac{n}{n_c}) = nH_0(S) \tag{2}$$

We can now choose from the literature any zero-order entropy coding method for the bitvectors that supports rank and select queries in $O(1)$ time. Some of the most popular methods are RRR [41] that we have vastly discussed in Section 3.1, that for each bitvector of length n uses $nH_0(B) + o(n \log\log n / \log n)$ bits. In [38] the authors showed[14] that this value can be further reduced to $nH_0(B) + o(n/\log^c n)$ for any positive constant c.

3.2.2.2 *Huffman-Shaped Wavelet Trees*

Since working in practice with compressed bitvectors can be less efficient then in theory, we want a method for still obtaining nearly zero-order entropy compression, but while maintaining the bitvectors in plain form. The key idea for the compression method that we are going to analyze is that, as noted by Grossi et. al in [18], the shape of the

14 In this case, the time complexity of rank and select queries is $O(c)$.

wavelet tree has no impact on the space occupance of the structure. They proposed to use this fact to alter the shape of the tree in order to optimize the average query time. Recalling how we built an Huffman Tree in 2.7.1, we can adapt the same idea to the wavelet tree: given the frequencies f_c with which each leaf node appears in the tree, we can create an Huffman-shaped wavelet tree, obtaining an average access time of

$$\sum_{c \in \Sigma} f_c \log \frac{1}{f_c} \leqslant \log \sigma \tag{3}$$

A counter effect noted by the authors in [18] is that in the worst case, we could and up with a time complexity of $O(\log n)$, for example in the case of a very infrequent symbol.However, if we choose i uniformly at random from $[1, n]$ then the average access[15] time is

$$O(\frac{1}{n} \sum_{c} f_c |h(c)| = O(1 + H_0(S)) \tag{4}$$

That is better than the $O(\log \sigma)$ time of the original balanced wavelet tree.

Remark 3.11. *On a further note, if we bound the depth of the Huffman Tree, we can keep worst care access time to* $O(\log \sigma)$*, with extra* $O(n/\sigma)$ *bits of redundancy*

Another possibile approach following the same idea of a Huffman-shaped wavelet tree, proposed in [29], is to use the frequencies with which the symbols appear in the string. If we use this frequencies to build the Huffman Tree, we can then attach to each node v a bitvector B_v in the same way that we would do for the balanced wavelet tree (4). In this way, the bits of B_v are the bits of of the path from the root to v in the Huffman Tree, i.e the Huffman codes of the symbols. Let's see the space occupance of this structure. Consider a leaf corresponding to a symbol c, at depth $|h(c|)$ (where $h(c)$ is the bitwise Huffman code for c), representing f_c symbols. Each of this occurrences leads to a bit in each bitvector that is in the path from the root to the leaf; that is $|h(c)|$ bits. Thus, the occurrences of c lead to $f_c|h(c)|$ bits in total. If we add this values to all the leaves we obtain same number of bits outputted by the Huffman coding of the string, that is

$$\sum_{c \in \Sigma} f_c |h(c)| \leqslant n(H_0(S) + 1) \tag{5}$$

If we also want to add the space to support the rank and select queries and the tree pointers needed to navigate the tree, we arrive to a space occupance of

$$n(H_0(S) + 1) + o(n(H_0(S) + 1)) + O(n \log \sigma) \tag{6}$$

15 And also for $rank_c(S, j)$ or $select_c(S, j)$ with $c = S = [1]$

For the sake of completeness, we also mention that the shape of an Huffman tree is not the only one that can be given to a wavelet tree. In [19] Grossi and Ottaviano gave the wavelet the shape of a trie, making it possible to handle a sequence of strings.

3.2.2.3 *Higher Order Entropy Coding*

TODO and TBD: Ask Grossi how in depth to go with this section, Ferragina and Manzini in [13] give a very technical explanation of the method. While in [34] Navarro gives a more high-level explanation. Furthermore, all this methods relies on the BTW transform, which is not covered in this thesis, do I add a section on it?

3.3 DEGENERATE STRINGS

Given a finite non-empty alphabet Σ, a *string* X of length N over Σ is a sequence of N symbols from Σ. We denote with Σ^* the sets of all the strings in Σ, including the trivial one ϵ on length 0. We can now introduce the concept of a *degenerate string*, presented for the first time by Fischer and Paterson in 1974 [16] and has been used in various contexts since then [4].

Definition 3.12 (Degenerate String)**.** *A* degenerate string *is a sequence* $X = X_1 X_2 \dots X_n$*, where each* X_i *is an element of* Σ^**. We call* n *the* length *of* X *and* $N = \sum_{i=1}^{n} |X_i|$ *the* size *of* X*.*

Definition 3.13 (Balanced Degenerate String)**.** *TODO*

TODO: Add an example of a degenerate string and add a few more lines to clarify the concept. Talk about their application in bioinformatics and why the literature is interested in them.

3.3.1 *Subset-Rank and Subset-Select*

We have seen in depth the rank and select operations in Section 3.1. Where given a string S from an alphabet $[1, \sigma]$, we showed how to answer the following queries efficiently:

- $\texttt{rank}_S(c, i)$: the number of occurrences of the symbol c in $S[1..i]$.
- $\texttt{select}_S(c, i)$: the position of the i-th occurrence of the symbol c in S.

We can now extend these operations to degenerate strings. This problem was recently studied for the first time by Alanko, Biagi, Puglisi and Vuohtoniemi in [3], where their goal was to support the following queries on degenerate strings:

- $\texttt{subset-rank}_X(c, i)$: the number of sets in $X[1..i] = X_1 \dots X_i$ that contain the symbol c.
- $\texttt{subset-select}_X(c, i)$: the position of the i-th set that contains the symbol c

Example 3.14. *Let's consider the following degenerate string over the alphabet* $\Sigma = \{A, B, C, D\}$*:*

$$X = \{AAB\}\{CD\}\{\}\{A\}\{BCD\}\{\}\{C\}\{AB\}\{\}\{D\}$$

Then for example we would have

$$\texttt{subset-rank}_X(C, 6) = 2 \qquad \texttt{subset-select}_X(A, 2) = 3$$

since the symbol C *appears in order in the sets* $\{CD\}$ *and* $\{BCD\}$ *and the second set containing the symbol* A *is* $\{A\}$ *ad index* 3*.*

This type of queries are crucial for solving various problems encountered in pangenomics, the study of entire genomes across a species. In the context of de Bruijn graphs, which can be used to represent relationships between overlapping substrings in biological sequences, researchers in [3] aimed to enable efficient membership queries.

Building upon this work, in [2] they introduced the *Spectral Burrows-Wheeler Transform* (SBWT). This transform represents a string's k spectrum (the collection of all k-length substrings) as a sequence of alphabet subsets, i.e a degenerate string. The authors demonstrated that the SBWT allows for efficient de Bruijn graph representation of all k-length substrings in a string S. Membership queries within this graph can be answered using just $2k$ `subset-rank` queries on S's SBWT.

Their experiments revealed significant performance improvements compared to previously existing methods. Their approach achieves two orders of magnitude faster query times while maintaining the same space usage. Additionally, with improved space efficiency, it offers a one order of magnitude speedup.

Not anymore since in [6] they improved everything

There is the big problem that in [6] they revisit the rank-select problem on degenerate strings, introducing a new, natural parameter and reanalyzing existing reductions to rank-select on regular strings. Plugging in standard data structures, the time bounds for queries are improved exponentially while essentially matching, or improving, the space bounds. Furthermore, they provide a lower bound on space that shows that the reductions lead to succinct data structures in a wide range of cases. Their most compact structure matches the space of the most compact structure of Alanko et al. [3] while answering queries twice as fast. They also provide an implementation using modern vector processing features; it uses less than one percent more space than the most compact structure of Alanko et al. [3] while supporting queries four to seven times faster, and has competitive query time with all the remaining structures.

There is of course a naive and straightforward way to support these queries in $O(1)$. Given a degenerate string X of length n over an alphabet $\Sigma = [1, \sigma]$, for each $c \in \Sigma$ store a bitmap B_c of length n where the i-th bit is set to 1 if and only if c is in X_i. This way we can answer the queries in $O(1)$ time. In fact,

$$\texttt{subset-rank}_X(c, i) = \texttt{rank}_{B_c}(1, i)$$

However, this approach requires $O(\sigma n)$ bits of space, which is not efficient if the alphabet is large or the degenerate string is long.

3.3.1.1 *Subset Wavelet Trees*

In order to support the subset-rank and subset-select queries on degenerate strings, Alanko et al. [3] introduced the *Subset Wavelet Tree* (SWT). This data structure is built on top of the Wavelet Tree (WT) [17] (already covered in Section 3.2) and extends it to handle degenerate strings. In this section, we will first see how the SWT is constructed and then how it can be used to answer subset-rank and subset-select queries efficiently.

Structure

Imagine we have an alphabet with $\sigma = 2^n$ symbols, where n is a natural number. We can recursively construct a tree with σ levels to represent all possible subsets of this alphabet. Each node in the tree corresponds to a unique subset. The root node represents the entire alphabet. Each child node of a node v represents a subset, A_v, derived from its parent's alphabet. Let's delve deeper into this recursive process. For each node (except the root), we define its child nodes as follows: the left child represents the first half of the parent's alphabet, while the right child represents the second half.

We also introduce Q_v, a subsequence containing all subsets that include at least one element from A_v. Notably, when A_v represents the entire alphabet, Q_v also includes the empty set. In addition to representing subsets, each node v in our tree holds two bit vectors, L_v and R_v, with a length equal to the size of the corresponding subsequence Q_v.

- $L_v[i] = 1$: This indicates that the i-th subset in Qv contains at least one character from the *first half* of the alphabet associated with node v (Av).
- $R_v[i] = 1$: This indicates that the i-th subset in Q_v contains at least one character from the *second half* of the alphabet associated with node v (A_v).

We can leverage these bit vectors L_v and R_v to create a single string using the alphabet $\{0, 1, 2, 3\}$. The i-th character in this string is formed by a simple calculation:

$$S_v[i] = 2 \cdot R_v[i] + L_v[i] \tag{1}$$

This formula essentially packs the information from both bit vectors into a single digit. A value of 0 indicates the subset doesn't contain elements from either half, 1 signifies the first half only, 2 signifies the second half only, and 3 represents elements from both halves.

3.3.1.2 *Subset-Rank Queries*

The bit vectors L_v and R_v enable efficient rank queries. To find the rank of a character c at position i in the original alphabet, we perform the following steps:

1. **Traverse the Tree**: We navigate from the root node down to the leaf node where the associated alphabet A_v is the single-element set containing only character c.
2. **Prefix Length Calculation**: During this traversal, for each visited node v, we calculate the length of the prefix within the current subsequence Qv. This prefix encompasses all subsets in the original data $(X_1, \ldots, X_i)$ that include at least one character from A_v. Similar to traditional wavelet tree queries, we leverage rank queries on the L_v and R_v bit vectors to determine this prefix length.

Algorithm 8 offers pseudocode for this approach.

3.3.1.3 *Subset-Select Queries*

To answer subset-select queries, we follow a similar process to subset-rank queries.

1. **Traversal from Leaf to Root:** We begin at the leaf node where the associated alphabet A_v is the single-element set containing only character c. We then traverse back towards the root node.
2. **Updating Position** (i): As we move through each node v during traversal, we adjust the position i.
3. **Prefix Length with c Characters**: We calculate the length of the prefix within the current subsequence Qv. This prefix encompasses all subsets in the original data $(X_1, \ldots, X_i)$ that collectively contain exactly i occurrences of character c. As done before, we use select queries on the L_v and R_v bit vectors to determine this prefix length.

Algorithm 9 provides pseudocode for this approach. This method leverages the bit vectors to efficiently locate specific character occurrences without iterating through the entire dataset.

Algorithm 8 Subset-Rank Query

Require: c: character from $[1, \sigma]$, i: index
Ensure: The number of subsets X_j such that $j \leqslant i$ and $c \in X_j$

```
function SUBSET-RANK(c, i)
    v ← root
    [l, r] ← [0, σ]                          ▷ Range of alphabet indices
    while l ≠ r do
        if c ⩽ (l+r)/2 then
            r ← ⌊(l+r)/2⌋
            i ← rank_1(L_v, i)
            v ← left child of v
        else
            l ← ⌈(l+r)/2⌉
            i ← rank_1(R_v, i)
            v ← right child of v
        end if
    end while
    return i
end function
```

Space Complexity

The subset wavelet answer subset rank and select queries in logarithmic time $(O(\log \sigma))$, since at each level (there are $\log \sigma$ levels) we perform constant time operations. However, the actual number of bits

Algorithm 9 Subset-Select Query

Require: c: character from $[1, \sigma]$, i: index
Ensure: The position of the j-th subset such that the i-th $c \in X_j$

```
function SUBSET-SELECT(c, i)
    v ← leaf node with alphabet A_v = {c}
    while v ≠ root do                          ▷ Traverse from leaf to root
        u ← parent of v
        if v = left child of u then
            i ← select_1(L_v, i)
        else
            i ← select_1(R_v, i)
        end if
        v ← u
    end while
    return i
end function
```

used vary based on the data we are working with. For a general set sequence, it's typically $2n(\sigma - 1) + o(n\sigma)$ bits.

Visualizing the tree as a complete binary tree with σ leaves (not explicitly stored) helps. If the sets are full, each internal node stores $2n$ bits[16], leading to a total space complexity of $(\sigma - 1)2n$ due to the number of internal nodes and their size. However, for balanced degenerate strings 3.13 (where most sets have one element), space usage improves. Since each set element corresponds to at most one symbol per level, the total sequence length is bounded by set sizes. This translates to a space complexity of $2n \log \sigma$ bits across all $\log \sigma$ tree levels.

We can sum this up in the following theorem

Theorem 3.15 (Space Complexity of Subset Wavelet Trees)**.** *The subset wavelet tree of a balanced degenerate string takes* $2n \log \sigma + o(n \log \sigma)$ *bits of space and supports subset-rank and subset-select queries in* $O(\log \sigma)$ *time.*

The $o(n \log \sigma)$ term in the space complexity is due to...? The space required to store the bitvectors?

Rank for Base-3 and Base-4 Alphabets

The subset wavelet tree (SWT) relies on efficiently answering regular rank queries on small alphabet sequences stored within its nodes. At the root node, the sequence uses a base-4 alphabet ($\Sigma = \{0, 1, 2, 3\}$), while all other nodes use a base-3 alphabet ($\Sigma = \{0, 1, 2\}$).

However, the SWT requires a more specific operation than a standard rank query. It needs the sum of two rank queries: $rank(i, \Sigma - 1)$ and

16 Since each set goes both to the left and to the right child at each level

either $\mathtt{rank}(i, \Sigma[0])$ or $\mathtt{rank}(i, \Sigma[1])$. We call these combined operations *rank-pair queries*. Here's how we can express rank-pair queries for base-4

$$\mathtt{rankpair}(i, 1) = \mathtt{rank}(i, 1) + \mathtt{rank}(i, 3) \tag{2}$$
$$\mathtt{rankpair}(i, 2) = \mathtt{rank}(i, 2) + \mathtt{rank}(i, 3) \tag{3}$$

For base-3 alphabets, we can express rank-pair queries as follows:

$$\mathtt{rankpair}(i, 0) = \mathtt{rank}(i, 0) + \mathtt{rank}(i, 2) \tag{4}$$
$$\mathtt{rankpair}(i, 1) = \mathtt{rank}(i, 1) + \mathtt{rank}(i, 2) \tag{5}$$

In the following section we will see different methods that Alanko et al. [3] proposed to answer rank and rank-pair queries on base-3 and base-4 alphabets. They developed these structures with the a clear and very specific goal in mind: answer efficiently membership queries on Spectral Burrows Wheeler Transform (SBWT) sequences. Moreover, they focused on 3 particular genomics dataset and exploited their specific proprieties to develop the most efficient data structures for their needs. This datasets are commonly used for k-mer indexing in bioinformatics and are the following:

1. **E. coli Pangenome:** This dataset consists of 3,682 E. coli genomes downloaded in 2020. It represents a subset of assemblies from NCBI's GenBank database[17] filtered for "Escherichia coli" and downloaded before March 22nd, 2016. The complete dataset is available at[18].
2. **Human Gut Illumina Reads:** This dataset contains 17,336,887 short DNA sequences (reads) of length 502 base pairs obtained from a study on human gut microbiota[19] investigating irritable bowel syndrome and bile acid malabsorption [25].
3. **SARS-CoV-2 Genomes:** This dataset comprises 1,234,695 complete genomes of the SARS-CoV-2 virus, downloaded from NCBI datasets.

The authors shows that when the Spectral Burrows Wheeler Transform (SBWT) is built on these datasets, the resulting degenerate strings present a very skewed distribution of the alphabet symbols, with the vast majority of the sets containing only one element.

17 ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/bacteria/assembly_summary.txt
18 zenodo.org/record/6577997
19 SRA identifier ERR5035349

3.3.2 *Rank Methods for Subset Wavelet Trees*

In [3] they compare 5 methods that supports `rank` and `rank-pairs` queries on small alphabet sequences (they need it to answer those queries on the sequences stored at the nodes of the subset wavelet tree).

3.3.2.1 *Wavelet Trees*

In their paper, Alanko et. al acknowledge the wavelet tree (Section 3.2) as the current go-to method for performing rank queries on sequences with non-binary alphabets. Wavelet trees will serve as a baseline for comparison with the other data structures they will evaluate.

We have seen in Section 3.2 that we can implement the wavelet trees in different ways, different choices can influence the trade-off between space usage and query speed. The authors experimented with two options: standard bitvectors (faster but larger, Section 3.2) and RRR bitvectors (smaller but slower, as explained in Section 3.2.2.1).

They specifically used implementations from the Succinct Data Structures Library (SDSL)[20] because they are known to be the fastest available WT implementations[21].

I have a lot of doubts about this: why don't use an huffman shaped wavelet tree? Why didn't they try to adapt the wavelet tree to this type of queries?

There are multiple possible implementations of the WT in the SDSL (all immutable): Balanced wavelet (wt_blcd), Balanced wavelet tree for integer alphabets (wt_int), Wavelet matrix for integer alphabets (wm_int), Huffman-shaped wavelet tree (wt_huff), Hu-Tucker-shaped wavelet tree (wt_hutu), Run-length compressed wavelet tree (wt_rlmn), Fast select wavelet tree for integer alphabets (wt_gmr).
TODO: Check their code to see which one they used both for the standard and the RRR version.

3.3.2.2 *Scanning Rank*

The data structure comprises three layers to efficiently store and query the sequence X.

20 https://github.com/simongog/sdsl-lite

21 The SDSL library does not provide an implementation for rank-pairs queries, but they are just the sum of two rank queries, so they can be easily implemented on top of the rank queries.

- **Highest Layer:** This layer divides X into superblocks of size s and maintains a table storing $\mathtt{rank}(i, c)$ for each symbol $c \in \Sigma$ and each superblock starting at position i. We can store these counts in a table of size $\sigma n/s$ words, so that we can access the count for any superblock j at the column j/s in constant time.
- **Middle Layer:** This layer further divides X into blocks of size b, a divisor of s. For each block starting at i, precomputed counts of each symbol c are stored, specifically the occurrences of c in the range between the block's start and its enclosing superblock's start. With $s = 2^{32}$, these counts require 32 bits each.
- **Lowest Layer:** This layer directly stores the sequence, packing 32 base-4 symbols per 64-bit word. This layer occupies approximately $64 \cdot \lceil n/32 \rceil$ bits of space, where n is the length of X.

A key optimization interleaves these counts with the actual sequence data within each block. In memory, a block consists of a 2-word header storing four (precomputed) counts, followed by $b/64$ words containing the packed symbols. This interleaving let us store the lower and middle layers in a single array A of $(2n/b + n/32)$ words (i.e., $64 \cdot (2n/b + n/32)$ bits) in memory. This ensures that the data structure is cache-friendly, as the entire structure can be loaded into memory in a single read operation.

Rank Queries

To answer $\mathtt{rank}(i, c)$ queries with this structure, we follow these steps:

1. Locate the block containing position i.
2. Retrieve the count for c from its header.
3. Add this count to the corresponding count from the superblock table.
4. Scan the block's data section to count occurrences of c up to position i.

TODO: add pseudocode for the rank query.

This scanning typically involves examining whole words and possibly one partial word, which collectively contain the relevant part of the input sequence. Counting occurrences of bit patterns within these words is accelerated through bitwise operations. Notably, $\mathtt{rank-pair}$ achieves a particularly fast implementation by using a single bitwise `AND`[22] and a single `popcount`[23] operation to count relevant symbol occurrences within a word.

22 https://en.cppreference.com/w/cpp/language/operator_logical
23 https://en.cppreference.com/w/cpp/numeric/popcount

3.3.2.3 *Sequence Splitting*

The authors propose a novel data structure designed to efficiently represent sequences with skewed distributions of symbols, such as those observed in subset sequences (TODO). The key idea is to exploit the dominance of certain symbols (e.g., 1 and 2 in base-4 sequences) by splitting the sequence into multiple components.

add a picture with the distributions

Given a base-4 sequence X of length n, we use the following components:

- **Bitvector L** representing the subsequence $X_{1,2}$ (only symbols 1 and 2), where each bit indicates whether the corresponding symbol is 1 (0) or 2 (1)
- **Bitvector R** representing the subsequence $X_{0,3}$ (only symbols 0 and 3), with each bit indicating whether the corresponding symbol is 0 (0) or 3 (1)
- **Predecessor data structure P** storing the positions i where $X[i] \in \{0, 3\}$. Both L and R are equipped with rank support structures, enabling efficient rank queries.

In skewed distributions, most sets are singletons. This means $X_{1,2}$ will be long, while $X_{0,3}$, P, and R will be relatively small. This compression is especially beneficial for memory-constrained scenarios. For base-3 sequences, the bitvector L is omitted, as the predecessor structure P already stores the indices of the non-singleton sets (where $X[i] = 2$).

Rank Queries

Answering Rank queries on base-4 sequences, denoted as $\text{rank}_X(i, c)$, involves two steps:

- **Predecessor Query**: A predecessor query on P is performed for position i, returning p, the number of elements in P smaller than i (i.e., the rank of the predecessor of i in P).
- **Binary Rank Query:** The result of the predecessor query, p, is subtracted from i to obtain the appropriate index for a binary rank query. If $c \in \{1, 2\}$, the query is performed on bitvector L:

$$\text{rank}_X(i, 1) = \text{rank}_L(i - p, 0) \tag{6}$$
$$\text{rank}_X(i, 2) = \text{rank}_L(i - p, 1) \tag{7}$$

If $c \in \{0, 3\}$, the query is performed on bitvector R:

$$\text{rank}_X(i, 0) = \text{rank}_R(p, 0) \tag{8}$$
$$\text{rank}_X(i, 3) = \text{rank}_R(p, 1) \tag{9}$$

Rank queries on base-3 sequences follow the same pattern as for base-4 sequences when dealing with singletons ($x \in \{0, 1\}$). More precisely

$$\text{rank}_X(i, 0) = \text{rank}_R(i - p, 0) \tag{10}$$
$$\text{rank}_X(i, 1) = \text{rank}_L(i - p, 1) \tag{11}$$

Since there's no second binary vector, the result of the predecessor query directly gives the rank for $c = 2$.

On the other hand, with this data structure, *rank-pair* queries can be answered more efficiently than two separate rank queries. This is because the predecessor query, which computes p, needs to be performed only once for both symbols in the query.

3.3.2.4 *Generalized RRR*

For this final method, the authors present a generalization of the RRR entropy compressed bitvector (see Section 3.1.1 and [41]) to accommodate base-3 and base-4 sequences. While generalizations of RRR exist in the literature [15], this specific adaptation draws inspiration from the work of Navarro and Providel [35].

Let X be a sequence of length n from a constant-sized alphabet σ. To efficiently answer rank queries, we employ a three-level indexing structure similar to the basic binary RRR structure. We divide X into blocks of size $b = O(\log n)$ and group them into superblocks of size $B = O(\log n)$, where B is a multiple of b. For each superblock, we precompute the counts of all symbols σ up to its starting position, storing these counts using $O(\log n)$ bits each. For each block, we precompute the counts of all symbols σ within the block, storing them using $O(\log \log n)$ bits each. The total space required for these precomputed counts is $O(n\sigma \log \log n / \log n) = o(n)$ since σ is constant.

A rank query $\text{rank}(i, c)$ is then answered by: (i) Looking up the precomputed count of symbol c up to the start of the superblock containing position i. (ii) Summing the precomputed counts of symbol c in the blocks preceding index i within the superblock. (iii) Counting the occurrences of symbol c in the prefix of length $p = i \mod b$ within the block containing index i.

To determine the symbol count within a block's prefix, additional meta-information is encoded for decoding the block's symbol sequence. Then, we loop to count the occurrences of symbol c within the prefix of length $i \mod b$. The authors introduce a equivalence relation within all the possibile σ^b blocks, such that two blocks are equivalent

if and only if they contain tha same multiset of symbols[24]. We can use the equivalence classes to efficiently choose the meta-information to store: each block will store the rank r of the block within the lexicographically sorted list of all the blocks in the same equivalence class.

The `unrank` Function

The class and the lexicographically rank of the block within the class completely determine the block's content (i.e the sequence of symbols it contains). This means that we can define a function

$$\text{unrank}(r, d_0, \ldots, d_{\sigma-1})$$

that takes as input the lexicographic rank r and the precomputed counts of the symbols in the block and returns the block's content. A naive implementation of this function would require to precompute and store all the possibile answers to the function, similarly to the third layer (the lookup table) of the RRR rank data structure. However, for large values of b this is not feasible.

Given the *multinomial coefficient*

$$\binom{n}{d_0 d_1 \ldots d_{\sigma-1}} = \frac{n!}{d_0! d_1! \ldots d_{\sigma-1}!} \tag{12}$$

defined so that the value is 0 if the sum of the d_i is greater than n or if any of the d_i is negative. We can introduce the lexicographic rank of a block $c_0, c_1, \ldots, c_{b-1}$ in the equivalence class as

$$\text{lexrank}(c_0, c_1, \ldots, c_{b-1}) = \sum_{i=0}^{b-1} \sum_{j=0}^{c_i-1} \binom{b-1-i}{D_0(i) \ \ldots \ D_j(i)-1 \ \ldots \ D_{\sigma-1}(i)} \tag{13}$$

where $D_k(i)$ is the number of occurrences of the symbol k in the suffix of length i of the block[25]. This function can be computed in $O(b\sigma)$ time. The term -1 that appears in the choices of the multinomial is there to avoid counting the block itself.

Formula 13 is a process that iterates through the symbols within a block from left to right, computing different ways to choose the remaining symbols in the block using the remaining counts, with the constraint that the complete block must be lexicographically smaller than the current block. The `unrank` function essentially reverses the

24 A multiset is a generalization of the concept of a set that, unlike a set, allows multiple instances of the same element.

25 $c_i, \ldots, c_{b-1}$

`lexrank` function. This can be done by incrementally adding the multinomial terms in the inner sum until the total surpasses the target rank (r). At this point, the current symbol (j) is appended to the block's sequence, and the process moves on to the next iteration of the outer sum.

Algorithm 10 provides a pseudocode implementation of this unranking process for a sequence based on a base-4 number system. The function prints the sequence of symbols in the block with rank r among the class of blocks with symbol counts d_0, d_1, d_2, and d_3

Algorithm 10 Base-4 block *unranking* algorithm

```
function Base4BlockUnrank(r, d_0, d_1, d_2, d_3)
    b ← d_0 + d_1 + d_2 + d_3                          ▷ Block size
    s ← 0                                              ▷ Blocks counted so far
    for i = 0 to b − 1 do
        for j = 0 to 3 do                              ▷ 0 to σ − 1
            d_j ← d_j − 1
            count ← (b−1−i choose d_0, d_1, d_2, d_3)
            d_j ← d_j + 1
            if s + count > r then
                print j
                d_j ← d_j − 1
                break
            else
                s ← s + count
            end if
        end for
    end for
end function
```

3.4 IMPROVEMENTS OVER PREVIOUS METHODS

In the previous section 3.3.1.1 we have seen how in [3] the authors introduced the Subset Wavelet Tree (SWT) to solve the subset rank-select problem. Their data-structure supports both `subset-rank` and `subset-select` queries in $O(\log \sigma)$ time and uses $2(\sigma - 1)n + o(n\sigma)$ bits of space in the general case.

In this section, we will detail the significant improvements made by Bille et. al in [6] over the previous methods. The authors introduce a series of novel reductions and data structures that not only enhance the theoretical bounds but also demonstrate substantial empirical improvements.

They made three significant contributions in this context First, they introduced the parameter N and revisited the problem through reductions to the regular rank-select problem, deriving flexible complexity bounds based on existing rank-select structures, as detailed in Theorem 3.16. Second, they established a worst-case lower bound of $N \log \sigma - o(N \log \sigma)$ bits for structures supporting `subset-rank` or `subset-select`, and demonstrated that, by leveraging standard rank-select structures, their bounds often approach this lower limit while maintaining optimal query times (Theorem 3.17). Lastly, they implemented and compared their reductions to prior implementations, achieving twice the query speed of the most compact structure from [3] while maintaining comparable space usage. Additionally, they designed a vectorized structure that offers a 4-7x speedup over compact alternatives, rivaling the fastest known solutions.

Theorem 3.16 (General Upper Bound)**.** *Let X be a degenerate string of length n, size N, and containing n_0 empty sets over an alphabet $[1, \sigma]$. Let $\mathcal{D}$ denote a $\mathcal{D}_b(\ell, \sigma)$-bit data structure for a length-ℓ string over $[1, \sigma]$, supporting:*

- *rank queries in $\mathcal{D}_r(\ell, \sigma)$ time, and*
- *select queries in $\mathcal{D}_s(\ell, \sigma)$ time.*

The subset rank-select problem on X can be solved under the following conditions:

(i) ***Case*** *$n_0 = 0$: The structure requires:*

$$\mathcal{D}_b(N, \sigma) + N + o(N) \text{ bits,}$$

and supports:

$$\textit{subset-rank in } \mathcal{D}_r(N, \sigma) + O(1) \text{ time,}$$

$$\textit{subset-select in } \mathcal{D}_s(N, \sigma) + O(1) \text{ time.}$$

(ii) ***Case*** $n_0 > 0$***:*** *The bounds from case (i) apply with the following substitutions:*

$$N' = N + n_0 \quad \textit{and} \quad \sigma' = \sigma + 1.$$

(iii) ***Alternative Bound:*** *The structure uses additional* $\mathcal{B}_b(n, n_0)$ *bits of space and supports:*

subset-rank in $\mathcal{D}_r(N, \sigma) + \mathcal{B}_r(n, n_0)$ *time,*

subset-select in $\mathcal{D}_s(N, \sigma) + \mathcal{B}_s(n, n_0)$ *time.*

Here, $\mathcal{B}$ *refers to a data structure for a length-*n *bitstring containing* n_0 *1s, which:*

- *uses* $\mathcal{B}_b(n, n_0)$ *bits,*
- *supports* $\mathsf{rank}(\cdot, 1)$ *in* $\mathcal{B}_r(n, n_0)$ *time, and*
- *supports* $\mathsf{select}(\cdot, 0)$ *in* $\mathcal{B}_s(n, n_0)$ *time.*

In theorem 3.16, (i) and (ii) extend prior reductions from [2], while (iii) introduces an alternative strategy to handle empty sets using an auxiliary bitvector. By applying succinct rank-select structures to these bounds, they achieved improvements in query times without increasing space usage. For instance, substituting their structure into Theorem 3.16 (i) results in a data structure occupying $N \log \sigma + N + o(N \log \sigma + N)$ bits, supporting subset-rank in $O(\log \log \sigma)$ time and subset-select in constant time. This improves the space constant from 2 to $1 + 1/\log \sigma$ compared to Alanko et al. [3], while exponentially reducing query times.

For $n_0 > 0$, Theorem 3.16 (ii) modifies the bounds to $(N + n_0) \log(\sigma + 1) + (N + n_0) + o(n_0 \log \sigma + N \log \sigma + N + n_0)$ bits, maintaining the same improved query times. When $n_0 = o(N)$ and $\sigma = \omega(1)$, the space matches the $n_0 = 0$ case. Alternatively, Theorem 3.16(iii) allows for tailored bitvector structures sensitive to n_0.

Theorem 3.17 (Space Lower Bound)**.** *Let* X *be a degenerate string of size* N *over an alphabet* $[1, \sigma]$*. Any data structure supporting* subset-rank *or* subset-select *on* X *must use at least* $N \log \sigma - o(N \log \sigma)$ *bits in the worst case.*

In Theorem 3.17 we aim to establish a lower bound on the space required to represent X while supporting subset-rank or subset-select. Since these operations allow us to reconstruct X fully, any valid data structure must encode X completely. Our approach is to determine the number L of distinct degenerate strings possible for given parameters N and σ, and to show that distinguishing between these instances necessitates at least $\log_2 L$ bits.

Proof. Let N be sufficiently large, and let $\sigma = \omega(\log N)$. Without loss of generality, assume $\log N$ and $N/\log N$ are integers. Consider the class of degenerate strings $X_1, \ldots, X_n$ where $|X_i| = \log N$ for each i and $n = N/\log N$. The number of such strings is given by

$$\binom{\sigma}{\log N}^{N/\log N} \tag{1}$$

This is because each X_i can be formed by choosing $\log N$ characters from σ symbols, and there are n such subsets. The number of bits required to represent any degenerate string X must be at least:

$$\begin{aligned}
\log \binom{\sigma}{\log N}^{N/\log N} &= \frac{N}{\log N} \log \binom{\sigma}{\log N} \\
&\geqslant \frac{N}{\log N} \log \left(\frac{\sigma - \log N}{\log N} \right)^{\log N} \\
&= N \log \left(\frac{\sigma - \log N}{\log N} \right) \\
&= N \log \sigma - o(N \log \sigma).
\end{aligned}$$

Thus, any representation of X that supports subset-rank or subset-select must use at least $N \log \sigma - o(N \log \sigma)$ bits in the worst case, concluding the proof. □

3.4.1 *Reductions*

Let $X, \mathcal{D}, \mathcal{B}$ be as in Theorem 3.16 and consider $\mathcal{V}$ a data structure (for example the one described by Jacobson in [24]), which uses $n + o(n)$ bits for a bitstring of length n and supports rank in constant time and select in $O(1)$ time.

The reductions in Theorem 3.16 rely on the construction of two auxiliary strings S and R derived from the sets X_i. When $n_0 = 0$, each S_i is the concatenation of elements in X_i, and R_i is a single `1` followed by $|X_i| - 1$ `0`s. The global strings S and R are formed by concatenating these, appending a `1` after R_n. The data structure consists of $\mathcal{D}$ built over S and Jacobson's structure $\mathcal{V}$ over R, using $\mathcal{D}(N, \sigma) + N + o(N)$ bits. Figure 12 from [6] illustrates this reduction for $n_0 = 0$.

Queries are supported as follows: subset-rank computes the start position of S_{i+1} using select_R, then evaluates the rank in S. Conversely, subset-select determines the ith occurrence of c in S and identifies the corresponding set via rank_R. Let's consider the pratical example in Figure 12: to compute $\text{subset-rank}(2, A)$, we first compute $\text{select}_R(3, 1) = 6$. Now we know that S_2 ends at position 5, so we return $\text{rank}_S(5, A) = 2$. To compute $\text{subset-select}(2, G)$ we compute $\text{select}_S(2, G) = 8$, and

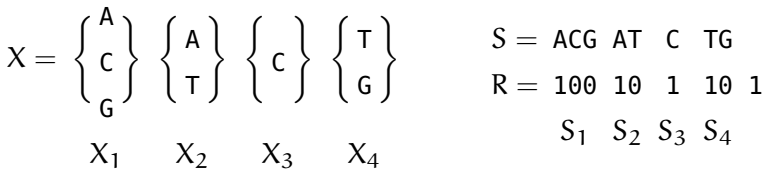


Figure 12: *Left:* A degenerate string X over the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ where $n = 4$ and $N = 8$. *Right:* The reduction from Theorem 3.16 (i) on X. White space is for illustration purposes only.

compute $\mathsf{rank}_R(8, 1) = 4$ to determine that position 8 corresponds to X_4.

Since rank and select on R are constant time, these operations achieve $\mathcal{D}_r(N, \sigma) + O(1)$ and $\mathcal{D}_s(N, \sigma) + O(1)$ time, as required by Theorem 3.16 (i).

For $n_0 \neq 0$, empty sets are replaced by singletons containing a new character $\sigma + 1$, effectively reducing the problem to the $n_0 = 0$ case with $N' = N + n_0$ and $\sigma' = \sigma + 1$. This achieves the bounds of Theorem 3.16 (ii).

ALTERNATIVE BOUND Let E be a bitvector of length n, where $E[i] = 1$ if $X_i = \emptyset$ and $E[i] = 0$ otherwise. Define X'' as the simplified string derived from X by removing all empty sets. The data structure consists in a reduction (i) applied to X'', along with a bitvector structure $\mathcal{B}$ built on E. This requires $\mathcal{D}_b(N, \sigma) + N + o(N) + \mathcal{B}_b(n, n_0)$ bits of space.

To support $\mathsf{subset\text{-}rank}_X(i, c)$, calculate $k = i - \mathsf{rank}_E(i, 1)$, which maps X_i to its corresponding set X''_k. Then, return $\mathsf{subset\text{-}rank}_{X''}(k, c)$. This operation runs in $\mathcal{B}_r(n, n_0) + \mathcal{D}_r(N, \sigma) + O(1)$ time.

To support $\mathsf{subset\text{-}select}_X(i, c)$, first determine $k = \mathsf{subset\text{-}select}_{X''}(i, c)$, and then return $\mathsf{select}_E(k, 0)$, which identifies the position of the k-th zero in E (i.e., the k-th non-empty set in X). This operation runs in $\mathcal{B}_s(n, n_0) + \mathcal{D}_s(N, \sigma) + O(1)$, achieving the stated performance bounds.

3.4.2 *Empirical Results*

The authors conducted a comprehensive evaluation of various data structures for subset rank queries on genomic datasets. Their work emphasizes both space efficiency and query performance, benchmarking methods from [3] alongside their proposed designs. The experiments utilized two primary datasets: a pangenome of 3,682 *E. coli* genomes and a human metagenome containing 17 million sequence reads. Testing was conducted in two modes: integrating the subset

rank-select structures into a k-mer query index and isolating these structures to evaluate their performance on 20 million subset-rank queries, which were randomly generated for controlled comparison. Each result reflects an average over five iterations to ensure robustness.

The study introduces the *dense-sparse decomposition* (DSD) as a novel method extending the principles of subset wavelet trees. This decomposition refines the classic split representation by categorizing sets into empty, singleton, and larger subsets, with optimized handling for each category. The authors incorporated advanced rank-select techniques into this framework, including SIMD-based optimizations. Compared to subset wavelet trees and their modern implementations, the DSD structures consistently demonstrated significant improvements. For example, the SIMD-enhanced DSD achieved query times that were 4 to 7 times faster than Concat (ef), a competitive baseline, while maintaining similar space efficiency. Furthermore, the DSD (rrr) variant provided comparable space usage to the compact Concat (ef) structure but offered double the query speed.

The experiments revealed nuanced trade-offs between space and time across all tested structures. While subset wavelet trees, such as Split (ef) and Split (rrr), remain strong contenders, the authors' DSD approach often outperformed them in both dimensions. The DSD (scan) structure, for example, provided a competitive balance, achieving space usage close to entropy bounds while delivering faster query times than comparable subset wavelet tree configurations. The SIMD-enhanced DSD design was particularly noteworthy, achieving near-optimal space efficiency with remarkable query performance.

4

SUCCINCT DAGS FOR EFFICIENT PREFIX QUERIES

A

ENGINEERING A COMPRESSED INTEGER VECTOR

This appendix outlines the design principles and engineering considerations behind *compressed-intvec*, a software library that we developed for the efficient storage and retrieval of integer sequences [28]. The library leverages the variable-length integer coding techniques discussed in Section 2.6 to achieve significant space savings compared to standard fixed-width representations, while providing mechanisms for acceptably fast data access.

MOTIVATION AND BITSTREAM ABSTRACTION Storing sequences of integers, particularly when many values are small or follow predictable patterns, using standard fixed-width types (such as 64-bit integers) is inherently wasteful. Variable-length integer codes, such as Unary, Gamma, Delta, and Rice codes (Section 2.6), offer a solution by representing integers using a number of bits closer to their information content, assigning shorter codes to smaller or more frequent values.

However, these codes produce binary representations of varying lengths, not necessarily aligned to byte or machine word boundaries. Therefore, storing a sequence of integers compressed with these methods requires packing their binary codes contiguously into a single, undifferentiated sequence of bits, commonly referred to as a bitstream. This necessitates the use of specialized bitstream reading and writing capabilities, abstracting away the complexities of bit-level manipulation. The implementation described here relies on the `dsi-bitstream` library for this purpose [48], ensuring that the variable-length codes can be written to and read from memory efficiently. The fundamental requirement for correctly decoding the concatenated sequence is the prefix-free (self-delimiting) property of the chosen integer code, which guarantees that the end of one codeword can be determined without ambiguity before reading the next.

ADDRESSING RANDOM ACCESS VIA SAMPLING While bitstream concatenation enables compression, it introduces a significant challenge for random access. Retrieving the i-th integer from the original sequence cannot be done by calculating a simple memory offset, as the bit lengths of preceding elements are variable. A naive approach

would require sequentially decoding the first i integers from the beginning of the bitstream, resulting in an unacceptable $O(i)$ access time.

To provide efficient random access, the *compressed-intvec* library employs a *sampling* technique. During the encoding phase, the absolute starting bit position of every k-th integer in the sequence is recorded. These positions, or samples, are stored in an auxiliary data structure, typically a simple array. The value k is a user-configurable sampling parameter that dictates a trade-off between random access speed and the memory overhead incurred by storing the samples.

To retrieve the i-th integer, the library first determines the index of the sample corresponding to the block containing the i-th element: $\texttt{sample\_idx} = \lfloor i/k \rfloor$. It retrieves the bit offset `start_bit` associated with this sample. The bitstream reader can then jump directly to this position. From `start_bit`, the decoder only needs to perform $i \pmod k$ sequential decoding operations to reach and return the desired i-th integer. If k is considered a constant (e.g., 32 or 64), this reduces the expected time complexity for random access to $O(1)$[1]. The space overhead for the samples is approximately $O((n/k)\log(\text{total\_bits}))$, which is generally sub-linear in the size of the compressed data for practical values of k. The choice of k allows tuning the balance between faster access (smaller k) and lower memory usage (larger k).

CODEC FLEXIBILITY AND DATA DISTRIBUTION The theoretical discussion in Section 2.6 highlights that the efficiency of different integer codes is highly dependent on the statistical distribution of the integers being compressed. For example, Gamma code is suited for distributions decaying roughly as $1/x^2$, Rice codes excel for geometrically distributed values (especially when integers cluster around multiples of 2^k), and minimal binary coding is optimal for uniformly distributed data within a known range [12].

Recognizing this dependency, the *compressed-intvec* library is designed with flexibility in mind. It employs generic programming paradigms (specifically, Rust traits) to allow the user to select the most appropriate integer coding scheme (*codec*) for their specific data distribution at compile time. The library provides implementations for several standard codecs, including Gamma, Delta, Rice, and Minimal Binary [28]. Some codecs, like Rice or Minimal Binary, require additional parameters (the Rice parameter k or the universe upper bound u, respectively), which are also managed by the data structure. Selecting

1 The underlying bitstream operations and single-integer decoding are sufficiently fast to assume that

a codec poorly matched to the data can significantly degrade compression performance, potentially even increasing the storage size compared to an uncompressed representation [28]. This flexibility is therefore crucial for achieving optimal results in practice.

BIBLIOGRAPHY

[1] N. Vereshchagin A. Shen V. A. Uspensky. *Kolmogorov Complexity and Algorithmic Randomness*. Mathematical Surveys and Monographs. Amer Mathematical Society, 2017.

[2] Jarno N Alanko, Simon J Puglisi, and Jaakko Vuohtoniemi. "Small searchable κ-spectra via subset rank queries on the spectral burrows-wheeler transform." In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*. SIAM. 2023, pp. 225–236.

[3] Jarno N. Alanko et al. "Subset Wavelet Trees." In: *21st International Symposium on Experimental Algorithms (SEA 2023)*. Ed. by Loukas Georgiadis. Vol. 265. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 4:1–4:14.

[4] Mai Alzamel et al. "Degenerate string comparison and applications." In: *WABI 2018-18th Workshop on Algorithms in Bioinformatics*. Vol. 113. 2018, pp. 1–14.

[5] David Benoit et al. "Representing trees of higher degree." In: *Algorithmica* 43 (2005), pp. 275–292.

[6] Philip Bille, Inge Li Gørtz, and Tord Stordalen. *Rank and Select on Degenerate Strings*. 2023.

[7] Bernard Chazelle. "A Functional Approach to Data Structures and Its Use in Multidimensional Searching." In: *SIAM Journal on Computing* 17.3 (1988), pp. 427–462.

[8] Francisco Claude, Patrick K Nicholson, and Diego Seco. "Space efficient wavelet tree construction." In: *String Processing and Information Retrieval: 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings 18*. Springer. 2011, pp. 185–196.

[9] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley, 2012.

[10] P. Elias. "Universal codeword sets and representations of the integers." In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 194–203.

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

[12] P. Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.

[13] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. "The myriad virtues of Wavelet Trees." In: *Information and Computation* 207.8 (2009), pp. 849–866.

[14] Paolo Ferragina and Giovanni Manzini. "Opportunistic data structures with applications." In: *Proceedings 41st annual symposium on foundations of computer science*. IEEE. 2000, pp. 390–398.

[15] Paolo Ferragina et al. "Compressed representations of sequences and full-text indexes." In: *ACM Transactions on Algorithms (TALG)* 3.2 (2007), 20–es.

[16] Michael J Fischer and Michael S Paterson. "String-matching and other products." In: (1974).

[17] Roberto Grossi, Ankur Gupta, and Jeffrey Vitter. "High-Order Entropy-Compressed Text Indexes." In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms* (Nov. 2002).

[18] Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. "When indexing equals compression: experiments with compressing suffix arrays and applications." In: *SODA*. Vol. 4. 2004, pp. 636–645.

[19] Roberto Grossi and Giuseppe Ottaviano. "The wavelet trie: maintaining an indexed sequence of strings in compressed space." In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 203–214.

[20] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. "Wavelet Trees: From Theory to Practice." In: *2011 First International Conference on Data Compression, Communications and Processing*. 2011, pp. 210–221.

[21] Roberto Grossi et al. *More Haste, Less Waste: Lowering the Redundancy in Fully Indexable Dictionaries*. 2009.

[22] T.S. Han and K. Kobayashi. *Mathematics of Information and Coding*. Fields Institute Monographs. American Mathematical Society, 2002.

[23] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[24] G. Jacobson. "Space-efficient static trees and graphs." In: *30th Annual Symposium on Foundations of Computer Science*. 1989, pp. 549–554.

[25] Ian B Jeffery et al. "Differences in fecal microbiomes and metabolomes of people with vs without irritable bowel syndrome and bile acid malabsorption." In: *Gastroenterology* 158.4 (2020), pp. 1016–1028.

[26] J Kärkkäinen. "Repetition-based text indexing." PhD thesis. Ph. D. thesis, Department of Computer Science, University of Helsinki, Finland, 1999.

[27] S Rao Kosaraju and Giovanni Manzini. "Compression of low entropy strings with Lempel–Ziv algorithms." In: *SIAM Journal on Computing* 29.3 (2000), pp. 893–911.

[28] Luca Lombardo. *compressed-intvec: Compressed Integer Vector Library*. Rust Crate. URL: https://crates.io/crates/compressed-intvec.

[29] Veli Mäkinen and Gonzalo Navarro. "New search algorithms and time/space tradeoffs for succinct suffix arrays." In: *Technical rep. C-2004-20 (April). University of Helsinki, Helsinki, Finland* (2004).

[30] Veli Mäkinen and Gonzalo Navarro. "Position-Restricted Substring Searching." In: *LATIN 2006: Theoretical Informatics*. Ed. by José R. Correa, Alejandro Hevia, and Marcos Kiwi. Springer Berlin Heidelberg, 2006, pp. 703–714.

[31] Veli Mäkinen and Gonzalo Navarro. "Rank and select revisited and extended." In: *Theoretical Computer Science* 387.3 (2007), pp. 332–347.

[32] Alistair Moffat, Radford M Neal, and Ian H Witten. "Arithmetic coding revisited." In: *ACM Transactions on Information Systems (TOIS)* 16.3 (1998), pp. 256–294.

[33] G. Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[34] Gonzalo Navarro. "Wavelet trees for all." In: *Journal of Discrete Algorithms* 25 (2014). 23rd Annual Symposium on Combinatorial Pattern Matching, pp. 2–20. ISSN: 1570-8667.

[35] Gonzalo Navarro and Eliana Providel. "Fast, small, simple rank/select on bitmaps." In: *International Symposium on Experimental Algorithms*. Springer. 2012, pp. 295–306.

[36] Giuseppe Ottaviano and Rossano Venturini. "Partitioned Elias-Fano indexes." In: *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*. SIGIR '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 273–282. ISBN: 9781450322577. DOI: 10.1145/2600428.2609615.

[37] Richard Clark Pasco. "Source coding algorithms for fast data compression." PhD thesis. Stanford University CA, 1976.

[38] Mihai Patrascu. "Succincter." In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. 2008, pp. 305–313.

[39] Giulio Ermanno Pibiri and Rossano Venturini. "Dynamic Elias-Fano Representation." In: *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*. Ed. by Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter. Vol. 78. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 30:1–30:14.

[40] Eli Plotnik, Marcelo J Weinberger, and Jacob Ziv. "Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel-Ziv algorithm." In: *IEEE transactions on information theory* 38.1 (1992), pp. 66–72.

[41] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets." In: *ACM Transactions on Algorithms* 3.4 (Nov. 2007), p. 43.

[42] Robert F Rice. *Some practical universal noiseless coding techniques*. Tech. rep. 1979.

[43] Jorma J Rissanen. "Generalized Kraft inequality and arithmetic coding." In: *IBM Journal of research and development* 20.3 (1976), pp. 198–203.

[44] K. Sayood. *Lossless Compression Handbook*. Communications, Networking and Multimedia. Elsevier Science, 2002, pp. 55–64.

[45] C. E. Shannon. "A mathematical theory of communication." In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[46] German Tischler. "On wavelet tree construction." In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2011, pp. 208–218.

[47] Sebastiano Vigna. "Broadword implementation of rank/select queries." In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2008, pp. 154–168.

[48] Sebastiano Vigna et al. *Dsi-bitstream: Bitstream readers/writers for the DSI utilities*. Rust Crate. URL: https://crates.io/crates/dsi-bitstream.

[49] Sebastiano Vigna. "Quasi-succinct indices." In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. WSDM '13. Association for Computing Machinery, 2013, pp. 83–92.

[50] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.