Algorithms for Massive Data

Academic Year 2022/2023

Nicola Aggio 880008

Index

1 Introduction

The goal of this course is to introduce algorithmic techniques for dealing with massive data: data so large that it does not fit in the computer's memory. Broadly speaking, there are **two** main **solutions** to deal with massive data: **(lossless) compressed data structures** and **(lossy) data sketches**.
A **compressed** data structure supports **fast queries** on the data and uses a **space proportional** to the **compressed data**. This solution is typically **lossless**: the representation allows to **fully reconstruct the original data**. Here we exploit the fact that, in some applications, data is extremely **redundant** and can be considerably reduced in size (even by orders of magnitude) without losing any information by exploiting this redundancy. Interestingly it turns out that, usually, **computation** on **compressed data** can be performed **faster** than on the original (uncompressed) data: the field of compressed data structures exploits the natural duality between compression and computation to achieve this goal. The main results we will discuss in the course, **compressed dictionaries** and **compressed text indexes**, find important **applications** in information retrieval. They allow to pre-process a large collection of documents into a compressed data structure that allows locating substrings very quickly, without decompressing the data. These techniques today stand at the **core** of **modern search engines** and sequence-mapping algorithms in computational biology. Importantly, **lossless techniques cannot break the information-theoretic lower bound** for representing data. For example, since the number of subsets of cardinality n of $\{1, .., u\}$ is $\binom{u}{n}$, the information-theoretic lower bound for storing such a subset is $\log_2 \binom{u}{n} = n \log(u/n) + O(n)$ bits. No lossless data structure can use asymptotically less space than this bound for all such subsets.
The second solution is to resort to **lossy compression** and **throw away** some **features** of the data in order to **reduce its size**, usually **breaking the information-theoretic lower bound**. We will show that any **set** of cardinality n can be stored with a **filter data structure** in just $O(n)$ bits bits, provided that we accept a small probability that membership queries fail. Moreover, we'll see how to **shrink even more the size of our data** while still being able to compute useful information on it. The most important concept here is **sketching**. Using clever randomized algorithms, we show how to **reduce large data sets** to **sub-linear representations**: for example, a **set** of cardinality n can be stored in just $O(\text{poly}\log(n))$ bits using a data sketch supporting useful queries such as set similarity and cardinality estimation (approximately and with a bounded probability of error). **Sketches** can be **computed off-line** in order to reduce the dataset's size and/or speed up the computation of distances, or on-line on data streams, where data is thrown away as soon as it arrives (only data sketches are kept).

1.1 Asymptotic notation

We now recall some basic informations about the asymptotic notation.

- $f(n) \in O(g(n))$, which can be intuitively thought as $f(n) \leq g(n)$, but the definition is: $\exists c, n_0 : \forall n \geq n_0, f(n) \leq cg(n)$;
- $f(n) \in \Omega(g(n)) = f(n) \geq g(n) \iff g(n) \in O(f(n))$;
- $f(n) \in \Theta(g(n)) = f(n) = g(n) \iff f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n)$;
- $f(n) \in o(g(n) = f(n) << g(n) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, e.g. $\frac{n \log\log n}{\log n}$;
- $f(n) \in \omega(g(n)) = f(n) >> g(n) \iff g(n) \in o(f(n))$.

2 Entropy and coding

In this chapter we cover some minimal notions of Information Theory and Data Compression required to understand the compact data structures we present in the course.
In broad terms, the **entropy** is the **minimum number of bits** needed to **unambiguously identify** an object from a **set**. The entropy is then a **lower bound** to the **space** used by the **compressed representation** of an object. The holy grail of compressed data structures is to use essentially the space needed to identify the objects, but choosing a representation that makes it easy to answer queries on them.

2.1 Worst-case entropy

The most basic notion of entropy is that it is the **minimum number of bits** required by identifiers, called **codes**, if we assign a **unique code to each element of a set** U and all the codes are of the **same length**. This is called the **worst-case entropy** of U and is denoted $H_{\text{wc}}(U)$. It is easy to see that:

$$H_{\text{wc}}(U) = \log |U| \text{ bits}$$

Why "worst-case"? If we used codes of length $l < H_{\text{wc}}(U)$, we would have that the number of distinct codes would be only $2^l < 2^{H_{\text{wc}}(U)} = 2^{\log |U|} = |U|$ (2^l comes from the fact that the alphabet is binary, thus 2^l distinct codes exist), which would be **insufficient** for giving a **distinct code** to each element in U. Therefore, if all the codes have the same length, this length must be at least $\lceil H_{\text{wc}}(U) \rceil$ bits. If they are of different lengths, then the longest ones still must use at least $\lceil H_{\text{wc}}(U) \rceil$ bits. This explains the adjective "worst-case": it is a lower bound to the longest code we assign to an element of U. If we go below this bound, we get a lossy representation, i.e. we loose some information.

Example. *If we consider $U = \Sigma^n$, i.e. the set of strings of size n from the alphabet $\Sigma = \{0, 1\}$, then:*

$$H_{wc}(U) = H_{wc}(\Sigma^n) = \log |\Sigma^n|$$

but if $\Sigma = \{0, 1\}$, then $|\Sigma^n| = 2^n$, so, in the general case, $|\Sigma^n| = |\Sigma|^n$, so

$$H_{wc}(\Sigma^n) = \log |\Sigma|^n = n \log |\Sigma|$$

Example. *If we consider the set U as the set of bitvectors of length n with m one's, then:*

$$H_{wc}(U) = \log \binom{n}{m} = m \log \frac{n}{m} + O(m)$$

Let's consider the worst-case entropy of the set of all the general ordinal trees of n nodes, T_n.
In an **ordinal tree**, each **node** has an **arbitrary number of children** and distinguishes their order. It is known that the number of general ordinal trees is:

$$|T_n| = (n-1)\text{-th catalan number } = \frac{1}{n} \left(\frac{2n-2}{n-1} \right)$$

Thus

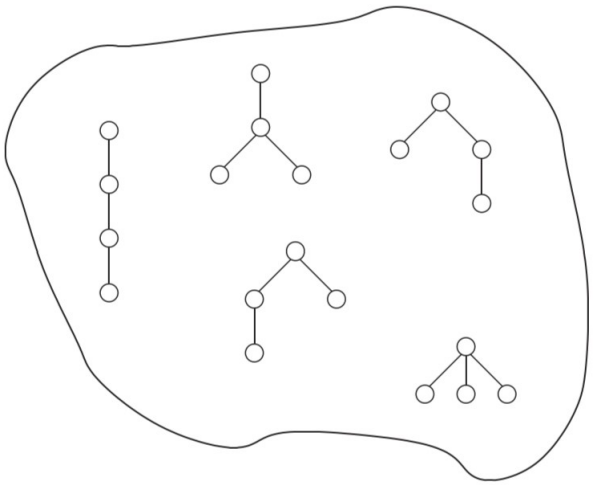


Figure 1: T_4, the general ordinal trees of 4 nodes

$$H_{\text{wc}}(U) = H_{\text{wc}}(T_n) = \log |T_n| = 2n - \Theta(\log n)$$

is the **minimum number of bits** into which any general ordinal tree of n nodes can be **encoded**. Unlike the examples of bit sequences and strings, the **classical representations** of trees are **very far** from this number; actually they use at least n pointers. Since such pointers must distinguish among the n different nodes, by the same reasoning on the entropy they must use at least $\log n$ bits, where n represents the number of pointers, while $\log n$ the space for each pointer. Therefore, classical representations of trees use at least $n \log n$ bits, whereas $2n$ bits should be sufficient to represent any tree of n nodes. Indeed, it is not very difficult to encode a general tree using $2n$ bits: traverse the tree in depth-first-order, writing a 1 upon arriving at a node, and a 0 when leaving it in the recursion. It is an easy exercise to see that one can recover the tree from the resulting stream of $2n$ 1s and 0s.

2.2 Shannon entropy

In classical Information Theory, one assumes that there is an **infinite source** that emits **elements** $x \in \Sigma$ with **probabilities** $Pr(x)$. By using codes of varying lengths of $l(x)$ bits (shorter codes for the more probable elements), one can reduce the **average length** of the codes:

$$\bar{l} = \sum_{x \in \Sigma} Pr(x) l(x)$$

Then, a natural question is how to assign codes that **minimize** the **average code length**. A fundamental result of Information Theory is that the **minimum possible average code length** for codes that can be univocally decoded is:

$$H(Pr) = \sum_{x \in \Sigma} Pr(x) \log \frac{1}{Pr(x)}$$

, which is called the **Shannon entropy** of the probability distribution $Pr : \Sigma \to [0.0, 1.0]$. We have three properties:

1. The measure $H(Pr)$ can be interpreted as **how many bits of information are contained in each element emitted by the source**. The **more biased** the **probability**

distribution, the **more "predictable"**, or the less "surprising," the **output** of the source is, and the **less information** is carried by its elements;

2. The formula of $H(Pr)$ also suggests that an **optimal code** for x should be $\log \frac{1}{Pr(x)}$ bits long, that is, assigning a length $l(x)$ which is inversely proportional to the probability of the element x. Actually, it can be proved that no other choice of code lengths reaches the optimal average code length $H(Pr)$. This means that, as anticipated, **more probable elements should receive shorter codes**. Note that, no matter how we assign the codes to reduce the average code length, the length of the longest code is still at least $\lceil H_{\text{wc}}(U) \rceil$;
3. $H(Pr)$ is maximized when the distribution is uniform, i.e. when $Pr(x) = Pr(y) \quad \forall x, y \in \Sigma$, and $Pr(x) = \frac{1}{|\Sigma|}$. If we plug this quantity in the formula of $H(Pr)$, we get:

$$H(Pr) = \sum_{x \in \Sigma} Pr(x) \log \frac{1}{\frac{1}{|\Sigma|}} = 1 \cdot \log |\Sigma| = H_{wc}(U)$$

Intuitively, if the probabilities are uniform, the code can't exploit assigning shorter codes to elements with higher probability, so the average code length is the worst possible, and in particular, it is equal to the worst-case entropy.

Example. *Assume that the 5 trees of T_4 arise with probabilities $\{0.6, 0.3, 0.05, 0.025, 0.025\}$. Their Shannon entropy is then $H = 0.6 \log \frac{1}{0.6} + 0.3 \log \frac{1}{0.3} + 0.05 \log \frac{1}{0.05} + 2 * 0.025 \log \frac{1}{0.025} \approx 1.445 < \log 5 = H_{wc}(T_4)$. Thus we could encode the trees using, on average, less than H_{wc} bits per tree. For example, we could assign 0 to the first tree, 10 to the second, 110 to the third, 1110 to the fourth, and 1111 to the fifth. The average code length is then $0.6*1+0.3*2+0.05*3+2*0.025*4 = 1.550$ bits. This is larger than H but smaller than H_{wc}.*

2.3 Zero-order empirical entropy

Consider the particular case $\Sigma = \{0, 1\}$, that is, where our infinite source emits bits. Assume that bit 1 is emitted with probability p and bit 0 with probability $1 - p$. The Shannon entropy of this source, also called **binary entropy**, is then:

$$H(p) = p \log \frac{1}{p} + (1 - p) \log \frac{1}{1 - p}$$

Picture 2.3 shows $H(p)$ as a function of p: as we can see, the **entropy** (i.e. the information) is **zero** when **one element has probability 1.0** and the **other 0.0**.
The concept of Shannon entropy also applies when the elements are sequences of those bits emitted by the source. Assume that the source emits each bit independently of the rest (this is called a "memoryless" or "zero-order" source). If we take chunks of n bits and call them our elements, then we have $\Sigma = \{0, 1\}^n$, and the Shannon entropy of the sequences is $nH(p)$.

2.3.1 Bit sequences

Assume we have a concrete **bit sequence**, $B[1, n]$, that we wish to compress somehow. We may have a good reason to expect that B has more 0s than 1s, however, or more 1s than 0s. Therefore, we may try to **compress** B using that property. That is, we will assume that B has been generated by a zero-order source that emits 0s and 1s.
Let m be the number of 1s in B. Then it makes sense to assume that the source emits 1s with probability $p = \frac{m}{n}$. This leads to the concept of **zero-order empirical entropy**:

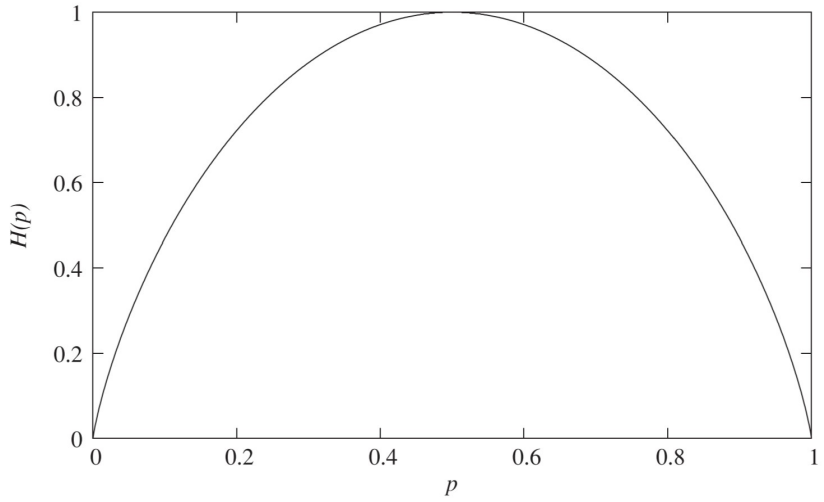


Figure 2: The binary entropy function

$$H_0(B) = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m}$$

, where:

- $\frac{m}{n} \log \frac{n}{m}$ represents the entropy of symbol 1;
- $\frac{n-m}{n} \log \frac{n}{n-m}$ represents the entropy of symbol 0.

Why zero-order? We use the adjective zero-order to define those sources that emit elements in a *memory-less* fashion, i.e. independently from each other.

Why empirical? We use the adjective empirical because it does not refer, as in Shannon's Theorem, to the actual probability distribution of the symbols, but rather to the observed distribution of the symbols in the text to encode.
The **practical meaning** of the zero-order empirical entropy is that, if a compressor attempts to compress B by using some fixed code $C(1)$ for the 1 and some fixed code $C(0)$ for the 0, then it **cannot compress** B to **less** than $nH_0(B)$ bits. Otherwise we would have

$$m|C(1)| + (n-m)|C(0)| < nH_0(B)$$

Calling $p = \frac{m}{n}$, this would give

$$\bar{l} = p|C(1)| + (1p)|C(0)| < H_0(B) = H(p)$$

, which means that this code would break the lower bound of the Shannon entropy.

A connection with worst-case entropy It is interesting that the concepts of **zero-order empirical entropy** and of **worst-case entropy** are **closely related**, despite their different origins. Consider the set $B_{n,m}$ of all the bit sequences of length n with m 1s. Then $|B_{n,m}| = \binom{n}{m}$, and its **worst-case entropy** is:

$$
\begin{aligned}
H_{\mathrm{wc}}(B_{n,m}) &= \log \binom{n}{m} = n\log n - m\log m - (n-m)\log(n-m) - O(\log n) \\
&= n\left(\frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}\right) - O(\log n) \\
&= nH_0(B) - O(\log n)
\end{aligned}
$$

, where we used the Stirling's approximation in the first line and expanded $n\log n = m\log n + (n-m)\log n$ in the second.

2.3.2 Sequences of symbols

The zero-order empirical entropy of a **string** $T \in \Sigma^n$, where each symbol x appears n_x times, is defined as:

$$H_0(T) = \sum_{x\in\Sigma} \frac{n_x}{n}\log\frac{n}{n_x}$$

More generally, if we have a string $S[1,n]$, where each symbol s appears n_s times in S, then:

$$H_0(S) = \sum_{1\le s\le\sigma} \frac{n_s}{n}\log\frac{n}{n_s}$$

Example. *Let S be "abracadabra", then $H_0(S) = \frac{5}{11}\log\frac{11}{5} + .. \approx 2.040$. Therefore, one could compress S using $nH_0 \approx 22.44$ bits, less than the $n\log|S| = 11\log 5 \approx 25.54$ bits of the worst-case entropy.*

2.3.3 Information theoretic lower bound

The information theoretic lower bound can be expressed as follows (from Navarro's book).

$$
\begin{aligned}
n\mathcal{H}_0(B) &= n\left(\frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}\right) \\
&= m\log\frac{n}{m} + (n-m)\log\left(1+\frac{m}{n-m}\right) \\
&\le m\log\frac{n}{m} + (n-m)\frac{1}{\ln 2}\,\frac{m}{n-m} = m\log\frac{n}{m} + \mathcal{O}(m)
\end{aligned}
$$

Figure 3: Information theoretic lower bound

2.4 High-order entropy

While the mere frequency of the symbols $\Sigma = [1,\sigma]$ can be an important source of compressibility for a sequence S, it is not the only one. For example, if one tokenizes the words in an English text, zero-order compression typically reduces the text to about 25% of its size. However, good compressors can reach less than 15% by exploiting the so-called **high-order entropy** or **k-th-order entropy**. This is a measure of the **information** carried by a **symbol given** that we **know** the k **symbols that precede it** in S. Indeed, one can better guess what the next word is in an English text if one knows some words preceding it. The **lower** the **surprise**, the **lower**

the **entropy**. In terms of the Shannon entropy, sources "with memory" remember the last k symbols emitted, and the probability of each emitted symbol may depend on this memory (this is a particular case of a Markov chain).
Before introducing the high-order entropy, we proceed with a definition. Given a text $T \in \Sigma^n$, let $s \in \Sigma^k$ be a string of length k, then we define T_s as the **context of s in T** as the concatenation of all the characters $T[i]$ s.t. $T[i+1, .., i+k] = s$, i.e. the string formed by collecting the symbols that precede each occurrence of the context s in T.

Example. *Let T be "abracadabra" and $k = 2$, then $T_{ra} = bb$ and $T_{ab} = \#d$.*

The **high-order entropy** of the text T is defined as:

$$H_k(T) = \sum_{s \in \Sigma^k} \frac{|T_s|}{n} H_0(T_s)$$

, i.e. the weighted average of the context's zero order entropy, where the weight is given by the frequency of the context T_s in the text T of length n.
Extending the concept of zero-order empirical entropy, $nH_k(S)$ is a **lower bound** to the **number of bits that any encoding** of S can **achieve** if the **code** of each **symbol** can be a **function** of **itself** and the k **symbols** preceding it in S. As before, any compressor breaking that barrier would also be able to compress symbols coming from the corresponding k-th-order source into less than its Shannon entropy.
As expected, it holds that

$$\log |S| = H_{wc}(S) \geq H_0(S) \geq H_1(S) \geq ... \geq H_{k-1}(S) \geq H_k(S)$$

for any k. Note that, for sufficiently large k values (at most for $k = n - 1$, and usually much sooner), it holds $H_k(S) = 0$ because all the contexts of length k appear only once in S. At this point, the model becomes useless as a lower bound for compressors.

2.5 Coding

The **entropy** tells us the **minimum average code length** we can achieve, and even suggests giving each symbol s a code with length $\log \frac{1}{Pr(S)}$ to reach that optimum. To obtain k-th-order entropy, we must use a different set of codes for each different previous context. However, in either case the entropy measure does not tell how to build a suitable set of codes.

Definition (Encoding). *An **encoding** is an **injective function** $C : \Sigma \to \{0, 1\}^*$ that assigns a distinct sequence of bits $C(s)$ to each symbol $s \in \Sigma$.*

Encodings are also simply called codes, overloading the meaning of the individual code assigned to a symbol. Since we will encode a sequence of symbols of Σ by means of concatenating their codes, even injective functions may be unsuitable, because they may lead to concatenations that cannot be unambiguously decoded.

Example. *Assume $\Sigma = \{a, b, c\}$, and $C(a) = 0$, $C(b) = 1$ and $C(c) = 00$. Then, $C(ac) = 000 = C(ca)$, so we cannot know which one of the two sequences was encoded.*

2.5.1 Unambiguous codes

Definition (Unambigous code). *An encoding is said to be **unambiguous** if, given a concatenation of bits, there is **no ambiguity** regarding the **original sequence** that was **encoded**.*

Said another way, if we define a new code C' on sequences of symbols of Σ, $C' : \Sigma^* \to \{0,1\}^*$,so that $C'(s_1,..,s_k) = C(s_1),..,C(s_k)$, then code C is unambiguous if and only if C' is an injective function.

Example. *Assume our set of codes is $C(s_1) = 1$, $C(s_2) = 10$, $C(s_3) = 00$. This code is unambiguous: if a bit sequence starts with 0, then it must start with $C(s_3) = 00$ and we can continue. Otherwise it starts with a 1, and we must see how many 0s are there up to the next 1 or the end of the sequence. If there are $2z$ 0s (for some $z \geq 0$), then it was $C(s1) = 1$ followed by z occurrences of $C(s_3) = 00$. If there are $2z + 1$ 0s, then it was $C(s_2) = 10$ followed by z occurrences of $C(s_3) = 00$.*

2.5.2 Prefix codes

The example shows a shortcoming of unambiguous codes: it might not be possible to decode the next symbol s in constant time or even in time proportional to $|C(s)|$. It is more useful that an encoding also be **instantaneous**, that is, that we have sufficient information to determine s as soon as we read the last bit of $C(s)$. It can be shown that instantaneous codes are precisely the so-called **prefix-free codes** or just **prefix codes**.

Definition (Prefix code). *A code C is a prefix code if **no code** $C(s)$ is a **prefix** of any **other code** $C(s')$.*

Clearly, with a prefix code there **cannot be ambiguity** with respect to whether we are seeing the final bit of a code or rather the middle of a longer code. Somewhat surprisingly, unambiguous codes that are not prefix codes are useless: there is always a prefix code that is at least as good as any given unambiguous code.
In this sense, we have that prefix codes can be **decoded** in **real time**, and they're the only codes we consider in this course.
Moreover, we have a very important property that states that if C is an unambiguous code with code length $l_1 = |C(x_1)|, .., l_\sigma = |C(x_\sigma)|$, where $\sigma = |\Sigma|$, then there exists a prefix code C' with the same length, i.e. $|C'(x_1)| = l_1, .., |C'(x_\sigma)| = l_\sigma$.
This property holds thanks to the **Kraft-McMillan inequality**: any unambiguous code C (and thus any prefix code C), satisfies

$$\sum_{s \in \Sigma} 2^{-l(s)} \leq 1$$

, where $l(s) = |C(s)|$, and on the other hand, a prefix code (and thus an unambiguous code) with lengths $l(s)$ always exists if the above inequality holds. Hence, given an **unambiguous code** C, it **satisfies the inequality**, and thus there **exists** a **prefix code** with the **same code lengths**.
This leads to a simple way of assigning reasonably good codes, called the ***Shannon-Fano codes***. Given that the optimal code length is $\log \frac{1}{Pr(s)}$, assign code length $l(s) = \lceil \log \frac{1}{Pr(s)} \rceil$. Then it holds:

$$\sum_{s \in \Sigma} 2^{-l(s)} = \sum_{s \in \Sigma} 2^{-\lceil \log \frac{1}{Pr(s)} \rceil} \leq \sum_{s \in \Sigma} 2^{-\log \frac{1}{Pr(s)}} = \sum_{s \in \Sigma} 2^{\log_2 Pr(s)} = \sum_{s \in \Sigma} Pr(s) = 1$$

and thus, by the Kraft-McMillan inequality, there is a prefix code with those lengths.
Actually, it is not difficult to find: process the lengths from shortest to longest, giving to each code the next available binary number of the appropriate length, where "available" means we have not yet used the number or a prefix of it. More precisely, if the sorted lengths are $l_1, .., l_\sigma$,

then the first code is 0^{l_1} (i.e. l_1 0s), and the code for s_{i+1} is obtained by summing 1 to the l_i-bit number assigned to s_i, and then appending $l_{i+1}\ \ l_i$ 0s to it. The Kraft-McMillan inequality guarantees that we will always find a free number of the desired length for the next code. Picture 2.5.2 gives the pseudocode of the *Shannon algorithm* for generating a sequence that satisfies the Kraft-McMillan inequality, and thus to produce a prefix-code with the same code length.

Algorithm 2.1: Building a prefix code over $\Sigma = [1, \sigma]$, given the desired lengths. Assumes for simplicity that the codes fit in a computer word.

Input : $S[1, \sigma]$, with $S[i].s$ a distinct symbol and $S[i].\ell$ its desired code length.
Output: Array S (reordered) with a new field computed, $S[i].code$, an integer whose $S[i].\ell$ lowest bits are a prefix code for $S[i].s$ (reading from most to least significant bit).

```
1 Sort S[1, σ] by increasing S[i].ℓ values
2 S[1].code ← 0
3 for i ← 2 to σ do
4 ⌊ S[i].code ← (S[i − 1].code + 1) · 2^(S[i].ℓ − S[i−1].ℓ)
```

Figure 4: Shannon algorithm

A ***Shannon-Fano*** code obtains an **average code length** that is **less than 1 bit over the Shannon entropy**. This is because the average code length is:

$$\bar{l} = \sum_{s\in\Sigma} Pr(s)\lceil \log \frac{1}{Pr(s)} \rceil < \sum_{s\in\Sigma} Pr(s) \left(\log \frac{1}{Pr(s)} + 1 \right) = H(Pr) + 1$$

If we consider a string $s \in \Sigma^n$, then the **Shannon encoding** takes $\leq n(H(Pr)+1) = nH(Pr)+n$ bits, which is **almost optimal**.

Example. *Consider the set T_4 with the probabilities given in the previous Example, {0.6, 0.3, 0.05, 0.025, 0.025}, which had Shannon entropy $H \approx 1.445$. The Shannon-Fano code gives lengths $\lceil \log \frac{1}{0.6} \rceil = 1$ for the first tree, $\lceil \log \frac{1}{0.3} \rceil = 2$ for the second, $\lceil \log \frac{1}{0.05} \rceil = 5$ for the third, and $\lceil \log \frac{1}{0.025} \rceil = 6$ for the last two. Then we start assigning code 0 to the first tree. For the second, the first available code of length 2 is 10, obtained by summing 1 to 0 and appending $2 \min 1 = 1$ 0s to it. For the third, we obtain 11000 by adding 1 to 10, getting 11, and then appending $52 = 3$ 0s. For the fourth and fifth we have codes 110010 and 110011. The average length of this code is $0.6 * 1 + 0.3 * 2 + 0.05 * 5 + 2 * 0.025 * 6 = 1.75$ bits. This is less than $H + 1$,but we found a better code in a previous Example.*

The example shows that, although *Shannon-Fano* codes always spend less than 1 bit over the entropy, they are not necessarily optimal. The next section shows how to build optimal codes.

2.6 Huffman codes

Huffman devised an **algorithm** that, given a **probability distribution** $Pr : \Sigma \to [0.0, 1.0]$, obtains a **prefix code** of **minimum average length**. Thus, **Huffman** is **optimal** among the codes that assign an integral number of bits to each symbol. In particular, it is **no worse** than *Shannon-Fano* codes, and so the number of bits it outputs is between $nH(Pr)$ and $n(H(Pr)+1)$, wasting less than 1 bit per symbol. Similarly, if Pr are the relative symbol frequencies in a string $S[1, n]$, then Huffman codes compress S to less than $n(H_0(S) + 1)$ bits.

2.6.1 Construction

The Huffman algorithm progressively converts a set of $|\Sigma|$ nodes into a binary tree. At every moment, it maintains a set of binary trees. The leaves of the trees correspond to the symbols $s \in \Sigma$, and each tree has a weight equal to the sum of the probabilities of the symbols at its leaves. The algorithm starts with $|\Sigma|$ trees, each being a leaf node corresponding to a distinct symbol $s \in \Sigma$, with weight $Pr(s)$. Then $|\Sigma|1$ tree merging steps are carried out, finishing with a single tree of weight 1.0 and with all the $|\Sigma|$ leaves.
The merging step always chooses two trees T_1 and T_2 of minimum weight and joins them by creating a new root, whose left and right children are T_1 and T_2, and whose weight is the sum of the weights of T_1 and T_2.
The single tree resulting from the algorithm is called the *Huffman tree*. If we interpret going left as the bit 0 and going right as the bit 1, then the path from the root to the leaf of each $s \in \Sigma$ spells out its code $C(s)$. The Huffman tree minimizes the average code length, $\sum_{s\in\Sigma} Pr(s)l(s)$.
The procedure for building a Huffman code is the following, where the input is represented by (Σ, Pr), and the output is a code $C : \Sigma \to \{0,1\}^*$:

1. If $|\Sigma| = 1$, return $C(x) = \epsilon$;
2. Let x, y be the characters with smallest probabilities $Pr(x)$ and $Pr(y)$;
3. Create a new character xy, with probability $Pr(xy) = Pr(x) + Pr(y)$;
4. $\Sigma = (\Sigma - \{x, y\}) \cup \{xy\}$;
5. $C \leftarrow \text{Huffman}(\Sigma, Pr)$, i.e. here there is the recursive call;
6. $C(x) = C(xy).0$, i.e. the left child of node xy;
7. $C(y) = C(cy).1$, i.e. the right child of node xy;
8. $\Sigma = (\Sigma - \{xy\}) \cup \{x, y\}$;
9. Return C.

Example. *Let S be "abracadabra", so $\Sigma = \{a, b, c, d, r\}$ and $Pr(a) = \frac{5}{11}$, $Pr(b) = Pr(r) = \frac{2}{11}$ and $Pr(c) = Pr(d) = \frac{1}{11}$. Picture 2.6.1 shows the Huffman algorithm on these probabilities.*

The final Huffman tree on the left assigns the codes $C(a) = 0$, $C(b) = 110$, $C(r) = 111$, $C(c) = 100$, and $C(d) = 101$. The average code length is $\frac{5}{11} * 1 + \frac{2}{11} * 2 + \frac{2}{11} * 3 + 2 * \frac{1}{11} * 4 \approx 2.091$, very close to $H(Pr) = H_0(S) \approx 2.040$. A Shannon-Fano code obtains a higher average code length, ≈ 2.727 bits. On the bottom right we show another valid Huffman tree that is obtained by breaking ties in another way (we leave as an exercise to the reader to determine the corresponding merging order). Its average code length is also ≈ 2.091.
The Huffman algorithm runs in time $O(|\Sigma| log |\Sigma|)$. This can be obtained, for example, by maintaining the trees in a priority queue to extract the next two minimum weights.

2.7 Variable-length codes for integers

Huffman codes are the **best** possible among those giving the **same code to the same symbol**. Sometimes, however, they can be inconvenient: a good example is the need to compress a sequence of natural numbers when usually most of them are small. In this case, we can choose a **fixed code** that **favors small number**s. We next show some of the most popular ones. Note that all these codes are prefix codes. From now on we assume we want to encode a natural number $x > 0$ and call $|x|$ its length in bits. If we want to encode the 0 as well (or up to some negative value), we may shift the values to encode.

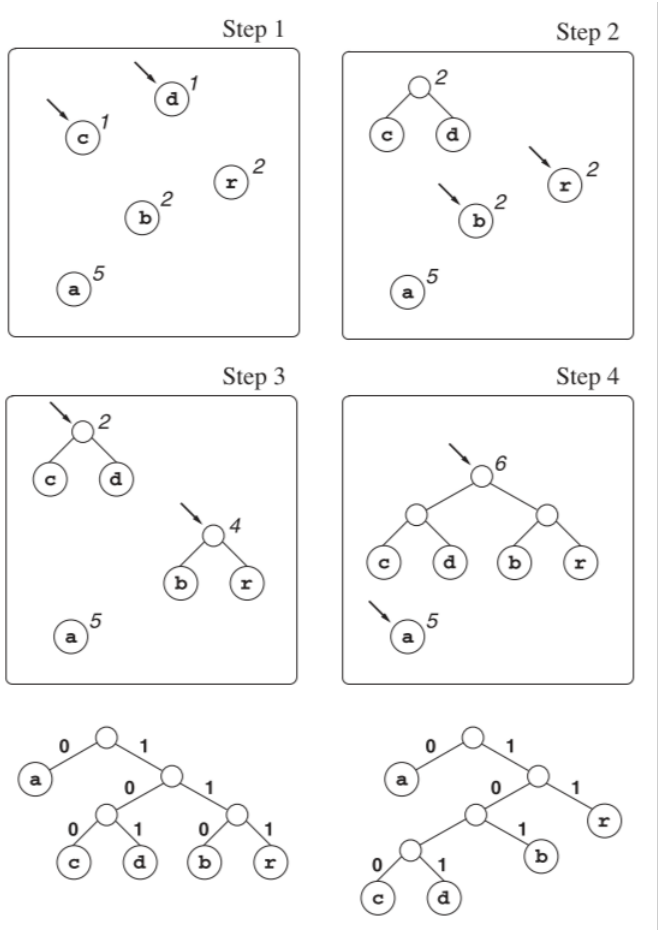


Figure 5: Huffman algorithm: an example

Our goal is then to encode natural numbers, i.e. to build a code $C : \mathbb{N}^+ \to \{0,1\}^*$ which is prefix.

2.7.1 Unary codes

The unary code is **convenient** when x is **extremely small**. It is defined as:

$$U(x) = 0^{x-1}1$$

For example, $U(3) = 001$. The unary code of x uses x bits.

2.7.2 γ-codes

This code is **convenient** when x is **small**, and it is defined as:

$$\gamma(x) = U(|x|)\tilde{x}$$

, where $\tilde{x}$ represents the least $|x| - 1$ significant bits of x (e.g. $5 =_2 101$, so $\tilde{5} = 01$), so it is basically the binary representation of x without the first bit.
For example, $\gamma(4) = U(|4|)\tilde{4} = U(3)\tilde{4} = 00100$. The gamma code of x uses $O(\log x)$ bits, so they become shorter than unary codes for $x \geq 6$. Moreover, they provide a prefix code, since the unary code is prefix, and it is concatenated with $\tilde{x}$.

2.7.3 δ-codes

When **numbers** are **too large** for γ-codes to be efficient, we can use δ-codes, which are defined as:

$$\delta(x) = \gamma(|x|)\tilde{x}$$

For example, $\delta(9) = \gamma(|9|)\tilde{9} = \gamma(4)001 = 00100001$. δ-codes use:

$$|\delta(x)| = |\gamma(|x|)| + |\tilde{x}| = 2\log\log x + \log x = \log x + O(\log\log x)$$

bits. The δ-codes are shorter than γ-codes for $x \geq 32$, and of the same length for $16 \leq x \leq 31$. Finally, they also provide a prefix-code.

2.8 Jensen's inequality

A tool that is frequently used to **analyze compression methods** is **Jensen's inequality**. It establishes that, if $f(x)$ is a concave function (that is, $f\left(\frac{x+y}{2}\right) \geq \frac{f(x)+f(y)}{2}$, like the logarithm), then:

$$\sum_{i=1}^{m} f(x_i) \leq m f\left(\frac{\sum_{i=1}^{m} x_i}{m}\right) \Leftrightarrow \frac{\sum_{i=1}^{m} f(x_i)}{m} \leq f\left(\frac{\sum_{i=1}^{m} x_i}{m}\right)$$

Roughly said, the **function of the average** is **larger** than the **average of the functions**.

2.8.1 Differential encoding of increasing numbers

Assume we have increasing numbers $0 = y_0 < y_1 < y_2 < .. < y_m = n$. As a way to compress them, we store them differentially: we encode $x_1, .., x_m$, where $x_i = y_i y_{i1}$. If m is not much smaller than n, then most x_i values will be small, and we can apply the encoding methods just seen. For example, assume we use $\delta - codes$. The total size of the encoding is:

$$\begin{aligned}
|C(x)| &= |\delta(x_1)| + \sum_{i=2}^{m} |\delta(x_i - x_{i-1})| \\
&= \log x_1 + O(\log\log x_1) + \sum_{i=2}^{m} \log x_i - x_{i-1} + O(\log\log x_i - x_{i-1}) \\
&= \text{ we use Jensen's inequality, since log and log log are concave functions} \\
&\leq m \log\left(\frac{x_1 + (x_2 - x_1) + .. + (x_m - x_{m-1})}{m}\right) + O\left(\log\log\left(\frac{x_1 + (x_2 - x_1) + .. + (x_m - x_{m-1})}{m}\right)\right) \\
&\leq m \log\frac{x_m}{m} + O\left(m \log\log\frac{x_m}{m}\right) \\
&\leq m \log\frac{n}{m} + O\left(m \log\log\frac{n}{m}\right)
\end{aligned}$$

bits, which is very similar to the lower bound $H_{\text{wc}} = m \log\frac{n}{m} + O(m)$, but we still cannot do any operation, since this does not represent a data structure.

2.9 Application: positional inverted indexes

A **(positional) inverted index** is a **data structure** that provides **fast word searches** on a natural language text T. Given a text $T \in \Sigma^n$, where Σ represents the alphabet, then the goal of an inverted index is to build a map between each word and the position where the word appears in the text:

$$M : \Sigma \to \{0, 1, .., n\}^*$$

Let n be the number of words in T, and n_s the number of times word s appears in T. Then the positions stored by the list of word s form a sequence $0 < p_1 < p_2 < .. < p_{n_s} \leq n$. If we encode the differences using δ-codes, then the index takes $nH_0 + n \log H_0 + O(n)$ bits.

3 Bitvectors

A **bitvector** is a bit array $B[1, n]$ that supports the following operations:

- access(B, i): returns the bit $B[i]$, for any $1 \leq i \leq n$;
- rank$_v(B, i)$: returns the number of occurrences of bit $v \in \{0, 1\}$ in $B[1, i]$, for any $0 \leq i \leq n$; in particular rank$_v(B, 0) = 0$. If omitted, we assume $v = 1$;
- select$_v(B, j)$: returns the position in B of the j-th occurrence of bit $v \in \{0, 1\}$, for any $j \leq n$; we assume select$_v(B, 0) = 0$ and select$_v(B, j) = n+1$ if $j >$ rank$_v(B, n)$. If omitted, we assume $v = 1$.

Bitvectors are fundamental in the implementation of most compact data structures. Therefore, an efficient implementation is of utmost importance.
While **access** is equivalent to operation read on bit arrays, **rank** and **select** correspond to the sum and search operations of partial sums, if we interpret the bits 0 and 1 as the numbers 0 and 1, respectively. We will take advantage of the fact that the array elements are bits to provide **improved solutions** for **rank** and **select**.
In addition, we put more emphasis on the **compression of bitvectors** than when we studied arrays. Many applications give rise to unbalanced bitvectors, that is, with far fewer 1s than 0s, or vice versa, and thus with low empirical zero-order entropy. We will compress bitvectors B to $nH_0(B) + o(n)$ bits, using those $o(n)$ extra bits to support the operations. We will also achieve high-order entropy, $nH_k(B) + o(n)$ bits.

3.1 Zero-order compression

In many cases of interest, B contains many more 0s than 1s, or vice versa. In those cases, we can encode B in a different form, so that we use **space** close to its **zero-order empirical entropy**, $nH_0(B)$ bits, and still perform the **operations** in **constant time**. In Chapter 2 we showed that $nH_0(B) \approx n$ when there are about as many 0s as 1s, therefore the technique we describe here is not useful in that case: it would actually increase the space and the time. Instead, it significantly reduces the space on unbalanced bitvectors.

3.1.1 Structure

Let $B = 1000101000110100$, then the **structure** of the **zero-order compression** of B is built as follows:

1. We choose $b \in \mathbb{N}$, and we **break** B into $B_1, B_2, ..$ **blocks** of length b. In the example, $B_1 = 1000$, $B_2 = 1010$, etc..;
2. We define the **class** c_i of a block B_i as the **number of 1s** in the block. In the example, $c_1 = 1$, $c_4 = 1$, etc.. Each **class** c_i is **encoded** using $\lceil \log(b + 1) \rceil$ bits (in our case 3 bits). Notice that this represents a **fixed-size encoding**;
3. We define $L_{b,c}$ as the **list of all the bitvectors** of length b with c 1s, **sorted** in increasing order. In the example, $L_{4,1} = 00001, 0010, 0100, 1000$.
4. We define the **offset** o_i of a block B_i as the integer i s.t. $B_i = L_{b,c_i}[i]$. In the example, since B_3 is contained in $L_{4,2}$ at position 1, $o_3 = 1$. Similarly, $o_4 = 3$ etc..Each offset o_i is **encoded** using a **variable-length encoding**:

$$1 \leq o_i \leq \binom{b}{c_i}$$

, so

$$|o_i| = \lceil \log \binom{b}{c_i} \rceil$$

5. Finally, the **compressed representation** of the bitvector B is given by an **array**:

$$\text{enc}(B) = \left((c_1, o_1), .., (c_{\frac{n}{b}}, o_{\frac{n}{b}}) \right)$$

Example. *Let* $B = 1000101000110100$ *and* $b = 4$*. Then:*

1. $B_1 = 1000$*,* $B_2 = 1010$*,* $B_3 = 0011$*,* $B_4 = 0100$*;*
2. *For the classes we use* $\lceil \log(5) \rceil = 3$ *bits:* $c_1 = 1 = 001$*,* $c_2 = 2 = 010$*,* $c_3 = 2 = 010$*,* $c_4 = 1 = 100$*;*
3. *For the offsets, since* $c_i = 1$ *or* $c_i = 2$*, we use* $\lceil \log \binom{4}{1} \rceil = 2$ *bits or* $\lceil \log \binom{4}{2} \rceil = 3$ *bits. Moreover, since we do not encode 0, we encode* $o_i - 1$*:* $o_1 = 4 - 1 = 3 = 11$*,* $o_2 = 5 - 1 = 4 = 100$*,* $o_3 = 1 - 1 = 0 = 000$ *and* $o_4 = 3 - 1 = 2 = 10$*.*

The concatenation of all these bits is a **decodable encoding** (from the classes we can derive the offsets) and **prefix encoding**.

3.2 Space usage

Before showing how to provide efficient access to this encoding, let us motivate it by showing that it actually compresses B.
Let $b = \lceil \frac{\log n}{2} \rceil$ (in our example $n = 16$, thus $b = 2$). How much space do the classes take?

$$O\left(\frac{n}{b} \log b\right)$$

, where

- $\frac{n}{b}$ is the **number of classes**;
- $\log b$ is the **space** taken by **each class**.

If we substitute $b = \lceil \frac{\log n}{2} \rceil$, we obtain:

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

bits.
The most interesting part, however, is the analysis of the **number of bits** required for **array** O, storing the **offsets**. Since each o_i uses $|o_i| = \lceil \log \binom{b}{c_i} \rceil$, we have:

$$\begin{aligned}
\sum_{i=1}^{n/b} |o_i| &= \sum_{i=1}^{n/b} \lceil \log \binom{b}{c_i} \rceil \\
&\leq \sum_{i=1}^{n/b} \left(\log \left(\binom{b}{c_i} + 1 \right) \right) = \sum_{i=1}^{n/b} \log \binom{b}{c_i} + \frac{n}{b} \\
&= \log \prod_{i=1}^{n/b} \binom{b}{c_i} + O(n) \\
&\leq \binom{b+b+..}{c_1+c_2+..} = \binom{n}{m} \text{ because all the choices in } \binom{b+b+..}{c_1+c_2+..} \text{ contain the choices of } \binom{b}{c_i} \\
&\leq \log \binom{n}{m} = H_{\text{wc}}(B_{n,m}) = nH_0(B)
\end{aligned}$$

Notice that in the last equality we exploited the connection between worst-case entropy and zero-order empirical entropy, explained in Chapter 2 (the term $O(\log n)$ can be omitted since it can also assume value 0).

Theorem. *We can encode B using $nH_0(B) + o(n)$ bits, where $H_0 = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m}$.*

3.3 Bitvector data structure

Now we consider a **data structure** over the compressed bitvector representation, in order to be able to perform some **operations**.

3.3.1 Model of computation

The model of computation we consider is the **word RAM model**, i.e. we work with **words** of size w (a word can be considered as an integer), with $w = \Theta(\log n)$, with n be the **size of the input**. Moreover, we assume **constant-time cost** (i.e. $O(1)$ time) for the following operations between words $W \in \{0,1\}^w$:

- $W_1 * W_2$;
- $W_1 \pm W_2$;
- Shift of i bits: $W >> i$ and $W << i$;
- Bitwise AND,OR and negation;
- popcount(W) = number of 1 in W.

3.3.2 Operations

We recall that the operations are:

- **Access**: $B[i] = b_i$;
- **Rank**: $B.\text{rank}(i) = \sum_{j=0}^{i-1} B[i]$;
- **Select**: $B.\text{select}(i)$ = position of the i-th '1' (assuming $i \geq 1$).

Example. *Let $B = 0110100101$, then:*

- $B[3] = 0$;
- $B.rank(4) = 2$;
- $B.select(4) = 7$

3.3.3 Naive implementation

The **naive implementation** defines an array $R[0, .., n-1]$ defined as $R[i] = \sum_{j=0}^{i-1} b_j$. In this case we have:

- **Space**: $n \log n$ bits;
- **Access/Rank**: $O(1)$ time, i.e. constant time;
- **Select**: $O(\log n)$, using binary search, or $O(1)$ adding another array.

3.3.4 Smarter implementation: balanced binary tree

In this case we **split** the bit **sequence** in **blocks** of $\log n$ bits, and store **each block** in a separate **leaf**. In this way, the internal nodes contain the partial size/rank of the bitvector. We have:

- **Space**: $O(n)$ bits;
- **Access/Rank/Select**: $O(\log n)$ time.

3.3.5 Main result: optimal space and time bitvector

Theorem. *In the word RAM model, there exists a data structure that solves the static bitvector problem with the following complexities:*

- ***Space**: $n + o(n)$ bits, which can be improved to $nH_0 + o(n)$;*
- ***Access/Rank/Select**: $O(1)$ time, i.e. constant time.*

For describing this DS, we use a fundamental simple trick, called **bit packing**. An example of bit packing is the following: consider a bitvector of length $n = 32$, defined as 11001011010010100101010010101110. We consider a word size $w = 5 = \Theta(\log n)$, and we pack the bitvector in $\lceil n/w \rceil$ words with padding:

$$11001 \quad 01101 \quad 00101 \quad 00101 \quad 01001 \quad 01011 \quad 10000$$

In this case the space is $\leq n + w = n + o(n)$ bits, and using bitwise operations, in $O(1)$ time we can:

- **Extract** a single bit;
- **Extract** any sub-sequence of consecutive w bits;
- **Count** the number of ones in any sub-sequence of consecutive w bits.

Contant time access The operations to do are:

1. Extract (c_i, o_i) in $O(1)$ time;
2. Convert (c_i, o_i) to decode the bit.

For the **classes**, we can extract them in $O(1)$ using the property of bit packing described above (since each $|c_i|$ is fixed). In particular, the sequence of classes can be represented as a sequence of n/b blocks of $\log b$ bits, so the access can be implemented in $O(1)$ time.
For the **offsets**, we know that each $|o_i|$ may differ from the others, so the idea is to build an uxiliary data structure as follows:

1. Sample one offset out of every $\log n$, which defines what we can call a superblock, and store a pointer to it in a vector;
2. For the remaining offsets, we store them as the bit distance from the first offset of their superblock (which is the one associated to the pointer).

The Picture below shows a scheme of this structure.

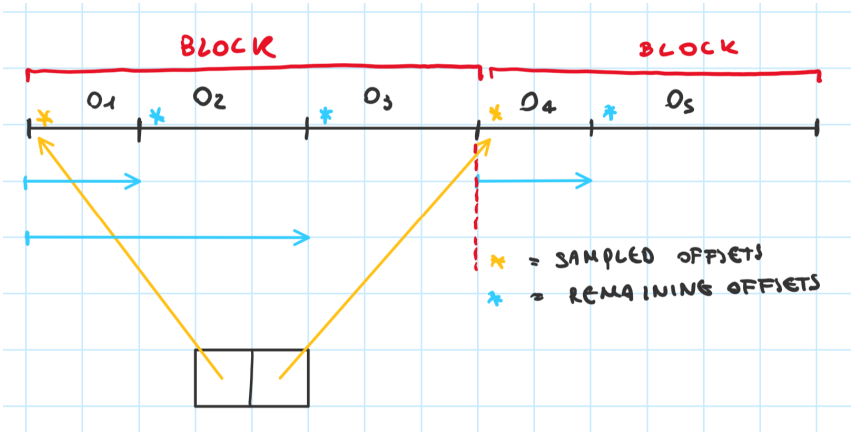


Figure 6: Storing Bitvector offesets

Some observations:

1. Each offset takes $\lceil \log \binom{b}{c_i} \rceil$, which is $\leq \log n$ bits;
2. Each superblock, therefore, groups at most $\log^2 n$ bits, because there are $\log n$ offsets in a superblock, each of at most $\log n$ bits;
3. There are $\frac{n}{\log^2 n}$ superblocks;
4. There are $\frac{n}{\log^2 n}$ pointers, one for each superblock, which take $\log n$ bits each to represent;
5. The maximum distance within a superblock is $\log^2 n$, which is the maximum number of bits in a superblock, meaning that such a distance takes $\log \log^2 n$ bits to represent.

The spatial complexity of this offsets representation is therefore

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

, where

- $\frac{n}{\log^2 n}$ is the number of superblocks;
- $\log n$ is the space for the pointers;
- $\frac{n}{\log n}$ is the number of remaining offsets;
- $\log \log^2 n$ is the space of superblock distances.

Finally, in order to **convert** (c_i, o_i) into the original bitvector, we need a sort of function f s.t.:

$$f : (c_i, o_i) \rightarrow B_i$$

The idea is to use a matrix T s.t. $T[c_i][o_i] = B_i$.
For the **space**, we have that:

- Inside each cell we store $|B_i| = b = \Theta(\log n)$ bits;
- There are $b = \Theta(\log n)$ combinations of c_i;
- $1 \leq o_i \leq \binom{b}{c_i} \leq 2^b = 2^{\log n/2} = \sqrt{n}$, so there are at most $\sqrt{n}$ offsets.

Thus, the total space occupied by T is $O(\sqrt{n} \log^2 n) = o(n)$ bits.

Theorem. *In the word-RAM model we can store any bitvector $B \in \{0,1\}^n$ of length n using $nH_0(B) + o(n)$ bits, so that aby subsequence $B[i, .., i + k]$, with $k = O(\log n)$ can be retrieved in $O(1)$ time.*

We recall that the information theoretic lower bound tells us that

$$nH_0(B) \leq m \log \frac{n}{m} + O(m)$$

, from which we derive that:

$$nH_0(B) + o(n) \leq m \log \frac{n}{m} + O(m) + o(n) = H_{wc}(B_{n,m}) + o(n)$$

, so it almost matches the information theoretic lower bound.

Constant time rank Before seeing the optimal solution, we underline that the rank operation can also be solved using two approaches:

- Exploiting a simple **scan** of the bitvector, and summing up the 1's in it. This solution takes $nH_0 + o(n)$ bits and $O(n)$ time;
- **Pre-computing** all the possible values of the rank, taking $O(n \log n)$ bits and $O(1)$ time.

However, we can achieve $nH_0 + o(n)$ bits[1] and $O(1)$ time. The idea is to divide the bitvector into **blocks** of $\log n$ bits, and **superblocks** of $\log^2 n$ bits. Then:

- There are $\frac{n}{\log n}$ blocks, since each block consists of $\log n$ bits;
- There are $\frac{n}{\log^2 n}$ superblocks, since each superblock contains $\log n$ blocks, each taking $\log n$ bits;
- For each superblock, we store explicitly the partial rank, using $\frac{n}{\log^2 n} \log n = \frac{n}{\log n} = o(n)$ bits;

[1] In order to obtain $o(n)$ we need to exploit the *macroblocks*

- For each block, we store the partial rank from the beginning of the corresponding superblock, using $\frac{n}{\log n} \log \log^2 n = o(n)$ bits (notice that the partial rank takes $\log \log^2 n$ since each superbloc consists of $\log^2 n$ bits);
- Finally, the rank in a block is answered in $O(1)$ time using bit-masks + popcount on the packed bitvector.

The total space required by this auxiliary data structure is:

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

Finally, we have a data bitvector (which takes n bits) and an auxiliary data structure (which takes $o(n)$ bits) that allow us to perform rank in $O(1)$, using a total space of $nH_0 + o(n)$ bits.

Example. *Consider a bitvector of length* $n = 32$*, then* $\log n = 5$*, and* $\log^2 n = 25$*.*

bitvector	11001	01101	00101	00101	01001	01011	10
superblocks	0					12	
blocks	0	3	6	8	10	0	3

Figure 7: Example of bitvector, blocks and superblocks

Computation of the rank Suppose that we want $B.\text{rank}_1(i)$. We would need to execute the following steps, which are all $O(1)$ time:

- Get the partial rank at the superblock of i (whose index can be identified with a simple division, $\lfloor i / \log^2 n \rfloor$), which has been precomputed;
- Sum the partial rank of such a superblock to the partial rank of the blocks that precede the block containing bit i (one way to know which block contains bit i is to perform $\lfloor i / \log n \rfloor$; the starting block of the superblock is also known, because we know which superblock we are in, hence the retrieval of the rank of the blocks preceding i can be done), which, as well, have been precomputed;
- Use a bitmask to extract the first bits of i's block, up to bitvector position i (this yields a bitvector where the bits of the block after i are 0). We refer to this "masked block" as X;
- Query a precomputed table of all possible block-rank pairs to get the rank of X. Such table takes a small amount of space, $o(n)$, because block size is small ($\log n$ bits), and therefore there is a relatively small amount of block-rank combinations.

Theorem. *Finally, we can conclude that we can build a DS on a bitvector* $B \in \{0,1\}^n$ *with* m *1's taking* $nH_0 + o(n)$ *bits, and supporting access, rank and select in* $O(1)$ *time. Please notice that* $nH_0 + o(n) \leq m \log \frac{n}{m} + o(m) + o(n)$*, where* $m \log \frac{n}{m} + o(m) = H_{wc}(B_{n,m})$ *(the worst case entropy), while* $o(n)$ *represents an extra term which is used to answer queries in* $O(1)$ *time.*

3.3.6 Applications

A good example of usage of bitvectors is when they are exploited as **data structures** for **sets**. A bitvector of n bits can be used to represent any set of cardinality n: the i-th bit is set to 1 if the i-th element is in the set, and 0 otherwise.
This solution is **good** in particular if m is **close** to n: by taking a look at the spatial complexity, we can see that it is influenced by both m, the number of 1s, and n, the total number bits/elements. On the other hand, if $m << n$, then we are **wasting** a lot of **space** (mainly due to the auxiliary data structures that takes $o(n)$) to store relatively **few elements**, because it means that the bitvector is sparse and the number of 0s is much larger.
An example of wasted space would be the set of all possible IP addresses, which has $n = 2^{32}$, used to store just one thousand addresses: $m = 1000$, $n = 2^{32} \Rightarrow m << n$.

3.4 Very sparse bitvectors - the Elias-Fano representation

Let's consider the following problem: given as input an **increasing sequence** $S[1, m]$ of m integers from the universe $[0, n)$ (notice that if $m << n$, then the sequence can be considered as a very sparse bitvector of length n with m 1's), our goal is to build a **DS** answering the following queries:

- Access: $S[i]$;
- S.predecessor(k), i.e., for a given k, find the largest i s.t. $S[i] \leq k$;
- S.successor(k), i.e., for a given k, find the smallest i s.t. $S[i] \geq k$.

Example. *Let* $S = 013479$*, then* $S[5] = 7$ *and* $S.successor(6) = 5$*, since* $S[5] = 7$.

We already discussed how static bitvectors result to be inefficient for such scenario, so we discuss some optimizations.

3.4.1 Naive solution

The **naive solution** uses $\log n$ bits for each element of S. In this case:

- **Space**: $m \log n$ bits;
- **Access**: $O(1)$ time, since we're storing the whole sequence of numbers, and in the word-RAM model the access is implemented in $O(1)$ time;
- **Predecessor/successor**: $O(\log m)$ time, exploiting binary search (we recall that the sequence is in increasing order).

3.4.2 Elias-Fano representation

This representation is just an **update** of the previous one, but the results are quite interesting, since:

- **Space**: $m \log \frac{n}{m} + O(m) = nH_0(B) + O(m)$ bits, which is optimal;
- **Access**: $O(1)$ time;
- **Predecessor/successor**: $\log \frac{n}{m}$ time.

S	$\log_2 n$
0	00000
5	00101
8	01000
12	01100
14	01110
17	10001
20	10100
31	11111

Figure 8: Representing each element using $\log n = 5$ bits

S	$\log_2 m$	$\log_2(n/m)$
0	000	00
5	001	01
8	010	00
12	011	00
14	011	10
17	100	01
20	101	00
31	111	11

Figure 9: Breaking the integers into prefix and suffix

Suppose, for example, that $m = 8$ and $n = 32$, so each $S[i] = \{0, 1, .., 31\}$, and $\log n = 5$. In this sense, each number can be written using 5 bits, as showed in Picture 3.4.2.
The idea now is to **break** the integers of 5 bits into a **prefix** of $\log m = 3$ bits and a suffix of $\log(n/m) = 2$ bits, obtaining the following results.
We can notice that:

- The orange sequence is non-decreasing;
- The first integer of the orange sequence is ≥ 0, while the last one is $\leq m - 1$;

Now, the idea is to store the **differences** between the prefixes using **unary encoding**, i.e. we encode the sequence of deltas using unary encoding with a bitvector of $2m$ bits. In this case, the deltas are thus:

$$1|01|01|01|1|01|01|001$$

Then, the next step is to encode the **suffixes** as is, i.e. using a bitvector that supports constant time access, using $m \log \frac{n}{m}$ bits.
To sum up, the Elias-Fano representation is the (EF_1, EF_2) pair, as showed in Picture 3.4.2.
Some notes:

$$
\begin{array}{rcl}
S & = & 0\ 5\ 8\ 12\ 14\ 17\ 20\ 31 \\
EF_1 & = & 1|01|01|01|1|01|01|001 \\
EF_2 & = & 00|01|00|00|10|01|00|11
\end{array}
$$

Figure 10: Elias-Fano data structure

- For the **prefixes** we use **unary encoding**, thus one 1 for each integer (i.e. m 1's) and $\leq m$ 0's, thus a total of $\leq 2m$ bits;
- For the **suffixes**, we use $m \log \frac{n}{m}$ bits.

Thus, the **total space** is $m \log \frac{n}{m} + 2m$, which matches the information theoretic lower bound. If we consider the Elias-Fano data structure, then we have:

- EF_1 can be encoded using our **bitvector data structure**, which uses $< n + o(n)$ bits. In this case, we have $\leq 2m + o(m)$ bits;
- EF_2 can be encoded using the **packed array** (see previous Theorem) in $m \log \frac{n}{m} + \Theta(\log m)$ bits;
- **Total space** of the data structure is $m \log \frac{n}{m} + 2m + o(m)$ bits;
- **Access**: $O(1)$ time. The reason is that we use select(i) on EF_1 (jump to the i-th 1 in EF_1), and we count the number of 0s before that position: this quantity in binary represents the prefix, and it used to access the EF_2 to retrieve the suffix in $O(1)$ time. Then, the final value is given by the prefix + suffix;
- **Predecessor/successor**: $O(\min(\log m, \log \frac{n}{m}))$ time, using binary search and access.

Theorem. *We can store a sequence of m integers $S[1, .., m]$ from $[0, n)$ using $m \log \frac{n}{m} + 2m + o(m)$ bits s.t. $S[i]$ is retrieved in $O(1)$ time and the predecessor/successor operations are performed in $O(\min(\log m, \log \frac{n}{m}))$ time. Equivalently, a bitvector $B \in \{0, 1\}^n$ with m 1's can be stores in $m \log \frac{n}{m} + 2m + o(m) = nH_0 + 2m + o(m)$ bits so that we can perform random access and rank/select in $O(\min(\log m, \log \frac{n}{m}))$.*

Please notice that if the bitvector is very sparse, we must use this representation, and we can also implement this representation in an inverted index, since it is even better than δ-encoding.

4 Compressed indexing

Definition (Inverted index). *Given an **alphabet** of words Σ (e.g. $\Sigma = \{tree, code, index, ..\}$), and a **text** $T \in \Sigma^n$, an **inverted index** is a **map** between the alphabet and $\mathbb{N}$, in particular is defined as:*

$$M : \Sigma \to \mathbb{N}^*$$

, and $M[w]$ represents the set of positions i s.t. $T[i] = w$.

One problem of inverted indexes is that they work only when the text to search can be broken into separate words (because each entry of the index is a single word of the text's vocabulary), so they are not suited to search single characters or substrings that span multiple words.
Inverted indexed are used a lot, also for the fact that texts can be broken into several blocks of words.
In general, compressed computation is a field of merging algorithms, data structures and information theory, and the question of this field is the following: I have a compressed representation C of a file X, with size $|C| << |X|$. Can i perform computation directly over C, without decompressing it? In general, the solution depends on the compressor C and on the problem type (i.e. input and queries). In this lecture we will see solutions for different compressors and one particular problem, the compressed indexing problem.

Definition (Compressed indexing problem). *Given as input a text $T \in \Sigma^n$, where Σ is an alphabet of symbols $\Sigma = \{a, b, c, ..\}$, our goal is to build a **data structure** $D(T)$ supporting efficiently these queries:*

- ***Count**: $D(T).count(P)$, where $P \in \Sigma^m$ $(m \leq n)$, that counts the number of occurrences of P in T;*
- ***Locate**: returns the positions where P occurs in T, i.e. the position i s.t. $T[i, .., i+|P|-1] = P$;*
- ***Extract**: extract any substring of length l that starts in i, i.e. $T[i, .., i + l - 1]$.*

The **constraint** of this problem is that $|D(T)|$ should be close to nH_k, the **high-order empirical entropy**.

Example. *Let $T = ATATAGATA$, then:*

- *count(ATA) = 3;*
- *locate(ATA) = $\{1, 3, 7\}$;*
- *extract($T[3, .., 7]$) = ATAGA.*

In the following sections we will see:

1. A solution taking nH_0 bits, called *Compressed Suffix Array* (*CSA*);
2. A solution taking nH_k bits, called *FM index.*

4.1 Notation

The notation used in the following sections is the following:

- $n = |T|$;
- Σ is the alphabet, and $\sigma = |\Sigma|$, assuming $\sigma \leq n$;

- P is the pattern of length $m = |P|$;
- occ is the number of occurrences of P in T (i.e. the count(P));
- l is the length of the substring to be extracted from T;
- Logarithms are in base 2.

4.2 Classic text indexing

The two classic methods that are used for text indexing are:

- Suffix tree;
- Suffix array.

Notice that both the methods do not provide a compressed structure.

4.2.1 Suffix tree

Example. *Let T = BANANA\$, where \$ $\leq c \quad \forall c \in \Sigma$ is the terminator character. The corresponding suffix tree is built as follows (notice that is built from the last char backwards).*

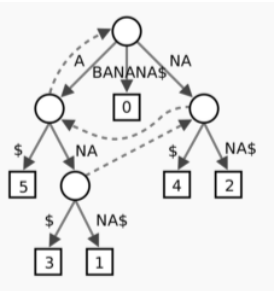


Figure 11: Suffix tree of BANANA

A Suffix tree has a node for each different suffix in the text. The first level of suffixes consists of all the lexicographically ordered unique suffixes, with the leftmost being the smallest. This definition is then recursively applied to each set of children until the whole set of suffixes is exhausted.
As we can see, it seems like this structure takes $O(n^2)$ strings, but each string of the edges is a substring of the initial text T, so we can store in the edges the positions of the suffix in T (tuples in the parenthesis). Finally, in the leaves we store the initial position of the given suffix.
Thus, we have $2n - 1$ nodes and n leaves, with a total space of $O(n \log n)$ bits, which is huge (if we consider the human genome, we have 3 billions characters, so the suffix tree takes $\approx$ 120GiB in this case), and which could be two orders of magnitude larger than nH_k, our desired result.
In this sense, despite supporting extract, count and locate operations in optimal time, there's the problem of space.

4.2.2 Suffix array

A suffix array represents an improvement of a suffix tree. Given an input \$-terminated text S, we lexicographically sorts the suffixes, and the corresponding indexes represent the suffix array. An example is represented in Picture 4.2.2.
Suffix array and text represent a data structure that implements all the queries:

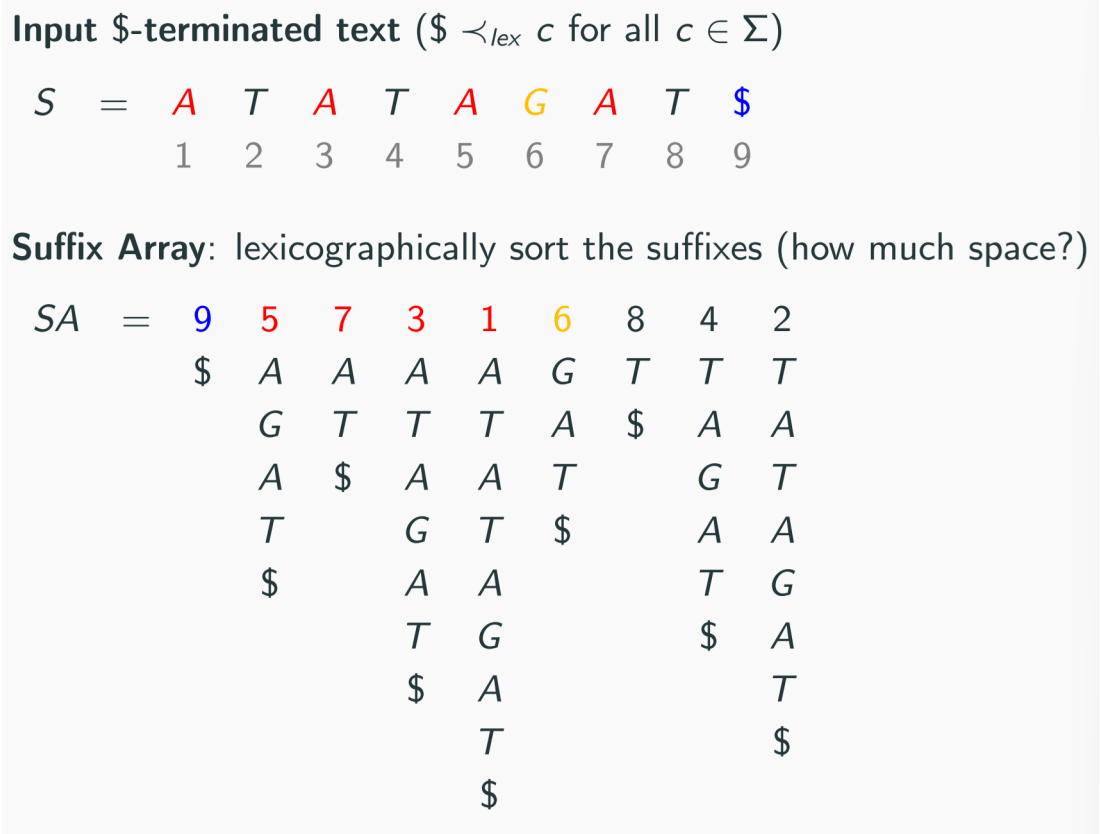


Figure 12: Suffix array - example

- **Count**: since we have an increasing list of suffixes, we can use binary search. Suppose for example that $P = \text{T\$}$, then since T\$ > ATATAGAT\$, we search in [6,8,4,2] etc.. (simple binary search). The complexity in this case is $O(m \log n)$ (very close to the optimal), where $\log n$ represents the number of steps of the binary search, and m represents the length of the pattern that is compared;
- **Locate**: $O(m \log n + \text{occ})$, still exploiting binary search. Notice that the term occ is used to enumerate the results;
- **Extract**, which is easy.

The **space** used by the suffix array is the space of the suffix array and the space of the text, thus $O(n \log n + n \log \sigma)$ bits, where

- $n \log n$ represents the size of the Suffix array, since it contains n numbers, each of which can be encoded using $\log n$ bits;
- $n \log \sigma$ represents the size of the text, where $\sigma = |\Sigma|$, since it contains n characters, each of which can be encoded using $\log \sigma$ bits.

If we use this structure for the human genome, we use 13GiB.

4.3 Compressed suffix array (CSA)

We consider the following array ψ, defined as follows:

$$\psi[i] = \text{SA}^{-1}[\text{SA}[i] + 1 \mod n]$$

, except for $\psi[1] = \text{SA}^{-1}[1]$.

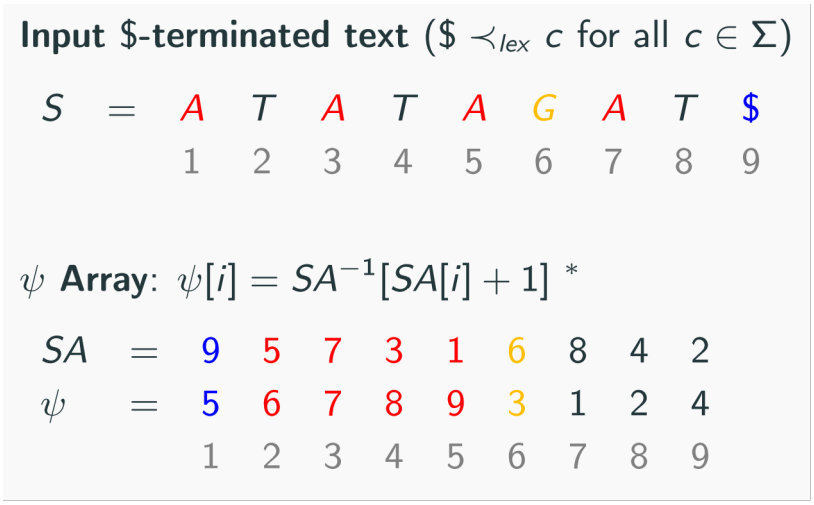


Figure 13: ψ array - example

Some observations:

- $\text{SA}[i] + 1$ is the index of the character that immediately follows the first character of $\text{SA}[i]$. Equivalently, $\text{SA}[i] + 1$ is the starting index of the suffix obtained by removing the first character of the suffix pointed to by $\text{SA}[i]$ (by doing $+1$ we are considering the second character of such a suffix, hence "removing" it);
- $\text{SA}^{-1}[k]$ is an index j such that $\text{SA}[j] = k$ (that is why the inverse notation is used, we are retrieving the index of the provided element, as if we were saying SA.indexOf(k));
- $\mod n$ is used because the SA array is treated as circular.

Therefore, each element $\psi[i]$ is therefore the index of the SA of the character following $\text{SA}[i]$. In the example, we have that:

- $\psi[1] = \text{SA}^{-1}[1] = 5$, since $\text{SA}[5] = 1$;
- $\psi[2] = \text{SA}^{-1}[\text{SA}[2] + 1] = \text{SA}^{-1}[6] = 6$, since $\text{SA}[6] = 6$;
- etc..

Notice that ψ is **increasing** by letter (color). Why? Applying ψ removes the first char from a suffix, so it preserves the relative ordering of suffixes starting with the same letter.

4.3.1 Time complexity

Evaluation of $\psi[i]$ In this case the evaluation is constant, i.e. $\psi[i]$ is computed in $O(1)$.

Count In this case the **range of suffixes** prefixed by a pattern P can be found with **binary search** using ψ and an array FL containing the first letter of each suffix (retrieved from the SA): in this case the running time is the one of a binary search, so $O(m \log n)$, as in a SA. In particular, we have $\log n$ steps of binary search, and at each step we extract at most m chars from the CSA, in order to establish whether the pattern is longer or shorter than the current suffix.

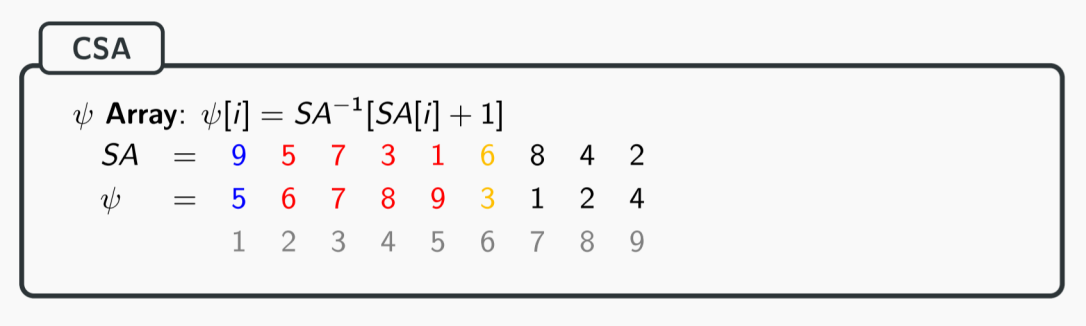


Figure 14: Compressed suffix array (CSA)

Locate (SA$[i]$) Given a pattern P of length m, performing the locate operation reduces to access $SA[i]$, where i is an index such that $FL[i]$ is the first letter of the pattern. This means that the first step to perform locate is to perform binary search as described for the count operation. Then, $SA[i]$ has to be computed from FL and ψ, because it is not stored.
The **naive implementation** takes $O(m \log n + n)$ time. Starting from $\psi[i]$, we perform ψ-jumps, that is, checking the character $FL[\psi^k[i]])$, until, after w jumps, we reach the \$-termination character. Since we know that such a character occurs at position n (where n is the length of the string), then $SA[i] = nw$. This is because it took w jumps to reach the end of the string, meaning that the original position of $FL[i]$ in the string is w characters away from the end, i.e. at position nw. The time complexity of this naive implementation is therefore $O(m \log n + n)$ (binary search + $O(n)$ jumps, because the worst case is that we start from the first letter of the string)
The **speedup** focuses on avoid performing the ψ-jumps, and it is based on the following operations:

1. We define a **bitvector** SAMPLED, which contains the positions of the SA that are multiple of $\log n$ (e.g. if $\log n = 3$ and SA $= 3245610$, then SAMPLED $= 1000101$);
2. We **sample** (i.e. store) SA$[i]$ every $\lceil \log n \rceil$ positions, and we define an array **SSA** (sampled suffix array), which contains the sampled elements (considering the previous example, we have that SSA $= 360$);
3. Finally, the **CSA** is defined by SSA and SAMPLED;
4. The *locate* operation is executed in the following way:

    - If we want to retrieve, for example, SA[1], we know that it was not sampled, since SAMPLED[1] $= 0$, thus we must use ψ at most $\log n$ times, in order to find the sampled position;
    - If we want to retrieve, for example, SA[4], then we know that the value was sampled, since SAMPLED[4] $= 1$, thus SA[4] $=$ SSA[SAMPLED.rank(4)] $= 6$.

Notice that the size of SAMPLED is $n + o(n)$ (bitvector), while SSA contains $\frac{n}{\log n}$ entries, each taking $\log n$ bits, so the overall size is n. Thus, the space of SSA and SAMPLED is in total $\leq 2n + o(n)$.
Finally, the complexity of the *locate* operation is $O((\text{occ} + m) \log n)$

Extracting text using ψ We now see how to extract the suffix starting at position SA[5]. The idea is to store ψ and FL.

Figure 15: Extract text using ψ: the first letters are underlined

Now, the idea is to retrieve the suffix starting at position SA[5] by accessing the array FL in the following way: FL[ψ[1]], FL[ψ[ψ[1]]] etc..:

1. FL[ψ[1]] = FL[5] = A;
2. FL[ψ[ψ[1]]] = FL[ψ[5]] = FL[9] = T;
3. FL[ψ[ψ[ψ[1]]]] = FL[ψ[ψ[5]]] = FL[ψ[9]] = FL[4] = A;
4. FL[ψ[ψ[ψ[ψ[1]]]]] = .. = T.

In this case the complexity is given by $O(\log n + l)$ time, with l be the length of the substring to extract.

4.3.2 Compressing FL

We recall that FL is the sequence of the first letter of each sorted suffix. This means that the characters that FL consists of are sorted as well, and there might be multiple consecutive occurrences of some of them.
For this reason, the same bitmask approach used to store the SSA can be adopted to decrease the space requirements. We consider a bitmask R, we have:

- Just one occurrence per character is stored in an array F;
- The i-th occurrence of the array R is set to 1 if it is the first occurrence of some character.

This allows us to retrieve $FL[i]$ as $F[R.\text{rank}_1(i)]$.

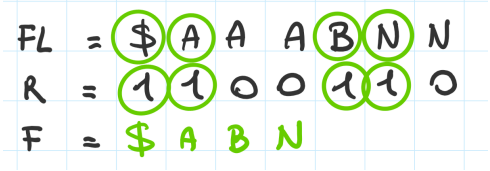


Figure 16: Compressing FL

The total space required by this data structure is $n+o(n)+\sigma \log n$ (bitvector of n bits + auxiliary data structures for $O(1)$ rank and select + σ characters * at most $\log n$ bits to represent each character, because we are assuming that $\sigma \leq n$).

4.3.3 Compressing ψ

The idea is to **store** each **increasing sub-sequence** of ψ by using **Elias-Fano**. A key point in compressing ψ is to note that each cluster is an increasing sequence of numbers, which are indexes of SA that refer to suffixes starting with the same character. Why are such sequences certainly increasing? Intuitively, the reason is that SA contains the starting indexes of all the sorted suffixes, meaning that suffixes starting with the same character are also lexicographically ordered.
In this sense, we have that:

$$\text{space } = \sum_{c \in \Sigma} |\text{EF}(\psi_c)|$$

, where $\text{EF}(\psi_c)$ represents the Elias-Fano representation of the subsequence ψ_c.
In general, $\text{EF}(\psi_c)$ contains n_c numbers, with n_c be the number of occurrences of c in T. Thus, each ψ_c is on the universe $[1, n]$ and contains n_c integers. Thus, we have that:

$$\begin{aligned}
\text{space } &= \sum_{c \in \Sigma} n_c \log\left(\frac{n}{n_c}\right) + O(n_c) \\
&= \sum_{c \in \Sigma} n_c \log\left(\frac{n}{n_c}\right) + \sum_{c \in \Sigma} O(n_c) \\
&= n \sum_{c \in \Sigma} \frac{n_c}{n} \log\left(\frac{n}{n_c}\right) + \sum_{c \in \Sigma} O(n_c) \\
&= nH_0(S) + O(n) \text{ bits}
\end{aligned}$$

, where $H_0(S) \leq \log \sigma$ ($\log \sigma = H_{\text{wc}(S)}$) if the frequencies are far from the uniform distribution.

Theorem. *CSA takes $nH_0 + O(n)$ bits and supports:*

- *The **count** operation in $O(m \log n)$ time, as the SA;*
- *The **locate** operation in $O((occ + m) \log n)$ time (sampling SA), i.e. more than the SA;*
- *The **extract** operation in $O(\log n + l)$ time (sampling SA^{-1}).*

Notice that in the case of a CSA, the space required for storing the human genome becomes 1GiB, so it is optimal.

4.4 Wavelet trees

Wavelet trees are powerful **structures** that permit to implement **access**/**rank**/**select** on general strings using **bitvectors**. This is done within **optimal space** and with a $\log \sigma$ slowdown of all query times. Finally, this structure finds countless applications (e.g. FM index, 2D geometric reporting, quantile queries, ...).

4.4.1 Sequence data structure

A static sequence (or string) data structure over a length-n string $S \in \Sigma^n$ is a data structure supporting efficiently the following operations:

- **Access**: $S[i]$;
- **Rank**: $S.\text{rank}_c(i) = |\{j \leq i : S[j] = c\}|$, i.e. the number of letters c before position i;
- **Select**: $S.\text{select}_c(i) = \min\{j : \text{rank}_c(j) = 1\}$, i.e. the position of the i-th character c in S.

4.4.2 Wavelet tree - definition by example

We now consider the S = MISSISSIPPI, and we consider the following (prefix-free) encoding of the alphabet:

- m = 00;
- i = 01;
- s = 10;
- p = 11.

Notice that this encoding is particularly naive, since it uses $\log \sigma$ bits for each character. The corresponding wavelet tree is represented in Picture 4.4.2.

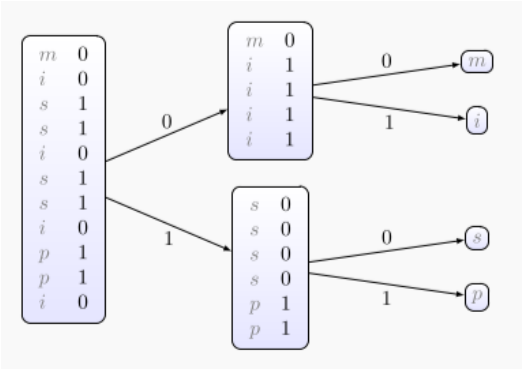


Figure 17: Wavelet tree using naive encoding

Some notes:

- The **tree** is perfectly balanced and it overall contains $\log \sigma$ **levels**: at the root we store a bitvector containing the first bit of the encoding of each character, i.e. 0 for m and i, 1 for s and p;
- Then, we recursively **split** the **bitvector** into two bitvectors, one containing the characters beginning with **0**, and the other one containing the characters beginning with **1**;
- At level i, we **store** the **bitvector** containing to the i-th bits of the encoding for each character;
- The recursion is repeated until we reach a node which stores the character explicitly, the leaf.

4.4.3 Required space

The peculiarity of this structure is that only the **bits**, the **tree topology** (which can be encoded implicitly using wavelet matrices) and the **leaves** are explicitly stored. The overall space of the structure is given by:

$$n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$$

bits, where:

- $n \log \sigma$ is given by the space required by the **bitvectors** ($\log \sigma$ levels and n bits for each level);
- $o(n \log \sigma)$ is an **overhead** for executing the operations we implement;
- $O(\sigma \log n)$ is given by the space required by the **edges** and the corresponding **labels** of the tree.

4.4.4 Access/Rank/Select operations

We now consider the operations on the wavelet tree: in general, all the three operations on strings can be **easily implemented** using access/rank/select operations on the **bitvectors**.

Access The access can be implemented using the access and rank operations on the bitvectors of the tree. An example is showed in Picture 4.4.4.

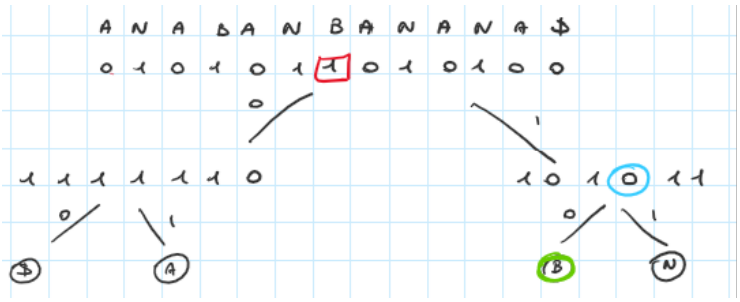


Figure 18: Access on wavelet tree

Access[7] is solved in the following way:

1. We retrieve $\text{rank}_1(7) = 4$;
2. We follow the right path (1) and we compute $\text{rank}_0(4) = 2$;
3. We follow the left path (0) and we reach a leaf containing Access[7] = B.

The **complexity** of the access operation using the bitvector DS is $\Theta(\log \sigma)$.

Rank Suppose we want to compute $\text{rank}_\text{N}(7)$. Then the operations are the following:

1. We compute $\text{rank}_1(7) = 4$;
2. We follow the right path (1) and we compute $\text{rank}_1(4) = 2$;
3. We follow the right path (1) and we reach a leaf, so the solution is 2.

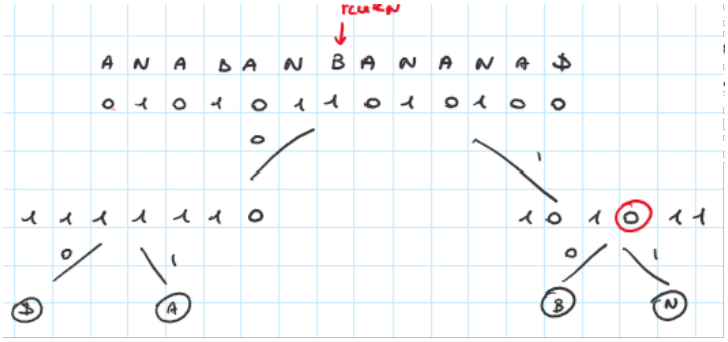


Figure 19: Rank on wavelet tree - example 1

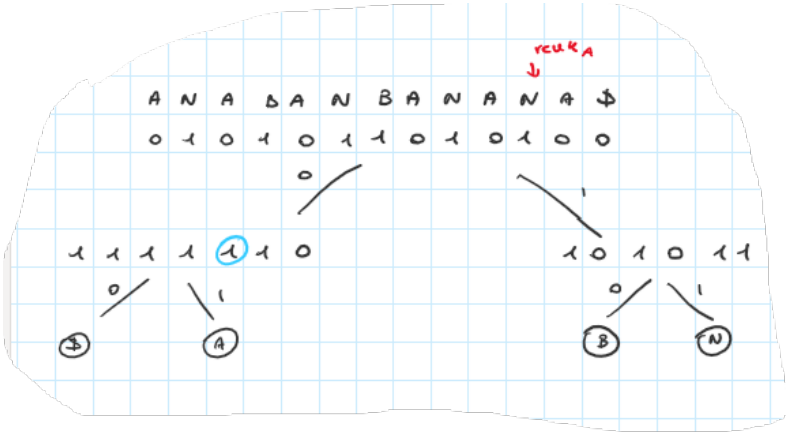


Figure 20: Rank on wavelet tree - example 2

Another example is provided in Picture 4.4.4.
Notice that the **choice** of the **path** and of the **base** of the rank is given by the **encoding of the letter** for which we're computing the rank operation: in this case the letter N is encoded as 11, so we follow path 1 and compute rank_1, twice.
In this case we want to compute $\text{rank}_\text{A}(11)$. Then:

1. We compute $\text{rank}_0(11) = 5$;
2. We follow the left path (0) and we compute $\text{rank}_1(5) = 5$;
3. We follow the right path (1) and we reach a leaf, so the solution is 5.

Notice that in this case we first compute rank_0 and follow path 0, then we compute rank_1 and follow path 1, since A is encoded as 01.
The rank operation has a **complexity** $\Theta(\log \sigma)$.

Select Suppose now we want to compute $\text{Select}_\text{A}(3)$, then:

1. We start from the leaf containing letter A, we follow edge 1 and we compute $\text{Select}_1(3) = 3$;
2. We follow edge 0 and we compute $\text{Select}_0(3) = 5$, which results to be the solution.

Again, we first follow edge 1 and compute Select_1 and then follow edge 0 and compute Select_0 because A is encoded as 01.
The select operation has a **complexity** $\Theta(\log \sigma)$.

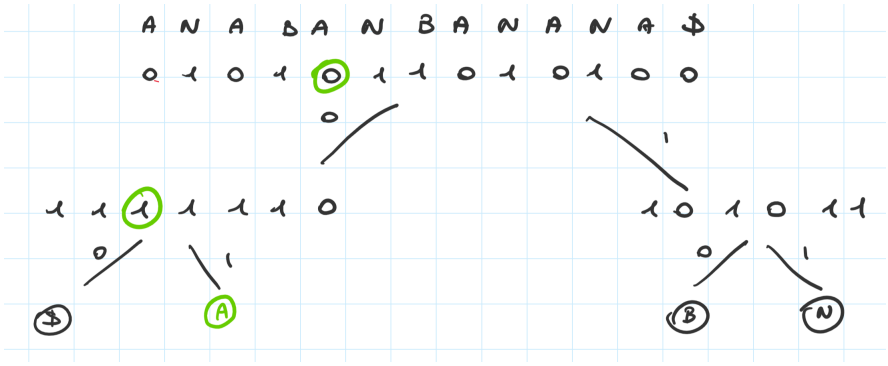


Figure 21: Select on wavelet tree

4.4.5 Using Huffman encoding

What if we used a different encoding for our alphabet? For example, the Huffman encoding (which is prefix-free)? In this case we would have:

- m = 001;
- i = 01;
- s = 1;
- p = 000.

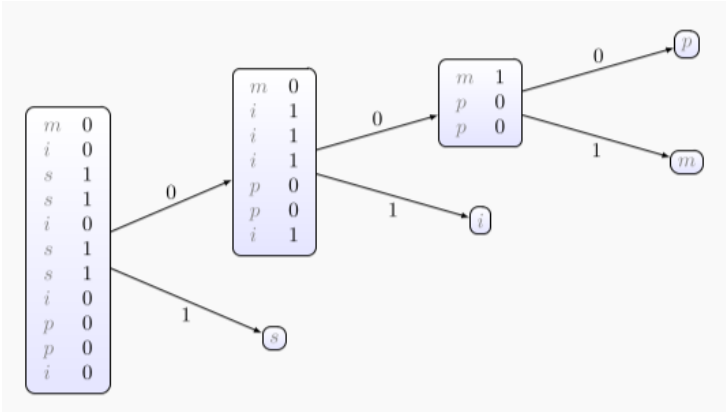


Figure 22: Wavelet tree using Huffman encoding

In this case we have that the **space** required by the structure is $n(H_0 + 1) + o(n(H_0 + 1)) + O(\sigma \log n)$ bits, which is **smaller** than the **space required** by the wavelet tree **using** the **naive encoding**. Moreover, in this case the operations take:

- $O(\sigma)$ time in the **worst case**, which is **higher** than the complexity of the operations executed in wavelet trees that use naive encoding;
- $O(H_0)$ time in the **average case**, which derives from $\sum_{c \in \Sigma} \frac{n_c}{n} \sigma_c$, where n_c represents the frequency of char c, while σ_c represents the length of the encoding of char c. In this case the complexity is **better** than the operations executed in wavelet trees that use naive encoding.

4.4.6 Possible improvement

A possible improvement can be reached by exploiting:

- Naive encoding;
- Zero-order compressed bitvectors.

In this case we have:

- **Space** complexity which is $nH_0 + o(n \log \sigma) + o(\sigma \log n)$ bits;
- **Time** complexity for the queries which is $O(\log \sigma)$.

4.5 Burrows-Wheeler transform

With the CSA we achieved nH_0 (space). What about nH_k? We use an apparently different (but actually equivalent) idea: the **Burrows-Wheeler transform**.
Given a string $S =$ mississippi\$, the idea is the following:

1. We consider all the possible **permutations** of the string S;
2. We **sort** the permutations in **lexicographic order**, and we create the **BWT matrix**. Since it takes too much space, we only store the 1st and the last column, which are denoted with **F** and **L**. The BWT matrix of $S =$ mississippi\$ is represented in Picture 2.

F											L
\$	m	i	s	s	i	s	s	i	p	p	i
i	\$	m	i	s	s	i	s	s	i	p	p
i	p	p	i	\$	m	i	s	s	i	s	s
i	s	s	i	p	p	i	\$	m	i	s	s
i	s	s	i	s	s	i	p	p	i	\$	m
m	i	s	s	i	s	s	i	p	p	i	\$
p	i	\$	m	i	s	s	i	s	s	i	p
p	p	i	\$	m	i	s	s	i	s	s	i
s	i	p	p	i	\$	m	i	s	s	i	s
s	i	s	s	i	p	p	i	\$	m	i	s
s	s	i	p	p	i	\$	m	i	s	s	i
s	s	i	s	s	i	p	p	i	\$	m	i

Figure 23: BWT matrix

3. The BWT(S) is given by the last column L, so BWT(S) = ipssm..ssi. Notice that we can easily reconstruct the first column of the matrix, $F =$ \$AAABNN, because it is the lexicographically ordered representation of L (this is due to the fact that L contains all the characters in T and that we define the BWT matrix in such a way that the rows are the lexicographically ordered rotations of T, i.e. F is lexicographically ordered).

A very important property of the BWT is the **LF property**, which states the following: let $c \in \Sigma$, then the i-th occurrence of c in L corresponds to the i-th occurrence of c in F. An example of this property is shown in Picture 4.5.
Given BWT(S), in order to find S we must perform the following operations:

1. Construct the column L;
2. Repeatedly apply LF property to re-construct S.

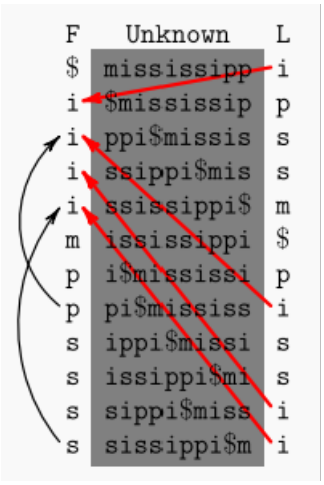


Figure 24: LF property (red arrows)

Example. *Let $BWT(S) = ESEGSMRATCS\$EE$, and we want to find S, given the lexicographic order $\$,A,C,E,G,M,R,S,T$.*

1. *The column F is given by $\$\ A\ C\ E\ E\ E\ E\ G\ M\ R\ S\ S\ S\ T$;*
2. *We construct the string S starting by the end: the first char in F is \$, and the corresponding char in L is E, so the last char of S (besides \$) is E;*
3. *Then, using the LF property, we retrieve the first E in F, and in this case the corresponding char in L is G, so the second last char of S is G, and so on..*

Finally, we re-construct $S = SECRETMESSAGE\$$.

4.5.1 Space required

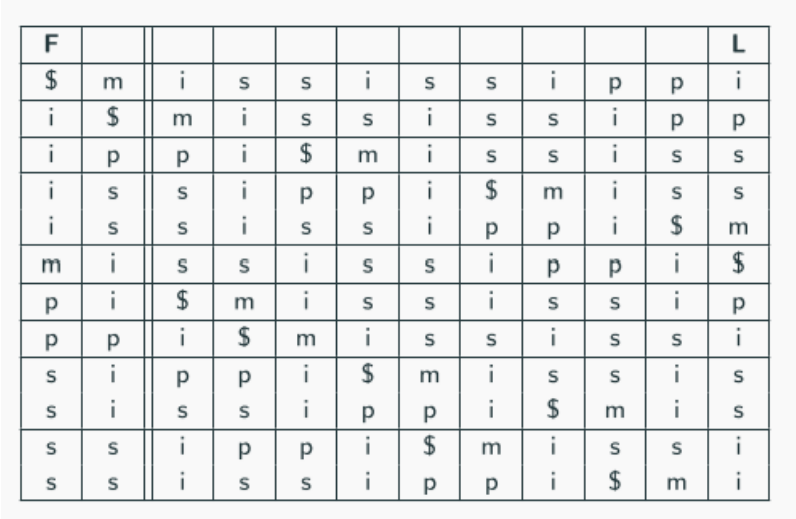
Before analyzing the way in which we can compress the BWT, we define T_w as the string of characters that precedes w in the text (from left to right). For example, if the $S = \text{BANANA}$, then $T_{\text{AN}} = \text{BN}$.
An important **property** of the BWT is that the characters are **partitioned by context**. For example, if we consider $k = 2$, we can isolate the first two columns of the BWT matrix, and notice that:

- **Characters** are **partitioned** by context;
- The **elements** in the column L are the **characters** that precede the **one** of the first **two columns**, i.e. they represent T_W, where W is the string in the first two columns.

  In this example, $A = T_{\$B}$, $N = T_{A\$}$ etc..

Now the idea is to **compress** each **context** independently using a **zero-order compressor**, e.g. **Wavelet Tree**, and use $n_w H_0(T_w) + o(n_w \log \sigma) + O(\sigma \log n_w)$ bits for each context. Notice that n_w represents the number of occurrences of w in T. We can prove that using this compression, we obtain H_k bits for space:

$$\begin{aligned} \text{space} &= \sum_{w \in \Sigma^*} n_w H_0(T_w) + o(n_w \log \sigma) + O(\sigma \log n_w) \\ &\leq nH_k + o(n \log \sigma) + O(\sigma^{k+1} \log n) \\ &\leq nH_k + o(n \log \sigma) + O(\sigma \log n) \text{ bits} \end{aligned} \tag{1}$$



F											L
$	m	i	s	s	i	s	s	i	p	p	i
i	$	m	i	s	s	i	s	s	i	p	p
i	p	p	i	$	m	i	s	s	i	s	s
i	s	s	i	p	p	i	$	m	i	s	s
i	s	s	i	s	s	i	p	p	i	$	m
m	i	s	s	i	s	s	i	p	p	i	$
p	i	$	m	i	s	s	i	s	s	i	p
p	p	i	$	m	i	s	s	i	s	s	i
s	i	p	p	i	$	m	i	s	s	i	s
s	i	s	s	i	p	p	i	$	m	i	s
s	s	i	p	p	i	$	m	i	s	s	i
s	s	i	s	s	i	p	p	i	$	m	i

Figure 25: Partitions in BWT

,where:

- n represents the length of the text;
- $O(\sigma^{k+1} \log n)$ is not very nice, since it explodes exponentially with k. In this sense, we try to minimize this quantity, trying to have k as large as possible: $k = \max\{\epsilon \log_\sigma n - 1, 0\}$, with $0 < \epsilon < 1$. Then, $O(\sigma^{k+1} \log n)$ becomes $\sigma^{k+1} \log n = \sigma^{(\epsilon \log_\sigma n - 1)+1} \log n = n^\epsilon \log n$, and if $\epsilon = 0.95$, then $n^\epsilon \log n = o(n)$.

4.5.2 Rank operation

We now take into consideration the **rank** operation on a H_k compressed BWT. The idea is to keep the following structures:

- A **bitvector** B which indicates whether the corresponding element in the column L is the first elements of the partition or not. This bitvector has size $nH_0(B)+o(n) = O(\sigma^k \log n)+ o(n)$, where:
  - σ^k represents the number of 1's in the bitvector;
  - $\log n$ represents the frequency of the 1's.

  Thus, the size of the bitvector is $O(\sigma^k \log n) + o(n)$ bits;
- For each partition, we **pre-compute the rank** for each of the characters of the string, w.r.t. column L. This structure takes $\sigma^{k+1} \log n$ bits of space.

By using these two structure, we're able to solve the Rank operation. Suppose our string is $S =$ BANANA\$, and we have the BWT matrix represented in Picture 4.5.2.
As we can see:

- Each char in the partitions represent T_w, for each string w in column F;
- We have the bitvector B in light blue;
- We have the structure for pre-computed ranks for each char in green.

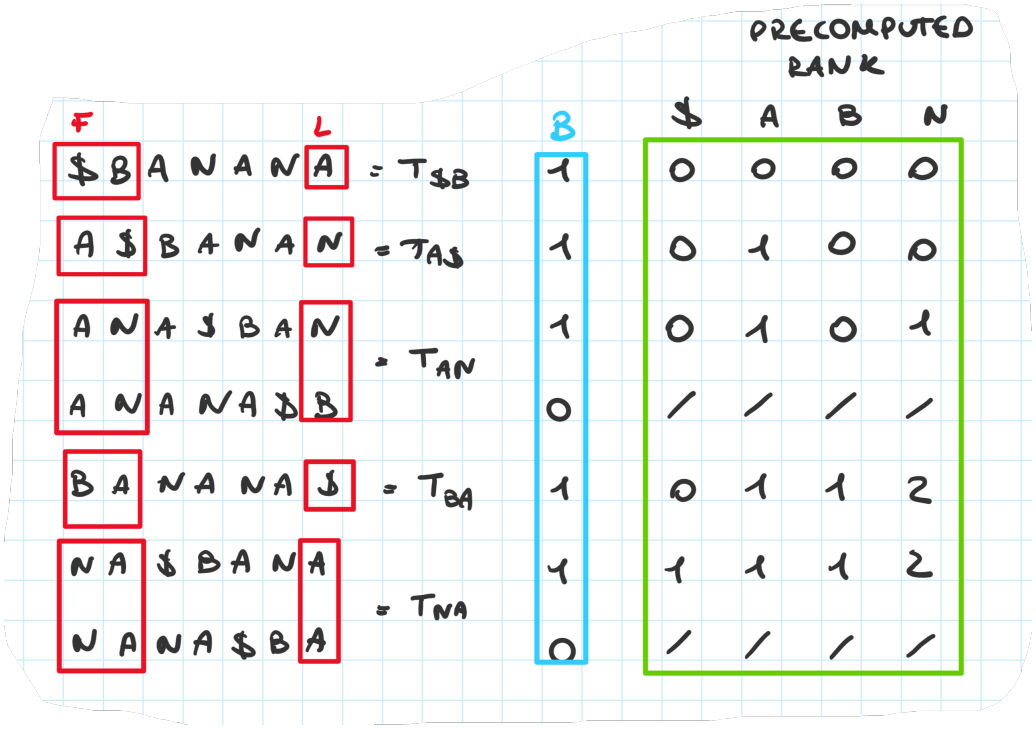


Figure 26: Rank operation

Now, the operations for computing $\text{Rank}_N(4)$ are the following:

1. We access $B[4] = 0$, and we count the number of 1's before $B[4]$, in this case this quantity is equal to 3;
2. Then, we access the 3rd partition, in this case T_{AN}, and we retrieve the pre-computed rank for char N, which is 1;
3. Finally, we add the number of N's before position 3, which is 1, and we obtain 2, the solution of $\text{Rank}_N(4)$.

The **complexity** of the rank operation on a H_k compressed BWT is $O(\log \sigma)$ time (which corresponds to the complexity of the rank operation on a WT: recall that the contexts of the BWT are stored in a WT).

Theorem. *We can store BWT(T) in* $nH_k + o(n \log \sigma) + O(\sigma \log n)$ *bits and answer rank query in* $O(\log \sigma)$ *time for* $k = \max\{0, \epsilon \log_\sigma n - 1\}$.

4.6 FM index

The idea of the **FM index** is to **store** the column **F** using an **array** C which stores the position preceding the first occurrence of each char (such array takes $\sigma \log n$ bits, one integer per character). In the previous example, we have:

- $C[\$] = 0$, since \$ is stored in F in position 1;
- $C[A] = 1$, since A is stored in F in position 2;
- etc..

Now the question is: can we retrieve $LF[i]$ using the array C and the BWT?
The answer is yes, since:

$$LF[i] = \text{BWT.rank}_{\text{BWT}[i]}(i) + C[\text{BWT}[i]]$$

For example, if $i = 6$, then:

$$LF[6] = \text{BWT.rank}_A(6) + C[A] = 2 + 1 = 3$$

and this is true, since the second occurrence of A (which is in position 6 in L) in F is in position 3.

4.6.1 Backward search

In order to retrieve the occurrence of a pattern (e.g. "si") in a FM index we can perform the following operations:

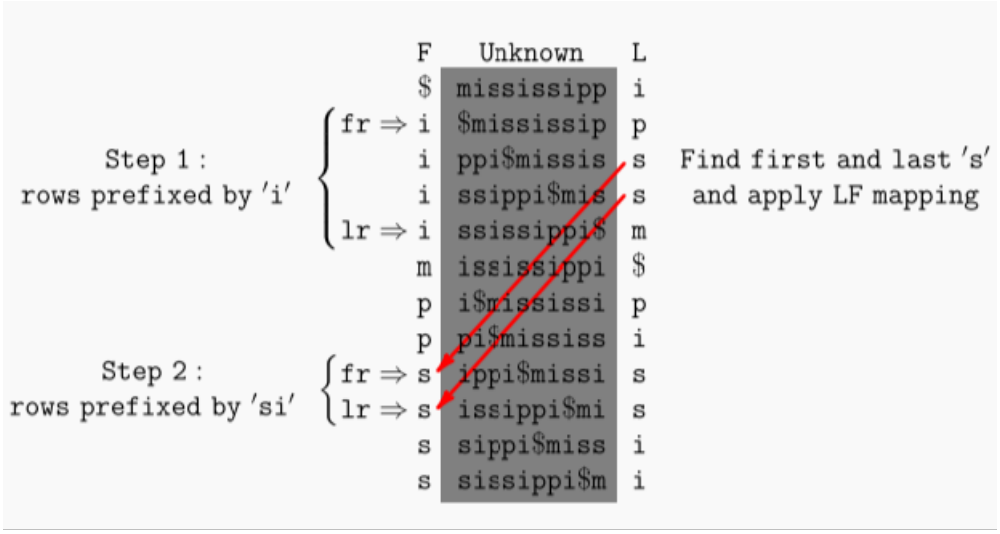


Figure 27: Backward search

1. We find the rows prefixed by "i" in column F;
2. We find the first and last "s" in L within the range of rows retrieved in (1);
3. We apply LF mapping on the first and last "s", and we retrieve the pattern "si".

Each step can be performed in $O(\log \sigma)$ time using wavelet trees, so the overall complexity is $O(m \log \sigma)$, where m is the length of the pattern we're searching.

4.6.2 Complexities

Theorem. *FM index takes* $nH_k + o(n \log \sigma)$ *bits for* $k = \max\{0, \epsilon \log_\sigma n - 1\}$, *and supports:*

- *Count (i.e. apply backward search and return the cardinality), in* $O(m \log \sigma)$ *time, i.e. less than a CSA (*m *represents the length of the pattern, while* $\log \sigma$ *represents the cost of the steps of the backward search);*
- *Locate (i.e. apply backward search) in* $O(m \log \sigma + occ \log^{1+\epsilon} n)$ *time (*$m \log \sigma$ *is given by the backward search, and we also need a sampling of SA every* $\frac{\log^{1+\epsilon} n}{\log \sigma}$ *positions);*

- *Extract (i.e. backward search with initial indexes in input) in* $O(l \log \sigma + \log^{1+\epsilon} n)$ *time, where* $l \log \sigma$ *is given by the cost of the backward search with* l *initial indexes.*

Notice that FM-indexes had a huge impact in medicine and bioinformatics: if today you get your own genome sequenced, it will be analyzed using software based on the FM-index.