Complexity and Information Theory

Pietro Marcatti

First Semester 2022/2023

1 Introduction

The information comunicated with a sentence, or any other way, is always dependant on the context. Same goes for the quality for the information. If an event has a low probability of occurring in a given context the information it provides is high, inversely if the probability is high the information is little.
We could try to describe information with a simple inverse model:

$$Information = \frac{1}{p(E)}, \quad \text{E=event}$$

$$p(E) \rightsquigarrow 0 \quad Information \rightsquigarrow \infty$$

$$p(E) \rightsquigarrow 1 \quad Information \rightsquigarrow 1$$

Instead if we were to adopt a logaritmic model:

$$Information = \log \frac{1}{p(E)} = -\log p(E), \quad \text{E=event}$$

$$p(E) \rightsquigarrow 0 \quad Information \rightsquigarrow \infty$$

$$p(E) \rightsquigarrow 1 \quad Information \rightsquigarrow 0$$

Let's consider an alphabet as a group of possible events for which we have a probability distribution:

$$A = a_1, a_2, a_3, \dots, a_k$$

$$P = p_1, p_2, p_3, \dots, p_k$$

meaning that p_i is the probability to observe the event a_1. We can then calculate the average quantity of information as follows:

$$\sum_{i=1}^{k} p_i \cdot (-\log p_i) = \underset{ShannonEntropy}{\mathcal{H}(P)} \tag{1}$$

1.1 Coin Flip Example

Fair Coin

$$A = \{H, T\}$$

$$P(H) = \frac{1}{2}, \quad P(T) = \frac{1}{2}$$

$$\mathcal{H}(P) = \frac{1}{2} \cdot (-\log \frac{1}{2}) + \frac{1}{2} \cdot (-\log \frac{1}{2}) = 1$$

Every time we filp the fair coin we expect to gain a bit of information. Unfari Coin

$$A = \{H, T\}$$

$$P(H) = \frac{9}{10}, \quad P(T) = \frac{1}{10}$$

$$\mathcal{H}(P) = \frac{9}{10} \cdot (-\log \frac{9}{10}) + \frac{1}{10} \cdot (-\log \frac{1}{10}) \rightsquigarrow 0.32$$

1.2 Properties of Entropy

- $\mathcal{H}(P) \geq 0$
- $\mathcal{H}(P) = 0$ if there is an event with probability 1
- $\mathcal{H}$ is continuous with respect to P
- If an event gets split the entropy should be additive
- $\mathcal{H}(p_1, p_2, \dots, p_k) \leq \log k = \mathcal{H}(\frac{1}{k}, \frac{1}{k}, \dots, \frac{1}{k})$

To prove the last property we can first show that for a equi-distribution we get $\mathcal{H} = \log k$. To prove that this is also the maximum we use Jensen disequality:

$$\forall f, f''(x) < 0 \quad in\ [a, b], (a, b) \in \mathbb{R}^2$$

$$f\Big(\sum_{i=1}^{k} \lambda_i \cdot x_i\Big) \geq \sum_{i=1}^{k} \lambda_i \cdot f(x_i), \quad \sum \lambda_i = 1$$

2 Data Compression

Before we can talk about data compression it is necessary to explore the meaning and the functiong of encodings. Given two alphabets $A = a_1, a_2, \ldots, a_k$ and $B = b_1, b_2, \ldots, b_k$ an encoding is function:

$$\varphi : A^* \longrightarrow B^* \tag{2}$$

For a function to be a suitable encoding it must be at least an injective function (one-way function). In information theory and in data encoding in particular we refer to injective encoding as **uniquely decodable** encodings. Prefix codes are uniquely decodable codes that can be decoded without delay and are written in the form:

$$A = \{a_1, a_2, \ldots, a_k\} \xrightarrow{\varphi} B = \{b_1, b_2, \ldots, b_D\} \tag{3}$$

where $\varphi_i = |\varphi(a_i)|$

2.1 Kraft-McMillen 1958 Inverse Theorem

If φ is a uniquely decodable code, then:

$$\sum_{i=1}^{k} D^{-l_i} <= 1 \tag{4}$$

D is the cardinality of the output alphabet, l_i is the lenght of the encoding for a_i. We can quickly see that, there can't be an uniquely decodable code that uses three encodings of length 1 to map over the binary alphabet:

$$A = \{a, b, c\}, \quad B = \{0, 1\}, \quad l_a = l_b = l_c = 1 \longrightarrow \frac{1}{2} + \frac{1}{2} + \frac{1}{2} > 1 \tag{5}$$

Instead, if we were to insist on using a length 10 encoding for everyone of the input characters:

$$A = \{a, b, c\}, \quad B = \{0, 1\}, \quad l_a = l_b = l_c = 10 \longrightarrow \frac{1}{2^{10}} + \frac{1}{2^{10}} + \frac{1}{2^{10}} << 1 \tag{6}$$

2.1.1 Proof 1

Assume the code is uniquely decodable and it is a prefix code, we can sort the input alphabet so that the length of the encoding satisfies the following:

$$a_1 \leq a_2 \leq \ldots \leq a_k$$

A character with an enconding of length i generates a sub-tree rooted in a_i with height $l - l_i$. This sub-tree contains $D^(l - l_i)$ nodes that cannot be used in the encoding because they would share an encoded prefix.

Insert diagram
In the original three the following number of leaves

$$D^l - \sum_{i=1}^{k-1} D^{l-l_1}$$

The $k-1$ term that bounds the sum is needed because we need to make sure that one last leaf is available to assign it to the last character a_k.

$$D^l - \textstyle\sum_{i=1}^{k-1} D^{l-l_i} \geq 1$$

Group by D^l

$$D^l(1 - \textstyle\sum_{i=1}^{k-1} D^{-l_i}) \geq 1$$

Divide by D^l

$$1 - \textstyle\sum_{i=1}^{k-1} D^{-l_i} \geq D^{-l}$$

D^{-l} is the last term of the sum

$$1 \geq \textstyle\sum_{i=1}^{k} D^{-l_i}$$

2.1.2 Proof 2 - General case

Let's define $N(n,h)$ the number of strings of alphabet A^n having encoding length h. It must be true that $N(n,h) \leq D^h$ because φ is uniquely decodable.

$$D^{-l_1} + D^{-l_2} + \ldots + D^{-l_k} \leq 1 \tag{7}$$

$\forall n, n \in \mathbb{N}$ let's consider the object $(D^{-l_1} + D^{-l_2} + \ldots + D^{-l_k})^n$. If written as a product it takes the following form:

$$D^{-l_1 \cdot n} + D^{-l_1 \cdot (n-1) - l_2} + \ldots + D^{-l_k \cdot n}$$

As n tends towards infinity the exponent will behave differently depending on the value of the base. If the base is < 1 it will be bound between 0 and 1, and in any case < 1. If the base is > 1 it will grow towards infinity faster than any polynomial function. Lastly, if the base is $= 1$ it will remain constant. To prove the theorem we will demonstrate that the object is dominated by a linear function, meaning that it cannot be in the second case. All of the members' exponents follow this chain of disequalities: $l_1 \cdot n \leq exp \leq l_k \cdot n$. That is, because we sorted the character by encoding length, 1 being the shortest while k the longest.

$$\begin{aligned}\underset{prob.0}{N(n,1)}D^{-1} + \ldots + N(n, l_1 \cdot n)D^{l_1 \cdot n} + \ldots + N(n, l_k \cdot n)D^{-l_k \cdot n} \\ \leq D^1 D^{-1} + \ldots + D^{l_k} D^{-l_k} \leq l_k \cdot n\end{aligned}$$

Since the object is dominated by $l_k \cdot n$ which is linear, it must mean that it also grows at most linearly, meaning that the base of the exponent must be ≤ 1.

2.2 Kraft-McMillen Direct Theorem

If given $l_1, l_2, \ldots, l_k$ and D such that $\sum_{i=1}^{k} D^{-l_i} \leq 1$ there must exist a prefix code φ having $l_1, l_2, \ldots, l_k$ as lengths of the encoding.

2.2.1 Proof

Proof can be given by construction by producing the complete D-ary tree

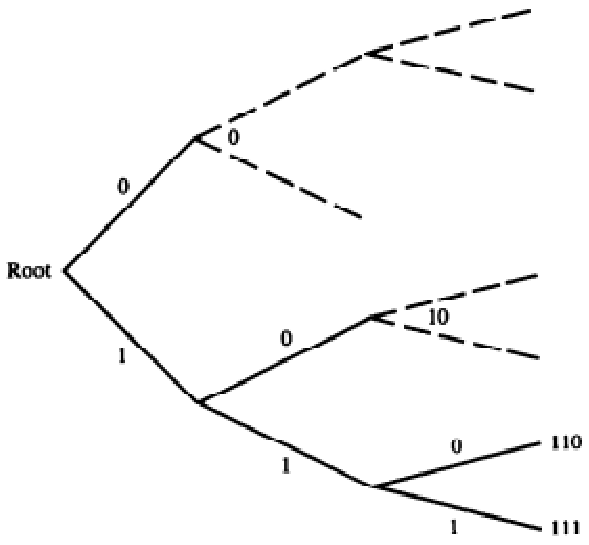


Figure 5.2. Code tree for the Kraft inequality.

Observation: Prefix codes compress as good as any other uniquely decodable code.

2.2.2 Average expected code length - Definition

Given the input alphabet A, P the probability and B the output we define the Expected Length as:

$$EL(\varphi) = \sum_{i=1}^{k} p_i \cdot l_i = \sum_{i=1}^{k} p_i |\varphi(a_i)| \tag{8}$$

Our goal is to find prefix codes that minimize EL.

2.3 1st Shannon Theorem 1948

If φ is uniquely decodable code then $EL(\varphi) \geq \mathcal{H}_D(P) = \sum_{i=1}^{k} p_i \cdot log_D p_i$ The proof relies on Kraft-McMillen theorem and this simple logarithmic property:

$$log_e x \leq x - 1, \quad -log_e x \geq 1 - x$$

2.3.1 Proof

$$\begin{aligned}
EL(\varphi) - \mathcal{H}_D(P) &= \textstyle\sum_{i=1}^{k} p_i \cdot l_i + \sum_{i=1}^{k} p_i \cdot log_D p_i \quad \Big\downarrow \textit{Group by } p_i \textit{ and force } l_i \textit{ as } log_D \\
&= \textstyle\sum_{i=1}^{k} p_i \cdot log_D(D^{l_i} p_i) \quad \Big\downarrow \textit{Change to base e, take out the common term} \\
&= \tfrac{1}{log_e D} \cdot \textstyle\sum_{i=1}^{k} p_i \cdot log_e(D^{l_i} p_i) \quad \Big\downarrow \textit{Group by } -1 \\
&= \tfrac{-1}{log_e D} \cdot \textstyle\sum_{i=1}^{k} p_i \cdot log_e \tfrac{1}{D^{l_i} p_i} \quad \Big\downarrow \textit{Using the above property} \\
&\geq \tfrac{-1}{log_e D} \cdot \textstyle\sum_{i=1}^{k} p_i \cdot (\tfrac{1}{D^{l_i} p_i} - 1) \\
&= \tfrac{-1}{log_e D} \cdot \left(\textstyle\sum_{i=1}^{k} \tfrac{1}{D^{l_i}} - \underbrace{\cancel{\textstyle\sum_{i=1}^{k} p_i}}_{=1} \right) \quad \Big\downarrow \textit{Group by } -1 \\
&= \tfrac{1}{log_e D} \underbrace{\left(1 - \textstyle\sum_{i=1}^{k} \tfrac{1}{D^{l_i}} \right)}_{KMM \longrightarrow \geq 0} \geq 0
\end{aligned}$$

An optimum code achieves the equality between the expected lenght and the entropy of the distribution.

2.4 Shannon Code

Let's recall the definition of expected lenght and entropy over the probability P and alphabet with cardinality D:

$$EL(\varphi) = \sum_{i=1}^{k} p_i \cdot l_i$$

$$\mathcal{H}_D(P) = \sum_{i=1}^{k} p_i \cdot (-\log_D p_i)$$

Observation: $\log_D \frac{1}{p_i} \in \mathbb{R}, \quad l_i \in \mathbb{N} \longrightarrow l_i = \lceil \log_D \frac{1}{p_i} \rceil$

If $\sum_{i=1}^{k} D^{-l_i} \leq 1$ then we can use the Direct Theorem to prove φ, with encoding lenghts $l_1, l_2, \dots l_k$, exists. The encoding is obtained with a greedy strategy on the D-ary tree:

$$\sum_{i=1}^{k} D^{-l_i} = \sum_{i=1}^{k} D^{-\lceil \log_D \frac{1}{p_i} \rceil} \leq 1$$

$$\lceil \log_D \frac{1}{p_i} \rceil = \log_D \frac{1}{p_i} + \beta_i \quad 0 \leq \beta_i < 1$$

$$\sum_{i=1}^{k} D^{\log_D p_i - \beta_i} = \sum_{i=1}^{k} D^{\log_D p_i} \cdot \frac{1}{D^{\beta_i}} = \sum_{i=1}^{k} p_i \cdot \frac{1}{\underbrace{D^{\beta_i}}_{\geq 1}} \leq 1$$

Example: $A = \{a, b\}$ $B = \{0, 1\}$ $P = \left\{1 - \frac{1}{32}, \frac{1}{32}\right\}$ $l_b = \log_2 2^5 = 5$ $\lceil \log_2 1 - \frac{1}{32} \rceil = 1 = l_a$ But this is not efficient because we are assigning unnecessarily long codes to characters with low probability.

2.4.1 Sub-Optimality of Shannon Codes

We can demonstrate that Shannon Codes are suboptimal:

$$\begin{aligned}
EL(S) &= \sum_{i=1}^{k} p_i(-\log p_i + \beta_i) \\
&= \sum_{i=1}^{k} p_i(-\log p_i) + \sum_{i=1}^{k} p_i\beta_i \\
&= \mathcal{H}_D(P) + \sum_{i=1}^{k} p_i\beta_i \leq \mathcal{H}_D(P) + 1 \qquad \sum_{i=1}^{k} p_i \cdot \beta_i \leq 1
\end{aligned}$$

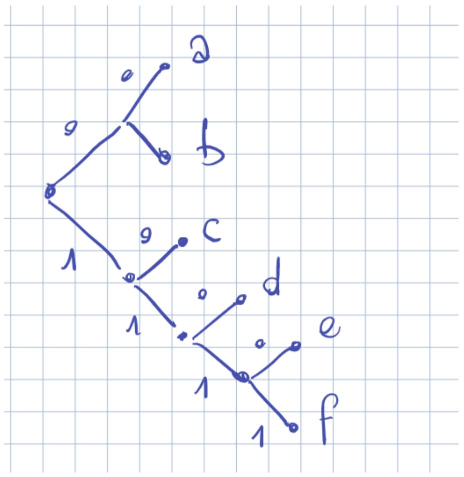
$$\mathcal{H}_D(P) \leq EL(\varphi) \leq \mathcal{H}_D(P) + 1$$

2.5 Shannon-Fano Code

Given the alphabet A with characters $a_1, a_2, \dots, a_k$ mapping to the probabilities $p_1 \leq p_2 \dots \leq p_k$. The aim is to balance the sum of the probabilities assigned to the branches of the root. To obtain balance we try to minimize this absolute value: $|\sum_{i=1}^{h} p_i - \sum_{i=h+1}^{k} p_i|$. We then repeat the calculation recursively.
Example:

$$A = \{a, b, c, d, e, f\}$$

$$P = \left\{ \underbrace{\frac{40}{100}, \frac{18}{100}}_{\frac{58}{100}}, \underbrace{\frac{15}{100}, \frac{13}{100}, \frac{10}{100}, \frac{4}{100}}_{\frac{42}{100}} \right\}$$



Shannon-Fano is compromising on splitting the characters set in two of equal (or best possible) probability because the problem of splitting a set in equally weighted subsets is NP-Complete.

2.6 Shannon on strings of lenght n over input alphabet

Example: $A = \{a, b\}$ $B = \{0, 1\}$ $P = \left\{\frac{3}{4}, \frac{1}{4}\right\}$ $\mathcal{H}_2(P) = 0.81$ $EL(\varphi) = 1.25$
Let's define the efficiency of a code as follows:

$$Eff. = \frac{\mathcal{H}_D(P)}{EL(\varphi)} \leq 1$$

Applying to our example we get:

$$Eff. = \frac{\mathcal{H}_2(P)}{EL(\varphi)} = 0.64$$

Let's try to calculate the efficiency now with an encoding for pairs of characters:

$$A' = \{aa, ab, ba, bb\} \; P' = \left\{\frac{9}{16}, \frac{3}{16}, \frac{3}{16}, \frac{1}{16}\right\} = P^2$$

$$\mathcal{H}_D(P') = 1.62 = \mathcal{H}_D(P^2) = 2 \cdot \mathcal{H}_D(P)$$

$$EL(\varphi) = 1.93 Eff. = 0.83$$

If we pretend we want to send a message 1000 characters long, with the single character encoding we expect a message lenght of 1.25k characters, Instead with the pairs encoding we expect $\frac{1.93}{2} \rightsquigarrow 0.97k$ characters.

Property: given two events x, y independent $\mathcal{H}(x \wedge y) = H(x) + H(y)$ Proof:

$$\begin{aligned} H(x \wedge y) &= -\sum_{i,j} p_i \cdot q_j \log(p_i \cdot q_j) = -\sum_{i,j} p_i \cdot q_j \log p_i - \sum_{i,j} p_i \cdot q_j \log q_j \\ &= -\underbrace{\sum_j q_j}_{=1} \cdot \sum_i p_i \log p_i - \underbrace{\sum_i p_i}_{=1} \cdot \sum_j q_j \log q_j = \mathcal{H}(x) + \mathcal{H}(y) \end{aligned}$$

From Shannon Theorem we then have $EL(\varphi_n) \geq \mathcal{H}_D(P^n) = n \cdot \mathcal{H}_D(P)$
For Shannon Codes (and all sub-optimal codes) we have:

$$n \cdot \mathcal{H}_D(P) \leq EL(\varphi_n) < n \cdot \mathcal{H}_D(P) + 1$$

If we want to reason in terms of a single input alphabet value we divide the disequality by n. We can see that for $n \rightsquigarrow \infty$ the expected lenght is squeezed between the terms of the disequality.

2.7 Huffman Codes

Given the input alphabet and the probability distribution $A = \{a_1, a_2, \dots, a_k\}$ $P = p_1 \leq p_2 \dots \leq p_k$ We start by taking the two leasts probable characters and we put them at the leaves, creating a phantom character with probability the sum of the two. I keep repeating this process until we obtain a complete tree.

2.8 History

- 1949 Shannon-Fano code
- 1952 Huffman code
- 1970 Huffman becomes popular
- 1977 Lempel and Ziv create the LZ77 Code (Variable length - Block) Uses a dynamic dictionary for compression.
- 1978 They update it with a new version LZ78 with static dictionary Gets patented by Sperry Corporation $\rightarrow$ Unisys 1981
- 1983 Another variant LZW (W - Welch) is a simpler implementation of LZ78. In the same year they request a patent and publish an article in 1984 withouth mentionign its patent.
- 1985 the patent gets actually officialized
- Unix and Compuserve start using the code
- 24/12/1994 Unisys decides to claim the royalties "The GIF Tax"
- 2004 the patent expires
- Unix decides to avoid the taxation by "downgrading" to LZ77

2.9 LZ-77

The idea is to focus on a part of the message i've already codified and i refer to it as search buffer. The portion just in front of the search buffer is the look-ahead buffer. The lenght of both buffers is a parameter of the implementation. I then operate like this: I look for the longest prefix of the look-ahead buffer that also appears as a substring in the search buffer. In reality it's a little more complex as the substring from the search buffer can overflow in the look-ahead buffer. Let's identify as j the position occupied by the first character of the substring and i the position of the first character of the prefix. We then consider the triple (o, l, s):

- o: distance between i and j
- l: length of the prefix-suffix
- s: the first mis-matching character

2.10 Example

input = *babbababbabbabba* Buffer size: 5
Encoding:

1st step S buffer = $\emptyset$, LA buffer = babba, triple = (0,0,b)

2nd step S buffer = b, LA buffer: abbab, triple (0,0,a)

3rd step S buffer: ba, LA-buffer = bbaba, triple= (2,1,b)

4th step S buffer: babb, LA-buffer = ababb, triple = (3,2,a)

5th step S buffer: bbaba la-buffer = bbabb, triple = (5,4,b)

6th step S buffer:bbabb la-buffer= abba, triple= (3,4,@)

Decoding:

1st step @@@@@b

2nd step @@@@@ba

3rd step @@@@@babb

4th step @@@@@babbaba

5th step @@@@@babbababbabb

6th step @@@@@babbababbabbabba

3 Complexity

3.1 Kolmogorov

3.1.1 History

We've talked about Shannon who worked in the US, studied at MIT and Princeton, collaborated with Bell Labs. We've talked about Fano who started at the Politecnico di Torino and then moved to MIT. Then we talked about Huffman who worked at Ohio State University, then founds the University of California Santa Cruz.
Kolmogorov, unlike the other names we've seen so far, does not come from the west. In 1965 he was working at the University of Moscow

3.1.2 Idea

Consider a computational model and define the complexity of a string as the lenght of the shortest program (in that computational model) that can generate that string. We'll use Turing Machines: $m = (k, \Sigma, \delta, s)$

k : finite set of states

Σ : finite alphabet $\quad \triangleright, \sqcup \in \Sigma$

δ : $K \times \Sigma \longrightarrow (K \cup \{yes, no, halt\}) \times \Sigma \times \{\rightarrow, \leftarrow, -\}$

s : $s \in K$ start state

3.1.3 Church-Turing Thesis

All computational models are turing-equivalent.
Turing machines with k tapes and I/O $m = (K, \Sigma, \delta, s)$

- $\delta : \Sigma^k x K \rightarrow \Sigma^k x (K \cup yes, no, halt) x \{\rightarrow, \leftarrow, -\}^k$
- The input tape (1st tape) cannot be modified but we can move backwards as much as possible
- The output tap (last tape) can only go forwards

3.1.4 Universal Machine Theorem

$$\exists u, u \text{Universal Turing Machine} | u(bin(m), x) = m(x)$$

How does the universal turing machine works: its tape is somethink like this: start $[01 \ldots 1010]$ x To be able to go on with the computation if we were in the middle of it, we need to know the configuration of the machine. The configuration is stored in one of the k tapes. We also need the state we are on and the position of the first tape. The tape and the position can be expressed as a triple (q, w, ς)

- q is the current character
- w is the string left of q
- $\varphi$is the string right of q

The Kolmogorov complexity of a string m is denoted by $K_u(x) = \underset{u(bin(m))=x}{min} |bin(m)|$

3.1.5 Observation

Unfortunately this notion is not computable.

$$\nexists\, A \quad \text{Turing Machine} \quad | \quad A(x) = K_u(x)$$

3.1.6 Conditional Kolmogorov Complexity

The Conditional Kolmogorov Complexity of x given y is:

$$K_u(x|y) = \underset{u(bin(m),y)=x}{min} |bin(m)|$$

This means that, obviously, if we provied some more information $K_u(x|y) \leq K_u(x)$. One of the most common information given is $|x|$.

Theorem 1 (Universality of Kolmogorov Complexity) *If $\mathcal{U}$ is a universal computer (Turing Machine), for any other computer $\mathcal{A}$ there exists a constant $c_{\mathcal{A}}$ such that*

$$K_{\mathcal{U}}(x) \leq K_{\mathcal{A}} + c_{\mathcal{A}}$$

for all strings $x \in 0, 1^*$, and the constant $c_{\mathcal{A}}$ does not depend on x.

Proof 1 *Assume that we have a program $p_{\mathcal{A}}$ for computer $\mathcal{A}$ to print x. Thus, $\mathcal{A}(p_{\mathcal{A}}) = x$. We can precede this program by a simulation program $\int_{\mathcal{A}}$ which tells the computer $\mathcal{U}$ how to simulate computer $\mathcal{A}$. Computer $\mathcal{U}$ will then interpret the instructions in the program for $\mathcal{A}$, perform the corresponding calculations and print out x. The program for $\mathcal{U}$ is $p = s_{\mathcal{A}} p_{\mathcal{A}}$ and its length is*

$$l(p) = l(s_{\mathcal{A}}) + l(p_{\mathcal{A}}) = c_{\mathcal{A}} + l(p_{\mathcal{A}})$$

where $c_{\mathcal{A}}$ is the lenght of the simulation program. Hence, for all strings x

$$K_{\mathcal{A}}(x) = \min_{p:\mathcal{U}(p)=x} l(p) \leq \min_{p:\mathcal{A}(p)=x} l(p) + c_{\mathcal{A}} = K_{\mathcal{A}}(x) + c_{\mathcal{A}}$$

The crucial point is that the lenght of this simulation program is independent of the length of x, the string to be compressed. For sufficiently long x, the length of this simulation program can be neglected, and we can discuss Kolmogorov complexity without talking about the constants.

Theorem 2 (Upper bound for conditional kolmogorov complexity)

$$K(x \mid l(x)) \leq l(x) + c$$

Proof 2 *Let's define a program for printing x as the program that says:*

$$\textit{Print the following l-bit sequence: } x_1 x_2 \ldots, x_{l(x)}$$

Note that no bits are required to describe l since l is given. The program is self-delimiting because l(x) is provided and the end of the program is thus clarly defined. The length of this program is $l(x) + c$

An apparently stronger conclusion is found in the book "An introduction to Kolmogorov Complexity and its applications" by M. Li and P. Vitanyi, even though this relies on an important detail. In fact, in this case, the alphabet for the machine is $0, 1, \sqcup$. If we allow ourselves to make this assumption we can write the following.

Theorem 3 (Upper bound for kolmogorov complexity - Li-Vitanyi)

$$K(x) \leq l(x) + c$$

Proof 3 *To obtain this stronger looking upper bound for the Kolmogorov complexity, Li and Vitanyi made an assumption about the tape of the machine and its alphabet. In fact, they consider the input tape of the machine as containing the starting symbol $\triangleright$ followed by all the binary digits of x and after it an infinite sequence of blank spaces $\sqcup$. This way they implicitly give the length of x because the machine won't find a $\sqcup$ under its cursor untile it hasn't traversed all x, gaining the information about it's length.*

Theorem 4 (Upper bound for kolmogorov complexity - Cover)

$$K(x) \leq K(x \mid l(x)) + 2 \log l(x) + c$$

Proof 4 *If the computer does not know $l(x)$ we must have some way of informing the computer when it has to come to the end of the string of bits that describes the sequence. We describe a simple but inefficient method that uses a sequence 01 as a "comma".*
Suppose that $l(x) = n$. To describe $l(x)$, repeat every bit of the binary expansion of n twice; then end the description with a 01 so that the computer knows that it has come to the end of the description of n.
For example, the number 5 (binary 101) will be described as 11001101. This description requires $2\lceil \log n \rceil + 2$ bits. Thus, the inclusion of the binary representation of $l(x)$ does not add more than $2 \log l(x) + c$ bits to the length of the program, and we have the bound in the theorem.

Lemma 1 *The set of strings having a Kolmogorov complexity $< a$ has less than 2^a elements.*

Proof 5 *If x has $K(x) < a$ thene there exists the program m that produces x and $|m| < a$. Each m produces at most 1 string. The question then shifts to how many programs of length $< a$ are there at most?*

$$\sum_{k=0}^{a-1} 2^k = 2^a - 1$$

Then there are at most $2^a - 1$ possible strings with Kolmogorov complexity $< a$.

Theorem 5 *There exists a string x such that*

$$K(x) \geq |x|$$

Proof 6 *Let $|x| = a$, there are 2^a strings of length a. At most $2^a - 1$ of these have Kolmogorov complexity $\leq a - 1$. Then there exists at least one string of length a having Kolmogorov complexity $\geq a$*

3.1.7 Kolmogorov Encoding

Consider the Kolmogorov code defined as follows

$$\varphi_K : A* \longrightarrow \{m | m \text{Turing Machine}\}$$

$\varphi_K(x) = bin(m)$ such that m is the shortest machine such that

$$m(\varepsilon) = x$$

$\varphi_K(x)$ is a U.D. code and a universal machine U decodes φ_K

$$|\varphi_K(x)| = K(x) = K_U(x)$$

Focusing on the language of the strings in SA^n, the average length of φ_K over A^n is

$$EL_n(\varphi_K) = \sum_{x \in A^n} p(x) \cdot K(x) = E_n(K)$$

$E_n(K)$ is the average of the Kolmogorov complexity over the strings of length n. From Shannon Theorem we get

$$EL_n(\varphi_K) \geq \mathcal{H}(P^n) = n \cdot \mathcal{H}_D(P)$$

which leads us to conlcude

$$E_n(K) \geq n \cdot \mathcal{H}(P)$$

$$\frac{E(K)}{n} \geq \mathcal{H}_3(P)$$

To set a lower bound we reason as follows: any U.D. code φ can be seen as a set of machines for producing strings. The couple {Decoder for φ ; $\varphi(x)$ } is an algorithm that produces x as output.

$$\forall x \quad K(x) \leq |Decoder(\varphi)| + |\varphi(x)|$$

$$
\begin{aligned}
E_n(K(x)) \leq E(|Decoder\varphi| + |\varphi(x)|) E_n(K(x)) \quad &\leq |Decoder\varphi| + \sum_{x \in A^n} p(x)|\varphi(x)| \\
&\leq |Decoder\varphi| + EL_n(\varphi)
\end{aligned}
$$

If applied to φ as Shannon Code over strings of length n:

$$
\begin{gathered}
E_n(K(x)) \leq |Decoder\varphi| + \mathcal{H}(P^n) + 1 \\
E_n(K(x)) \leq K(P) + n\mathcal{H}(P) + 1 \\
\frac{E_n(K(x))}{n} \leq \mathcal{H}(P) + \frac{K(P)+1}{n} \\
\mathcal{H}(P) \leq \frac{E_n(K(x))}{n} \leq \mathcal{H}(P) + \frac{K(P)+1}{n}
\end{gathered}
$$

4 Computational Complexity

Determining the complexity of a problem means giving bounds on the complexity of any possible algorithm for solving a problem. We will focus on decision problems which is a class of problems for which the goal is to say, given an instance of the problem, if it has an answer or not. Other classes of problems that we will not explore are: Functional problems and optimization problems.

Decision Problems :

Functional Problems :

Optimization Problems :

Having a fast (polynomial) solution for an optimization problems implies that i have a fast one also for the Functional version and the decision version of the problem. The same goes inversely with the decision being hard, complex (non polynomial).

4.0.1 Computation Model - Turing Machines

The model of computation we are going to use to talk about computational complexity are Turing Machines which we are going to define here for ease of use.

Definition 1 (Turing Machine) *A Turing Machine is a quadruple $M = (K, \Sigma, \delta, s)$. Here K is a finite set of states; $s \in K$ is the initial state. Σ is a finite set of symbols, we say Σ is the alphabet of M. We assume that K and Σ are disjoint sets. Σ always contains the special symbols $\sqcup, \triangleright$: the blank and the first symbol o starting symbol. Finally δ is a transition function, which maps $K \times \Sigma$ to $(K \cup h, \text{"yes"}, \text{"no"}) \times \Sigma \times \leftarrow, \rightarrow, --$. We assume that h (the halting state), "yes" (the acceptin state), "no" (the rejecting state) and the cursor directions $\leftarrow, \rightarrow, --$, for "stay" are not in $K \cup \Sigma$. The function δ is the "program" of the machine. It specifies, for each combination of current state $q \in K$ and current symbol $\sigma \in \Sigma$, a triple $\delta(q, \sigma) = (p, \rho, D)$*

Definition 2 (Configuration of a Turing Machine) *We can define the operation of a Turing machine formally using the notion of a configuration. Intuitively, a configuration contains a complete description of the current state of the computation. Formally a configuration of M is a triple (q,u,w), where $q \in K$ is a state, and w, u are strings in Σ^*. u is the string to the left of the cursor, including the symbol scanned by the cursor, and u is the string to the right of the cursor, possibly empty. Finally q is the current state.*

4.0.2 Time Complexity

This is a decision problem. There's a problem P, a class of models of computation M. The goal is to find in M the fastest machine m that solves P. By fastest we mean that executes least instructions. We need to fix a notion of dimension of the input which is usually done in the definition of the model of computation.

Definition 3 (Time complexity for M TM, on input x) *We say that M, Turing Machine, on input x has time complexity t if*

$$(s, \triangleright, x) \underbrace{\longrightarrow}_{t\ steps} (H, w, u), \quad H \in h, "yes", "no"$$

Definition 4 (Time complexity for M TM) *M has time complexity* $f : \mathbb{N} \longrightarrow \mathbb{N}$ *if:*

$$\forall x \in \Sigma^* \quad (s, \triangleright, x) \underbrace{\longrightarrow}_{t\ steps} (H, w, u)$$

with $t \leq f(|x|)$ *This definition is a worst case time complexity.*

4.0.3 URM

We already know about turing machines but now we'll introduce Unlimited Registry Machines (URM):
Every URM has an infinite set of registers $(R_0, R_1, \dots, R_n)$, in any of these registers there can be an arbitrarily long natural number.

- $S(i) \rightarrow R_i = R_i + 1$.
- $Z(i) \rightarrow R_i = 0$.
- $T(i, j) \rightarrow R_j = R_i$.
- $J(i, j, k) \rightarrow if\ R_i = R_j$ jump to instruction k

Example, P:
Compute $x + 1$. For the turing machine this means receiving the binary digits of x and I want on output the result of $x + 1$. The complexity is linear with the number of digits.
With the URM the complexity is 1, I just need $S(0)$. The problem is that we are hiding the size of the input so this model is not reasonable.
Let's make an addition to the URM model and add the instructions $P(i) \rightarrow R_i = R_i * R_i$ and we execute these instructions: $T(0, 1), J(1, 2, 6), P(0), S(2), J(3, 3, 2)$.
The output goes: $x, x^2, x^4, x^8, \dots x^{2^x}$ with complexity $\Theta(n)$.
With a turing machine only writing the digits of the output requires at least $\Theta(2^x log(x)) \rightsquigarrow \Omega(2^x)$.

Definition 5 (Uniform complexity Cost) *Each instruction of the models has complexity* $\Theta(1)$*. This is not reasonable when we are using models include operations that make the involved integers grow too fast. An example of this behaviour is the URM with the product operation.*

Definition 6 (Logarithmic Complexity Cost) *Each instruction of the models has complexity that depends on the number* k *of digits it manipulates.*

It is clar that it's fundamental to analyze the complexity of all the instructions in your computational model. Let's do it for the URM model:

- $S(i) \to R_i = R_i + 1 \quad \Theta(log(R_i))$
- $Z(i) \to R_i = 0 \quad \Theta(1)$
- $T(i, j) \to R_j = R_i \quad \Theta(log(r_i))$
- $J(i, j, k) \to if\ R_i = R_j$ jump to instruction $k \quad \Theta(min(log(R_i), log(R_j)))$
- $P(i) \to R_i = R_i * R_i \Theta((log(R_i)^2))$

To decide when to use a Logarithmic criteria for computing the complexity cost I have to look for operations in the algorithm that in a polynomial number of steps makes the input grow exponentially, moreover these are used a number of times that depends on the size of the input.
If we now analyze the cost of the models with the Logarithmic complexity cost we get that the URM and the Turing machine are related.

Thesis 1 (Computational Church Turing Thesis) *All reasonable models of computation are polynomially related.*

$$M_1 : \Theta(f(n)) \rightsquigarrow M_2 : \Theta(p(f(n))), \quad p\ polynomial$$

In this case reasonable means that we have to use the logarithmic criteria.

We define as $P = \{L | L$can be decided in polynomial time on a deterministic Turing Machine$\}$.
P is invariant with respect to the model of compuation (consequence of the extended Church tesis).

4.0.4 Review of k-tape TM

Later we'll now demonstrate that the loss in time complexity that we experience by moving from one reasonable model of computation to another is at most quadratic.
To show this result we reintroduce now some concepts about a k tape Turing Machine or k-TM. A configuration for a k-TM takes the following form

$$(q, w_1, u_1, w_2, u_2, \ldots, w_k, u_k), \quad w_i, u_i \in \Sigma^*$$

Every w_i is the string left of the cursor on k-th tape.
The initial configuration for a k-TM on input x is:

$$(s, \triangleright, x, \triangleright, \varepsilon, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon)$$

A language $L \subseteq \Sigma^*$ is decided by a k-TM M if

$$\forall x \in \Sigma^* \begin{cases} M(x) = "yes" & if\ x \in L \\ M(x) = "no" & if\ x \notin L \end{cases}$$

A language $L \subseteq \Sigma^*$ is accepted by a k-TM M if

$$\forall x \in \Sigma^* \begin{cases} M(x) = "yes" & if\ x \in L \\ M(x) \uparrow & if\ x \notin L \end{cases}$$

The difference is that a machine that accepts a language diverges if the input is not in the language. In fact, for a machine that does not terminate on input x we write$M(x) \uparrow$ and we say it diverges while the notation for a terminating TM is $M(x) \downarrow$.
The notion of computation: a computation step for a machine M is a binary relationship between two configuration:

$$(q, u_1, w_1, \ldots, u_k, w_k) \longrightarrow (q^{'}, u_1^{'}, w_1^{'}, \ldots, u_k^{'}, w_k^{'})$$

(For reference Papadimitriu section 2.1)

4.1 Time Complexity

Definition 7 (Time complexity) *Given M a k-TM and x the input for M, we say that M on input x takes time t if*

$$(s, \triangleright, x, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon) \longrightarrow (H, \ldots)$$

with $H \in yes, no, halt$. We say that M operates in time at most $f(n)$ if:

$$\forall x with |x| = m \quad M on x takes time at most f(|x|)$$

. Here $(H, \ldots)$ is short for final configuration.

Definition 8 (Time complexity classes) *Given a language $L \subseteq \Sigma^*$, L is decidable in $TIME(f(n))$ if and only if $\exists k - TM M$ that decides L and operates in time at most $f(n)$.*

$$TIME(f(n)) = \{L \mid L \subseteq \Sigma^* \exists M k - TM\ s.t.\ M\ decides\ L\ in\ time\ f(n)\}$$

Example: L is the language of all strings that are palindromes. $\Sigma = \{0, 1\} \cup \{\sqcup, \triangleright\}$ we get $\Theta(n^2)$

4.1.1 Polynomial relationship between models of computation

We can first start looking at this relationship by showing the time complexity for the problem of deciding the language of palindromes on a 1-TM and a k-TM. Here we summarize the program ideas:

1-TM : We suppose that the tape is originally $\triangleright, x_1, \ldots, x_n, \sqcup$. The machine starts reading the first character x_1, stores the information about the digit in its state, replaces x_1 with $\triangleright$ then moves right until it finds $\sqcup$. At this point it goes back one step and confronts the character from the state and the one under the cursror. If they match it goes back all the way to the new $\triangleright$ and starts over, otherwise it rejects.

k-TM : Starting from the first character the machines reads the value and copies it on a different tape, all the way to x_n. After it copied all the input it moves back to the start on either of the tape and starts moving one cursor forward and one backwards meanwhile checking for matching character at every step.

Theorem 6 (Theorem 2.1 Papadimitriu) *:*
Given M a k-TM that operates in time f(n), then there exists a 1-TM M' operates in time at most $\Theta(f(n)^2)$ such that $\forall x \quad M(x) = M'(x)$. Fundamental hypothesis is that $f(n) \geq n$

Proof 7 *For the sake of brevity we'll only give the idea for the proof. Basically M' has to mimic the k tapes of M with it's only tape. To do that we specify an alphabet that is $\Sigma \cup \underline{\Sigma} \cup \{\triangleright', \triangleleft\}$. We will use the underlined characters to store the information of where is the cursor on the k-th tape of the machine M and the special starting symbol will be use to delimit the start of every k-th string, meanwhile the $\triangleleft$ delimits the end of each k-th "tape".*
To perform any steps of M, M' will have to scan the entire tape once to store in its state the information about every symbol under the k-th cursor and once more to perform the necessary modifications, needing in total 4 traversals of the entire tape. Particular attention must be given to the case in which the k-th cursor is on the last symbol of its sub-tape and wants to move to the right. To allow for such a move we must shift the entire string starting by marking the tape end symbol with an underbar $\underline{\triangleleft}$, then going all the way to the end of the tape of M' and shifting every character one position. We can now move back to $\underline{\triangleleft}$, move it to the right as $\triangleleft$ and placing $\sqcup$ in its previous position.

Theorem 7 (Speed-up Theorem) *If $L \in TIME(f(n))$, then*

$$\forall \epsilon > 0, \quad \exists L \in TIME(\epsilon \cdot f(n) + n + 2)$$

The multiplicative constant in front of the higher degree term is dependant on the model of computation.

Proof 8 *Idea: M' will have to process many digits in a signle "macro" step to reduce the mulitplicative constants. Hypothesis:*

$L \in TIME(f(n)) \to \exists Mk - TM$ decides L in time $f(n)$ Demonstration:

$$\exists M'\ 2 - TM \quad \textit{decides } L \textit{ in time } f'(n) = \varepsilon \cdot f(n) + n + 2, \quad \forall \varepsilon > 0$$

M' has to simulate m steps of M with a constant number of steps (around 6 steps on M' constitute a macro-step of M') (m will depend on $\varepsilon \rightsquigarrow \frac{7}{m}$).
M will make $f(n)$ steps to complete the computation while the steps of M' will be $6 \cdot \frac{f(n)}{m}$
M in m steps can at most read and change m symbols on the tape. So if M' prime has to fit into a single step the m steps of the M machine, its alphabet will have to be Σ^m.This means that we will encode m symbols of the alphabet of M into a single symbol of the alphabet of M'
Each slot of the tape of M' contains an element of the alphabet Σ^m.
A key point is to understande that M with m steps can at most change slots of its tape that were grouped into 2 different slots of the M' machine.
At the beginning of the computation M' reads the input x and encodes it into tuples of Σ^m. In order to simulate the m steps of M the machine M' reads the current tuple, the one on the left and the one on the right and it stores that information on the state. Then it changes at most 2 of the three tuples.
The multiplicative constant 6 comes from the number of moves M' has to make (these can be optimized):

- *Read the information on the starting tuple and store it in the state and move to the left*
- *Read the information and store it in the state and move back to the starting tuple*
- *Move to the right tuple*
- *Read the information and store it in the state, move back to the starting tuple*
- *Change the symbol on the tape and move to either the right or left tuple*
- *Change the symbol on the tape*

The n additive term is because M' has to read the string x of M and encode it in blocks of m, and it has to read the start symbol and blank space of the tape of M which are the 2 steps. Finally M' has to move back to its starting position so there are an additional $\frac{n}{m}$ steps.
$TIME(f(n))$ only makes sense with $f(n) \geq n$

4.2 Space Complexity

Definition 9 (Space Complexity) *Suppose that, for a k-string Turing Machine M and input x, computation of M*

$$(s, \triangleright, x, \ldots, \triangleright, \varepsilon) \xrightarrow{M^*} (H, w_q, u_1, \ldots, w_k, u_k)$$

where $H \in \{h, "yes", "no"\}$ is a halting state. Then the space required by M on input x is $\sum_{i=1}^{k} |w_i| + |u_i|$. If, however, M is a machine with input output (I/O TM), then the space required by M on input x is $\sum_{i=2}^{k-1} |w_i| + |u_i|$.
Suppose now that f is a function from $\mathbb{N}$ to $\mathbb{N}$. We say that Turing Machine M operates within space bound $f(n)$ if, for any input x, M requires at most $f(|x|)$. Finally, let L be a language. We say that L is in the space complexity class SPACE($f(n)$) if there is a Turing Machine with input output that decides L and operates within space bound $f(n)$.

I/O turing machines are linearly related to a generic k-string Turing Machines, so we'll use I/O machines to talk and determine space complexity and generic k-string Turing Machines for time complexity.

Theorem 8 (Space Speed Up Theorem) $L \in SPACE(f(n)) : \forall \varepsilon > 0 \; L \in SPACE(\varepsilon \cdot f(n) + 2)$

RAM computational model

A Random Access Machine, or RAM, is a computing device which, like the Turing Machine, consists of a program acting on a data structure. A RAM's data structure is an array of registers, each capable of containing an arbitrarily large integer, possibly negative. These registers are divided into working registers named r_i and input registers i_j.
Formally a RAM program $P = (\pi_1, \ldots, \pi_n)$ is a finite sequence of instructions, where each instruction π_i is one of the following.

READ J : $r_0 = i_j$

READ ↑j $= r_0 = i_{r_j}$

STORE J $= r_j = r_0$

STORE ↑ j $r_{r_j} = r_0$

LOAD J $r_0 = r_j$

LOAD ↑ j $r_0 = r_{r_j}$

LOAD =j $r_0 = j$

ADD $r_0 = r_0 + r_j$

ADD ↑ j $r_0 = r_0 + r_{r_j}$

ADD =j $r_0 = r_0 + j$

SUB $r_0 = r_0 - r_j$

SUB ↑ j $r_0 = r_0 - r_{r_j}$

SUB =j $r_0 = r_0 - j$

HALF $r_0 = \lfloor \frac{r_0}{2} \rfloor$

JUMP j $k = j$, (k is the program counter)

JPOS j if $r_0 > 0$then $k = j$

JNEG j if $r_0 < 0$ then $k = j$

JZERO j if $r_0 = 0$ then $k = j$

HALT $k = 0$

Definition 10 (RAM configuration) *A configuration of a RAM is a pair*

$$C = (k, R)$$

where k is the program counter and tells the instruction to be executed and R is the set of working and index regsiters with their content

$$R = \{(r_{j1}, j_j), \ldots, (r_{jk}, j_k)\}$$

Definition 11 (Computation step) *Let us fix a RAM program P and input $I = (i_1, \ldots, i_n)$. Suppose that $C = (k, R)$ and $C' = (k', R')$ are configurations. We say that (k, R) yields in one step (k', R') , written $(k, R) \xrightarrow{P,I} (k', R')$, if the following holds: k' is the new value of k after the execution of π_k the k-th instruction of P. R', on the other hand, is the same as R with the difference that registers involved in the π_k instruction have to altered accordingly.*
For instance the program P is $P = (\pi_1, \ldots, \pi_n)$ and the instruction $\pi_k = ADDj$ and $R = (r_{j1}, j_1), \ldots, (r_{jh}, j_h$ from the configuration

$$(k, R) \xrightarrow{P,I} (k + 1, (R \setminus \{(r_0, r_0)\} \cup \{(r_0, r_0 + r_j)\}))$$

This completes the definition of the relation $\xrightarrow{P,I}$. We can now define $\xrightarrow{P,I^k}$ (yields in k steps) and $\xrightarrow{P,I^+}$ (yields).

Definition 12 (Function computed by RAM) *Let P be a RAM program, let D be a set of finite sequences of integers, and let ϕ be a function from D to the integers. We say that P computes ϕ if, and for any $I \in D, (1, \emptyset) \xrightarrow{P,I^*} (0, R)$, where $(0, \phi(I)) \in R$*

The RAM program P on input I takes time t if $(1, \emptyset) \xrightarrow{P,I^t} (0, R)$. We then say that P operates in time $f(n)$ if $\forall I : l(I) = n$, P on I takes time $f(n)$ where $l(I) = \sum_{j=1}^{h} l(i_j)$ where $l(i_j)$ is the length of the binary representation of that integer i_j.

Simulation of a Turing Machine with RAM

Since the RAM only operates on integer we have to encode the alphabet of the TM into integers.

$$D_\Sigma = \{(i_i, \ldots, i_n, 0) | n \geq 0 \forall 1 \leq j \leq n \quad 1 \leq i_j \leq l, \}$$

M decides $L \subseteq \Sigma^*$, while a RAM P simulates the Turing machine M (that decides L) if P computes the function

$$\phi_L : D_\Sigma \longrightarrow \mathbb{N}$$

where

$$\phi_L : (i_1, \ldots, i_n, 0) = \begin{cases} 0 & \sigma_1, \ldots, \sigma_n \notin L \\ 1 & \sigma_1, \ldots, \sigma_n \in L \end{cases}$$