Languages and Algorithms for Artificial Intelligence

Third Module

Exercise Book

Ugo Dal Lago Francesco Gavazzo

1 Mathematical Preliminaries

Exercise 1.1. Prove that for all $n \geq 0$, the following hold:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

Exercise 1.2. For a language $\mathcal{L} \subseteq \{0,1\}^*$, prove that the following are equivalent:

1. $\varepsilon \in \mathcal{L}$,
2. for all $n \geq 0$, $\varepsilon \in \mathcal{L}^n$.

Exercise 1.3. What is the cardinality of the set S^n, where S is any finite alphabet? Prove your claim.

Exercise 1.4. What is the cardinality of the subset of $\{0,1\}^{2n}$ consisting of all and only the *palindrome* words? Prove your claim.

Exercise 1.5. What is the cardinality of the subset of $\{0,1\}^n$ consisting of all and only the words which have even *parity* (and the parity of a binary string is the number of occurrences of the symbol 1 inside it)? Prove your claim.

Exercise 1.6. Relate the following pair of functions (f_i, g_i) by way of $O(\cdot)$, $\Omega(\cdot)$ or $\Theta(\cdot)$ notation:

$$\begin{aligned}
f_1(n) &= n^2 & g_1(n) &= 4n^1 + 100\log(n) \\
f_2(n) &= n\log(n) & g_2(n) &= 10n\log(\log(n)) \\
f_3(n) &= 2^n n^2 & g_3(n) &= 3^n \\
f_4(n) &= 100n & g_4(n) &= \frac{1}{100} n\log(n) \\
f_5(n) &= \log^3(n) & g_5(n) &= n^{\frac{2}{3}} \\
f_6(n) &= 10^{-3}n^3 & g_6(n) &= 10^4 n^3 + 10^5 n^2 \log^3(n) \\
f_7(n) &= 2^{3^n} & g_7(n) &= 2^n \times 3^n \\
f_8(n) &= n\log(n^3) & g_8(n) &= n\log n \\
f_9(n) &= \log(\log(n)) & g_9(n) &= (\log(n))^2 \\
f_{10}(n) &= n + \log(2^n) & g_{10}(n) &= n\log(n)
\end{aligned}$$

Exercise 1.7. Define appropriate encodings of the following countable sets into the set of $\{0,1\}^*$ binary strings:

- The set $\mathbb{Q}$ of all rational numbers.

- The disjoint union $\mathbb{N} \uplus \mathbb{Z}$ of the set $\mathbb{N}$ of the natural numbers and of the set $\mathbb{Z}$ of the integer numbers.
- The class of all finite, directed graphs, namely pairs of the form (V, E) where V is a finite set, and E is a subset of $V \times V$.

2 The Computational Model

Exercise 2.1. Define an efficient 1-tape Turing Machine computing the function $inverse : \{0,1\}^* \to \{0,1\}^*$ such that $inverse(x)$ is the binary string obtained by flipping all bits in x, e.g. $inverse(01011)$ is 10100. Give the Turing Machine explicitly as a triple in the form (Γ, Q, δ).

Exercise 2.2. Define an efficient 2-tape Turing Machine accepting only words $w \in \{0,1\}^*$ having the same number of 0's and 1's. For example, 011100 is accepted and 101 is rejected. Give the Turing Machine explicitly as a triple in the form (Γ, Q, δ).

Exercise 2.3. Define an efficient 1-tape Turing Machine computing the successor function on the natural numbers, when natural numbers are encoded in in pure binary. In order to make your task simpler, you can safely suppose that the binary string encoding a natural number has least signifcant bits on the left and most significant bits on the right, e.g. 12 is encoded as 0011 rather than as 1100. Give the Turing Machine explicitly as a triple in the form (Γ, Q, δ).

Exercise 2.4. Do Exercise 2.3, but assume, now, that natural numbers are encoded as usual, e.g., 12 is encoded as 1100.

Exercise 2.5. A pair (x, y) of binary strings *of equal length* can be easily encoded into a single binary string in many ways. Pick one, and write $\llcorner(x,y)\lrcorner$ for the encoding of the pair. Define an efficient 3-tape Turing Machine computing the function $xor : \{0,1\}^* \to \{0,1\}^*$ such that $xor(\llcorner(x,y)\lrcorner) = x \oplus y$, where $\oplus$ is the bitwise exclusive or operator, i.e. $100 \oplus 101$ is 001. Try to give the Turing Machine explicitly as a triple in the form (Γ, Q, δ). What happens if we just allow two tapes rather than three?

Exercise 2.6. Prove that the following language is undecidable

$$\mathcal{L}_E = \{\lfloor M \rfloor \mid M \text{ is a Turing machine and } \varepsilon \in \mathcal{L}(M)\}$$

using two methods:

1. by showing that it is reducible to $\mathcal{L}_{\text{halt}}$.
2. via Rice's theorem.

3 Polynomial Time Computable Problems

Exercise 3.1. Show that the following problem is computable in polynomial time.

> Given a list $A = [a_1, \ldots, a_n]$ of natural numbers and a number $v \in \mathbb{N}$, return an index $i \in \{1, \ldots, n\}$ such that[1] $A[i] = v$, if any, and return -1 otherwise.

Exercise 3.2. Show that the following problem is computable in polynomial time.

> Sort a list $A = [a_1, \ldots, a_n]$ of natural numbers.

Hint. You do not need to be efficient: a naive sorting algorithm works fine for solving this exercise.

Exercise 3.3. Determine whether the following algorithms run in polynomial time, where for strings s_1, s_2 we denote by $s_1 :: s_2$ their concatenation. Motivate your answer.

[1]We denote by $A[i]$ the i-th element of A.

```
Data: A string $s \in \{0,1\}^*$
$p \leftarrow s$;
$\ell \leftarrow |s|$;
$i \leftarrow 1$;
while $i < \ell$ do
    $p = p :: p$
end
return $p$
```

```
Data: A string $s \in \{0,1\}^*$
$p \leftarrow s$;
$\ell \leftarrow |s|$;
$i \leftarrow 1$;
while $i < \ell$ do
    $p = p :: s$
end
return $p$
```

Exercise 3.4. Recall that a *directed* graph is a pair $G = (V, E)$ where V is a set of vertexes and $E \subseteq V \times V$ is the 'edge' relation between vertexes. Notice that we do not require E to be symmetric. We represent graphs using the so-called adjacency matrices. Formally, we regard graphs as pairs (V, A) where $V = \{v_1, \ldots, v_n\}$ is a set of vertexes and A is an $n \times n$-matrix over $\{0, 1\}$. Intuitively, A encodes the edge relation according to the convention that $A_{i,j} = 1$ if and only if there is an edge from v_i to v_j. A universal sink is a vertex v_i such that for all $j, k \leq n$ with $j \neq i$ we have

$$A_{i,k} = 0 \quad A_{j,i} = 1$$

Notice that if a graph has a universal sink, then the latter is unique. Show that determining whether a graph has a universal sink is computable in polynomial time.

4 Between the Feasible and the Unfeasible

Exercise 4.1. Suppose that $\mathcal{L}_1, \mathcal{L}_2 \in \mathbf{NP}$, i.e., that the two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are both in the class **NP**. Prove that $\mathcal{L}_1 \cap \mathcal{L}_2$ is itself in **NP**. What can we say about $\mathcal{L}_1 \cup \mathcal{L}_2$?

Exercise 4.2. The node cover problem *NODE COVER* is the following: given a directed graph $G = (V, E)$ and an integer $b \geq 2$, determine whether there exists a set $C \subseteq V$ such that:

1. $|C| \leq B$,
2. $\forall (v, u) \in E.\ v \in C$ or $u \in C$.

Show that *NODE COVER* is **NP** complete by reducing from *INDSET*.

Exercise 4.3. The set packing problem *SP* is the following: given a list of sets $S_1, \ldots, S_n$ and an integer $k \geq 2$, determine whether there exist k sets $P_1, \ldots, P_k \in \{S_1, \ldots, S_n\}$ such that $P_1, \ldots, P_k$ are pairwise disjoint. That is:

$$\forall i, j \in \{1, \ldots, k\}.\ i \neq j \implies P_i \cap P_j = \emptyset$$

Show that *SP* is **NP** complete by reducing from *INDSET*.

Exercise 4.4. Given two (undirected) graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ we say that:

1. G_1 is a subgraph of G_2 if and only if $V_1 \subseteq V_2$ and $E_1 = E_2 \cap (V_1 \times V_1)$.
2. A function $h : V_1 \to V_2$ is a *homomorphism* from G_1 to G_2 if and only $(v, u) \in E_1$ implies $(h(v), h(u)) \in E_2$.
3. G_1 is isomorphic to G_2 if and only if there exists a bijective homomorphism h from G_1 to G_2.

The subgraph isomorphism problem SI is the following: given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, determine whether G_1 has a subgraph isomorphic to G_2. Show that SI is NP-complete by reducing from $CLIQUE$.
HINT. Observe that a k-clique of a graph G is a connected subgraph of G with k vertexes, and that all connected graphs with k vertexes are pairwise isomorphic. Therefore, to check whether G has a k clique it is enough to build *a* k-connected graph H (which one does not matter), and check whether G has a subgraph isomorphic to H.

Solutions to Selected Exercises

Exercise 1.1.

- **base case:** we show the property holds for $n = 0$: both $\sum_{i=0}^{0} i$ and $\frac{0(0+1)}{2}$ are equal to 0.
- **inductive step:** Assume that the property holds for some $n \geq 0$, i.e. that

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

We proceed to show that it holds for $n + 1$:

$$\begin{aligned}\sum_{i=0}^{n+1} i &= 1 + 2 + \ldots n + (n+1) \sum_{i=0}^{n} i + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \qquad \text{by the induction hypothesis} \\ &= \frac{n(n+1) + 2(n+2)}{2} = \frac{(n+1)(n+2)}{2}\end{aligned}$$

□

Exercise 1.2. For a language $\mathcal{L} \subseteq \{0,1\}^*$, we recall the inductive definition of $\mathcal{L}^n$:

$$\mathcal{L}^0 = \{\varepsilon\} \quad \text{and} \quad \mathcal{L}^n = \mathcal{L} \cdot \mathcal{L}^n = \{w \cdot w' \mid w \in \mathcal{L}, w' \in \mathcal{L}^n\}$$

where $\cdot$ denotes the operation of string concatenation.

- (2) $\Rightarrow$ (1): Taking $n = 1$, we have $\varepsilon \in \mathcal{L}^1 = \mathcal{L}$ and we are done.
- (1) $\Rightarrow$ (2): We prove the implication by induction on n:
  - **base case:** for $n = 0$, $\varepsilon \in \mathcal{L}^0 = \{\varepsilon\}$ and we are done.
  - **inductive step:** Assume that the property holds for some $n \geq 0$, i.e. that $\varepsilon \in \mathcal{L}^n$, we proceed to show that it holds for $n + 1$. Since $\varepsilon \in \mathcal{L}$ by assumption (1) and $\varepsilon \in \mathcal{L}^n$ by induction hypothesis, we have $\varepsilon = \varepsilon \cdot \varepsilon \in \mathcal{L} \cdot \mathcal{L}^n = \mathcal{L}^{n+1}$ as desired.

□

Exercise 1.5. For every n, let E_n and O_n be the subsets of $\{0,1\}^*$ consisting of the strings of length n having even and odd parity, respectively. As an example:

$$E_3 = \{000, 011, 110, 101\};$$
$$O_3 = \{001, 010, 100, 111\}.$$

It seems that half of the strings of $\{0,1\}^*$ are in E_3 and half are in O_3. Is this a general rule. The answer is positive, and indeed, we will now prove that $|E_n| = |O_n| = 2^{n-1}$ for every $n \geq 1$. The first thing we prove is that for every such n, it holds that

$$E_n = \{x \in \{0,1\}^* \mid x = 0 \cdot y \wedge y \in E_{n-1}\} \cup \{x \in \{0,1\}^* \mid x = 1 \cdot y \wedge y \in O_{n-1}\}$$
$$O_n = \{x \in \{0,1\}^* \mid x = 0 \cdot y \wedge y \in O_{n-1}\} \cup \{x \in \{0,1\}^* \mid x = 1 \cdot y \wedge y \in E_{n-1}\}$$

Every string in E_n, if $n \geq 1$ either starts with a 0 or with a 1. In the former case, the rest of the string is itself in E_n, in the latter case, it is of course in O_n. Viceversa, any string in the form $0 \cdot y$ where $y \in E_{n-1}$ and any string in the form $1 \cdot y$, where $y \in O_{n-1}$ are in E_n. Similarly for strings in the form $0 \cdot y$ where $y \in O_{n-1}$ and any string in the form $1 \cdot y$, where $y \in E_{n-1}$ which are in O_n by definition. As a consequence, we can safely conclude that

$$|E_n| = |E_{n-1}| + |O_{n-1}|; \qquad |O_n| = |O_{n-1}| + |E_{n-1}|. \tag{1}$$

The fact that $|E_n| = |O_n| = 2^{n-1}$ for every $n \geq 1$ can thus be proved by induction on n:

- If $n = 1$, then $E_n = \{0\}$ and $O_n = \{1\}$ and the thesis holds.
- Suppose the thesis holds for n, and let us prove that if must hold for $n + 1$:

$$|E_{n+1}| = |E_n| + |O_n| = 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n;$$
$$|O_{n+1}| = |O_n| + |E_n| = 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n.$$

In both cases, we make use of Equation (1), followed by the induction hypotheses. Please notice how we had to prove a *stronger* result rather than the one required by the exercises, namely that *both* E_n and O_n have a given cardinality. □

Exercise 1.6.

- Consider f_4 and g_4 defined by defined as follows:

  $$f_4(n) = 100n \qquad g_4(n) = \frac{1}{100} n \log(n)$$

  Intuitively, we expect g_4 to grow asymptotically strictly faster than f_4. Let us thus prove that f_4 is $O(g_4)$:

  $$f_4(n) \le c \cdot g_4(n) \Leftrightarrow 100n \le \frac{c}{100} n \log(n)$$
  $$\Leftrightarrow 100 \le \frac{c}{100} \log(n) \Leftrightarrow \log(n) \ge \frac{10000}{c}$$

  As a consequence, *for every* $c > 0$, any n such that $\log(n) \ge \frac{10000}{c}$ would make the inequality $f_4(n) \le c \cdot g_4(n)$ true, which thus holds for sufficiently large n. Similarly, one can prove that $g_4 \in \Omega(f_4)$:

  $$g_4(n) \ge c \cdot f_4(n) \Leftrightarrow \frac{1}{100} nlog(n) \ge 100cn \Leftrightarrow \log(n) \ge 10000c$$

  As a consequence, for every c, the inequality $g_4(n) \ge c \cdot f_4(n)$ holds whenever $\log(n) \ge 10000c$, thus for sufficiently large n.

- Consider the functions f_7 and g_7 defined by:

  $$f_7(n) = 2^{3n} \qquad g_7(n) = 2^n \times 3^n$$

  It is useful to first simplify the two expressions before doing the asymptotic analysis, we indeed have for all $n \ge 0$,

  $$f_7(n) = 2^{3n} = 6^n \qquad g_7(n) = 2^n \times 3^n = 6^n$$

  Since $f_7(n) = g_7(n)$ for all $n \ge 0$, we can conclude immediately that $f_7 \in \Theta(g_7)$.

- Consider the functionsf_8 and g_8 defined by

  $$f_8(n) = n \log(n^3) \qquad g_8(n) = n \log n$$

  We first simplify the expression of f_8: $f_8(n) = 3n \log(n)$ and obtain that for all $n > 0$,

  $$\frac{1}{3}(3n \log(n)) = n \log(n) \le 3n \log(n) \Leftrightarrow \frac{1}{3} f_3(n) = g_3(n) \le f_3(n)$$

  so that $g_8 \in \Theta(f_8)$.

□

Exercise 2.1. The alphabet Γ can be defined as $\{\rhd, \square, 0, 1\}$, while the set of states Q is $\{q_{\texttt{init}}, q_s, q_r\}$. The transition function is specified as follows:

$$\begin{aligned}
(q_{\texttt{init}}, \rhd) &\longmapsto (q_s, \rhd, \texttt{S}) \\
(q_s, \rhd) &\longmapsto (q_2, \rhd, \texttt{R}) \\
(q_s, 0) &\longmapsto (q_s, 1, \texttt{R}) \\
(q_s, 1) &\longmapsto (q_s, 0, \texttt{R}) \\
(q_s, \square) &\longmapsto (q_r, \square, \texttt{S}) \\
(q_r, \square) &\longmapsto (q_r, \square, \texttt{L}) \\
(q_r, 0) &\longmapsto (q_r, 0, \texttt{L}) \\
(q_r, 1) &\longmapsto (q_r, 1, \texttt{L}) \\
(q_r, \rhd) &\longmapsto (q_{\texttt{halt}}, \rhd, \texttt{S})
\end{aligned}$$

In all the other cases (e.g. when the state is $q_{\texttt{init}}$ and the symbol is not $\rhd$, the behavior of the machine is not relevant, i.e., δ can be defined arbitrarily defined. □

Exercise 2.2. The general idea is that we write every 1 we encounter in the input tape into the working tape until we reach the end of the word. We then go back and move left on the working tape only when there is a matching 0 in the input tape.

We take $\Gamma = \{\rhd, \square, 0, 1\}$ for the alphabet, $Q = \{q_{\texttt{init}}, q_{\texttt{halt}}, q_r, q_l\}$ for the set of states and the transition function $\delta : Q \times \Gamma^2 \to Q \times \Gamma \times \{L, S, R\}^2$ as follows:

$$\begin{aligned}
\delta : (q_{\texttt{init}}, (\rhd, \rhd)) &\longmapsto (q_r, \rhd, (R, R)) \\
(q_r, (0, \square)) &\longmapsto (q_r, \square, (R, S)) \\
(q_r, (1, \square)) &\longmapsto (q_r, 1, (R, R)) \\
(q_r, (\square, \square)) &\longmapsto (q_l, \square, (L, L)) \\
(q_l, (0, 1)) &\longmapsto (q_l, \square, (L, L)) \\
(q_l, (1, 1)) &\longmapsto (q_l, 1, (L, S)) \\
(q_l, (\rhd, \rhd)) &\longmapsto (q_{\texttt{halt}}, \rhd, (S, S)) \quad \text{same number of 0's and 1's} \\
(q_l, (0, \rhd)) &\longmapsto (q_l, \rhd, (S, S)) \quad \text{more 0's, the TM is stuck} \\
(q_l, (\rhd, 1)) &\longmapsto (q_l, 1, (S, S)) \quad \text{more 1's, the TM is stuck}
\end{aligned}$$

□

Exercise 2.6.

1. We show that $\mathcal{L}_E$ is undecidable by showing that it is reducible to $\mathcal{L}_{\text{halt}}$ which we have seen to be undecidable.

   - We first define a computable function $\Phi : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ satisfying the following equivalence:

     $$\Phi(\alpha, x) = \varepsilon \quad \Leftrightarrow \quad M_\alpha \text{ halts on input } x.$$

     The TM corresponding to Φ can be defined as follows: run M_α on x and then erase the output tape so that the output is the empty string ε.

   - Assume that $\mathcal{L}_E$ is decidable, i.e. there exists a Turing machine M_E such that for an input (α, x), it returns 1 if M_α outputs ε on input x and returns 0 otherwise.

   - We now define a TM M_{halt} which for an input (α, x) returns 1 if $M_E(\Phi(\alpha), x)$ returns 1 and returns 0 otherwise. Therefore, $M_{\text{halt}}(\alpha, x)$ returns 1 if and only if α halts on x.

2. We show that $\mathcal{L}_E$ is undecidable by using Rice's theorem:

- $\mathcal{L}_E$ is not trivial: the "constant" Turing machine which always outputs 1 is not it $\mathcal{L}_E$ and the "constant" Turing machine which always outputs ε is in $\mathcal{L}_E$.
- $\mathcal{L}_E$ is semantic: for two Turing machines M and N with $\lfloor M \rfloor$ in $\mathcal{L}_E$ and for all input x, M and N have the same output, we must have $\lfloor N \rfloor$ in $\mathcal{L}_E$. If $\lfloor M \rfloor$ in $\mathcal{L}_E$, then it ouputs ε for some input x so by hypothesis, N must also output ε for x and therefore $\lfloor N \rfloor$ is in $\mathcal{L}_E$.

Therefore, we can apply Rice's theorem and conclude that $\mathcal{L}_E$ is undecidable.

□

Exercise 3.1. First, we design an algorithm solving the desired task.

```
**Data:** $A = [a_1, \dots, a_n]$, $v$
**Result:** An index $i \in \{1, \dots, n\}$ s.t.
         $A[i] = v$, if any, $-1$, otherwise
$i \leftarrow 1$;
while $i \leq n$ do
    if $A[i] = v$ then
        return $i$
    else
        $i \leftarrow i + 1$
    end
end
return $-1$
```

How to prove that this algorithm works in polynomial time? As seen in class, we do that in four steps.

1. We encode the input as a binary string. Our analysis of the complexity of the algorithm will be given with respect to the length (call it ℓ) of such a string.
2. We prove that the number of instructions of the algorithm is bounded by a polynomial in ℓ.
3. We argue that each instruction can be simulated by a Turing machine in polynomial time.
4. We show that all 'intermediate' data and results of the algorithm are bounded by a polynomial in ℓ.

We begin with step 1. In order to encode $A = [a_1, \dots, a_n]$ as a binary string, we first need to encode its components a_i. For that, we can choose one standard encoding of the natural numbers in $\{0,1\}^*$. Let us write $\llcorner a_i \lrcorner$ for the encoding of a_i in binary. Since using n bits we can encode $2^n - 1$ (natural) numbers, the encoding of a number $a \in \mathbb{N}$ requires $\log a + 1$ bits[2]. Therefore, we have $|\llcorner a_i \lrcorner| = \log a_i + 1$. The next step is to understand how to encode the whole A. For that, we regard a list of elements $[b_1, \dots, b_n]$ as a 'pair of pairs' of the form $(((b_1, b_2), b_3), \dots b_n)$. Recall that given a pair (b_1, b_2) of bitstrings, we can define the string $\llcorner (b_1, b_2) \lrcorner \in \{0,1\}^*$ by first translating (b_1, b_2) as the string $b_1 \# b_2 \in \{0, 1, \#\}$ and then encoding $b_1 \# b_2$ as a string $\llcorner (b_1, b_2) \lrcorner \in \{0,1\}^*$. For the latter point, we simply map 0 to 00, 1 to 11, and # to 01. As a consequence, we see that $|\llcorner (b_1, b_2) \lrcorner| = 2|b_1| + 2|b_2| + 2$. Now, given a list $[b_1, \dots, b_n]$ of bitstring by regarding it as a pair $(((b_1, b_2), b_3), \dots b_n)$, we see that we obtain an encoding $\llcorner [b_1, \dots, b_n] \lrcorner$ of length $\sum_{i=1}^{n} 2|b_i| + 2(n-1)$. Applying these general considerations to A (and recalling that $|\llcorner a_i \lrcorner| = \log a_i + 1$), we obtain:

$$\llcorner A \lrcorner = \llcorner [\llcorner a_1 \lrcorner, \dots, \llcorner a_n \lrcorner] \lrcorner \qquad\qquad |\llcorner A \lrcorner| = \sum_{i=1}^{n} 2(\log a_i + 1) + 2(n-1)$$

[2]Actually, we should take the floor of $\log a$.

Finally, we pair A with v (recall that both A and v are input of our algorithm), so $\llcorner(\llcorner A\lrcorner, \llcorner v\lrcorner)\lrcorner$ gives an encoding of the input of our algorithm in binary notation. Notice that

$$\ell = |\llcorner(\llcorner A\lrcorner, \llcorner v\lrcorner)\lrcorner| = 2(\sum_{i=1}^{n} 2(\log a_i + 1) + 2(n-1)) + 2(\log v + 1) + 2.$$

We now move to step 2. The latter is straightforward. Our algorithm consists of:

- 1 assignment ($i \leftarrow 1$).
- n iteration of:
  - An inequality check ($i \leq n$).
  - A conditional branching performing:
    - An equality check ($A[i] = v$),
    - Either a return instruction or an assignment ($i \leftarrow i + 1$).
- 1 return instruction

Therefore, the number of the instruction is of the form $b + c \cdot n$, for suitable constants b, c, and thus it is bounded by a polynomial in ℓ. In order to prove step 3, we have to argue that all the aforementioned instructions can be simulated by a TM in polynomial time. For instance, an equality check can be simulated as follows. Say we have two values a and b stored in different portions of a tape of a TM. In order to check whether a is equal to b, the machine simply moves back and forth between a and b checking whether they are bitwise equal. This can be done in polynomial time with respect to the length of a and b, provided that the 'distance' between a and b in the tape is itself bounded by a polynomial in the length of a and b. This will be indeed ensured by step 4. Similar arguments can be used to show that all other instructions can be simulated efficiently by a TM.

Finally, in order to prove step 4 we simply observe that the only intermediate value computed by our algorithm is i, which can be at most n (and therefore it is bounded by a polynomial in ℓ). $\square$

Exercise 3.3. In order to prove that the two algorithms run in polynomial time we have to follow the four steps of previous exercise. These are mostly straightforward. In fact, step 1 is trivial, as the input s is already in binary. For step 2 we simply observe that we have ℓ iterations, and that overall the number of instructions is of the form $b + c \cdot \ell$, for suitable constants b, c, and thus polynomial in ℓ. Moreover, it is not hard to see that all the instructions can be easily simulated by a TM. What goes wrong is step 4. In fact, in the first algorithm at each iteration we concatenate p with itself. That means that if before entering into the while-loop $|p| = \ell$, then after one iteration we will have

$$|p| \text{ after iteration } 1 = 2(|p| \text{ at iteration } 0) = 2\ell$$

Similarly, we obtain

$$\begin{aligned} |p| \text{ after iteration } 2 &= 2|p|(\text{ at iteration } 1) = 4\ell \\ |p| \text{ after iteration } 3 &= 2|(p| \text{ at iteration } 2) = 8\ell \\ &\vdots \end{aligned}$$

and thus

$$|p| \text{ after iteration } n = 2(|p| \text{ at iteration } n-1) = 2^n \ell$$

That means that the length of p is exponential in ℓ, and therefore the first algorithm cannot run in polynomial time. Notice that this is not true for the second algorithm, where we have $|p| \approx \ell^2$. $\square$

Exercise 3.4. Let $G = (V, A)$ be a graph with $V = \{v_1, \ldots, v_n\}$. We design an algorithm for determining whether G has a (necessarily unique) universal sink. The key observation is noticing that if v_i is a universal sink, then the i-th row in A contains only 0s, whereas the i-th column contains all 1s (except in the entry $A_{i,i}$). Graphically:

$$\begin{array}{c c c c c c c}
 & & & & i & & \\
 & & & & 1 & & \\
 & & & & 1 & & \\
 & & & & 1 & & \\
i & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & 1 & & \\
 & & & & 1 & &
\end{array}$$

Let us write $US(v)$ for the property "v is a universal sink". Then, we see that for all vertexes $v_j, v_k \in V$ we have that

$$A_{j,k} = 1 \implies \neg US(v_j);$$
$$A_{j,k} = 0 \implies \neg US(v_k).$$

We can thus proceed as follows. Let POS be a list of potential universal sinks. Obviously, we begin assuming $POS = V$. We sequentially pick pairs of vertexes (v_i, v_j) in POS and look at $A_{j,k}$. If $A_{j,k} = 1$, then we know that v_j cannot be universal sink, and thus we remove it from POS. Otherwise, $A_{j,k} = 0$ meaning that v_k cannot be universal sink, and thus we remove it from POS. Proceeding this way, we will end up with POS containing a single vertex v_i. We then check whether v_i is universal sink by checking whether the i-th row of A contains 0s only, and whether the i-th column of A contains all 1s (except for $A_{i,i}$). If we succeed then we v_i is a universal sink. Otherwise, there is no universal sink.

```
Data: $V = [v_1, \ldots, v_n]$, $A$
Result: An index $i \in \{1, \ldots, n\}$ s.t.
        $US(v_i)$, if any, $-1$, otherwise
$POS \leftarrow [1, \ldots, n]$;
// We use only labels of vertexes
$i \leftarrow 1$;
$j \leftarrow 2$;
while $j \leq n$ do
    if $A_{i,j} = 1$ then
        $POS = POS.remove(i)$;
        $i \leftarrow j$;
        $j \leftarrow j + 1$;
    else
        $POS = POS.remove(j)$;
        $j \leftarrow j + 1$;
    end
end
$i = POS.fst$;
// We have $POS = [i]$ for some $i$
// Check row
$j \leftarrow 1$;
while $j \leq n$ do
    if $A_{i,j} = 0$ then
        $j \leftarrow j + 1$;
    else
        return $-1$
    end
end
// Check column
$j \leftarrow 1$;
while $j \leq n$ do
    if $A_{j,i} = 1$ or $j = i$ then
        $j \leftarrow j + 1$;
    else
        return $-1$
    end
end
return $i$
```

Notice that we might have been more efficient in checking columns and rows. However, our implementation is closer to what we would do when programming a TM and makes our analysis easier.

We now show that the above algorithm works in polynomial time. First, the encoding of the input is standard. The input, in fact, consists of a natural number n (the number of vertexes[3]) and of an $n \times n$-matrix of bits. We encode the latter as a list made of n elements each of which consisting of n bits. Therefore, such an encoding has length $2n^2 + n - 1$. Pairing the latter with the encoding of n (which has length $\log n$), we obtain an input of length $2(2n^2 + n - 1) + 2 \log n + 1$.

How many instructions our algorithm has? The first loop consists of $n - 1$ iterations, each of which essentially consists of assignments, checking values of the matrix, and removing elements from a list. After that, we have loops for checking row and columns, which consist of n iteration each (and thus a total of $2n$ iterations). Summing up, we have $3n - 1$ iterations, plus a fixed number of equality checking, assignments (plus arithmetic operations), and basic operations on lists. The algorithm thus has running time $O(n)$, and thus it is polynomial in $2(2n^2 + n - 1) + 2 \log n + 1$.

[3]Recall that we actually work with labels $1, \ldots, n$ of vertexes, rather than with vertexes themselves.

The last two points to prove in order to conclude that our algorithm runs in polynomial time are showing that each instructions can be simulated by a TM in polynomial time, and that all intermediate values/results have length polynomially bounded by the length of the input. The latter point is straightforward, whereas for the former we essentially proceed as in previous exercises. $\square$

Exercise 4.1. By the fact that $\mathcal{L}_1 \in \mathbf{NP}$ and $\mathcal{L}_2 \in \mathbf{NP}$, we know that there are polynomials p_1, p_2 and polytime TMs $\mathcal{M}_1, \mathcal{M}_2$ such that:

$$\mathcal{L}_1 = \{x \in \{0,1\}^* \mid \exists y_1 \in \{0,1\}^{p_1(|x|)}.\mathcal{M}_1(x, y_1) = 1\};$$
$$\mathcal{L}_2 = \{x \in \{0,1\}^* \mid \exists y_2 \in \{0,1\}^{p_2(|x|)}.\mathcal{M}_2(x, y_2) = 1\}.$$

Now, let $\mathcal{P}$ be the TM which, on input (x, y_1, y_2), simulates $\mathcal{M}_1$ on input (x, y_1) and $\mathcal{M}_2$ on input (x, y_2), and returns 1 iff *both* return 1. Clearly, $\mathcal{P}$ works in polynomial time. Now:

$$\begin{aligned}\mathcal{L}_1 \cap \mathcal{L}_2 &= \{x \in \{0,1\}^* \mid (\exists y_1 \in \{0,1\}^{p_1(|x|)}.\mathcal{M}_1(x, y_1) = 1) \wedge (\exists y_2 \in \{0,1\}^{p_2(|x|)}.\mathcal{M}_2(x, y_2) = 1)\};\\ &= \{x \in \{0,1\}^* \mid \exists y_1 \cdot y_2 \in \{0,1\}^{p_1(|x|)+p_2(|x|)}.(\mathcal{M}_1(x, y_1) = 1) \wedge (\mathcal{M}_2(x, y_2) = 1)\};\\ &= \{x \in \{0,1\}^* \mid \exists y_1 \cdot y_2 \in \{0,1\}^{p_1(|x|)+p_2(|x|)}.(\mathcal{P}(x, y_1, y_2) = 1)\}.\end{aligned}$$

In other words, the class $\mathbf{NP}$ is closed by intersections. Closure by unions can proved similarly, but not exactly in the same way. In particular, from $\mathcal{M}_1$ and $\mathcal{M}_2$, one can form another machine $\mathcal{Q}$, different from $\mathcal{P}$, which on input (x, y) (where $y \in \{0,1\}^{\max\{p_1(|x|), p_2(|x|)\}}$), simulates $\mathcal{M}_1(x, y_1)$ and $\mathcal{M}_2(x, y_2)$, (where y_1, y_2 are prefixes of y of lengths $p_1(|x|)$ and $p_2(|x|)$, respectively) and returns 1 iff *either one or the other* return 1. Clearly, $\mathcal{Q}$ works in polynomial time. Now:

$$\begin{aligned}\mathcal{L}_1 \cup \mathcal{L}_2 &= \{x \in \{0,1\}^* \mid (\exists y_1 \in \{0,1\}^{p_1(|x|)}.\mathcal{M}_1(x, y_1) = 1) \vee (\exists y_2 \in \{0,1\}^{p_2(|x|)}.\mathcal{M}_2(x, y_2) = 1)\};\\ &= \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{\max\{p_1(|x|), p_2(|x|)\}}.(\mathcal{M}_1(x, y_1) = 1) \vee (\mathcal{M}_2(x, y_2) = 1)\};\\ &= \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{\max\{p_1(|x|), p_2(|x|)\}}.(\mathcal{Q}(x, y) = 1)\}.\end{aligned}$$

$\square$

Exercise 4.2. We first show that *NODE COVER* belongs to $\mathbf{NP}$. For that, we consider the language $\mathcal{L}_{\mathit{NODE\ COVER}}$ of *NODE COVER*, where for readability we write, for an object A (such as a graph or a set), A in place $\llcorner A \lrcorner$ (the encoding of A in binary):

$$\mathcal{L}_{\mathit{NODE\ COVER}} = \{(G, b) \mid \exists C \subseteq V.\ |C| \leq b \text{ and } (\forall (v, u) \in E.\ v \in C \text{ or } u \in C)\}.$$

Notice that $\mathcal{L}_{\mathit{NODE\ COVER}}$ is already in the form of a ‘certificate-verifier’, as we can take the very set C as a certificate and define a machine M taking G, b, and C as input and outputting 1 if and only if C meets the conditions required by $\mathcal{L}_{\mathit{NODE\ COVER}}$. The machine M works as follows: given[4] G, we check for all nodes v_i, v_j such that $A_{i,j} = 1$ whether either v_i or v_j belong to C. Such a checking can be done by slightly modifying Exercise 3.1. What remains to be done is to show that C has size polynomial in the size (of the encoding) of (G, b), and that M runs in polynomial time. The latter is trivial, as $C \subseteq V$, whereas for the former we mimic the argument of Exercise 3.1. We conclude that $\mathcal{L}_{\mathit{NODE\ COVER}}$ belongs to $\mathbf{NP}$.

Having proved $\mathcal{L}_{\mathit{NODE\ COVER}} \in \mathbf{NP}$, we show that we can reduce *INDSET* to *NODE COVER* in polynomial time. As we know *INDSET* to be NP-complete, we can conclude *NODE COVER* to be NP-complete as well. In order to perform the reduction, we need to come with a machine M working in polynomial time such that

$$(G, k) \in \mathcal{L}_{\mathit{INDSET}} \iff M(G, k) \in \mathcal{L}_{\mathit{NODE\ COVER}}.$$

That is, any graph G has an independent set of size at least k (≥ 2) if and only if $M(G, k)$, which will be something of the form (G', b), belongs to $\mathcal{L}_{\mathit{NODE\ COVER}}$ (meaning that G' has a cover of

[4]Recall that we encode the edge relation E of a graph $G = (V, E)$ as a $|V| \times |V|$-matrix A such that $A_{i,j} = 1$ iff the ith and jth nodes of G are connected.

size at most b). The key observation is to notice that $I \subseteq V$ is an independent set of a graph $G = (V, E)$ if and only if $V \setminus I$ is a cover of G. In fact, if I is an independent set, then for all $v, u \in I$ we have $(v, u) \notin E$. As a consequence, for all $(v, u) \in E$ either v or u does not belong to I, and thus it is in $V \setminus I$, meaning that $V \setminus I$ is a cover of G. Vice versa, if $V \setminus I$ is a cover of G, then all nodes $v, u \in I$ are not connected (i.e. $(v, u) \notin E$), otherwise one of them would belong to $V \setminus I$, as the latter is a cover of G. We conclude that I is an independent set of G. Moreover, as $|V \setminus I| = |V| - |I|$, we see that the above argument gives us the following result:

$$G \text{ has an independent set of size at least } k \iff G \text{ has a cover of size at most } |V| - k.$$

Our reduction is thus obtained using the machine M that takes (G, k) in input and returns $(G, |V| - k)$. Obviously, M runs in polynomial time. □

Exercise 4.3. First, we prove that $SP \in \mathbf{NP}$. Consider the language $\mathcal{L}_{SP}$ of SP:

$$\mathcal{L}_{SP} = \{(S_1, \ldots S_n, k) \mid \exists P_1, \ldots, P_k \in \{S_1, \ldots, S_n\}.\ \forall i, j \in \{1, \ldots, k\}.\ i \neq j \implies P_i \cap P_j = \emptyset\}.$$

As before, $\mathcal{L}_{SP}$ has the form 'certificate-verifier', as we can take $P_1, \ldots, P_k$ as a certificate, and design a machine M that on input $S_1, \ldots S_n, k, \exists P_1, \ldots, P_k$ returns 1 if and only if $P_1, \ldots, P_k$ are pairwise disjoint. The machine M checks for any set $P_i \in \{P_1, \ldots, P_k\}$ and for any element $p \in P_i$, whether p does not belong to $\bigcup_{j \neq i} P_j$. This essentially reduces to check whether an element belongs to a list/set. In order to conclude that $\mathcal{L}_{SP}$ is in $\mathbf{NP}$, we have to show that the size of the certificate $P_1, \ldots, P_k$ is polynomial in the size of $(S_1, \ldots, S_n, k)$ and that M runs in polynomial time (recall that the input of M is $S_1, \ldots S_n, k, \exists P_1, \ldots, P_k$). The former is trivial, as $P_1, \ldots, P_k \in \{S_1, \ldots S_n\}$, whereas for the latter we slightly modify the argument of Exercise 3.1.

Next, we reduce $INDSET$ to SP. Let $G = (V, E)$ be a graph and $k \geq 2$ be an integer. The main idea behind the reduction is to consider sets

$$S_v = \{\{v, u\} \mid (v, u) \in E\}$$

for any node $v \in V$, and to observe that for $v \neq u$ we have:

$$S_v \cap S_u = \emptyset \iff (v, u) \notin E$$

In fact, assume $S_v \cap S_u = \emptyset$ and, for the sake of a contradiction, assume also $(v, u) \in E$. Then we would have both $\{v, u\} \in S_v$ and $\{v, u\} \in S_u$, this way contradicting $S_v \cap S_u = \emptyset$. Therefore, we have $(v, u) \notin E$. For the vice versa, assume $(v, u) \notin E$ and say, for the sake of a contradiction, that there exists $\{w, z\} \in S_v \cap S_u$. In particular, since $\{w, z\} \in S_v$, either $w = v$ or $z = v$. Without loss of generality, we assume $w = v$. Similarly, since $\{w, z\} = \{v, z\} \in S_u$ we must have $u = z$ (recall that $u \neq v$), so that we have $(v, u) \in E$, this way contradicting the hypothesis. As an immediate consequence, we obtain the following result:

$$G = (\{v_1, \ldots, v_n\}, E) \text{ has an independent set of size } k \iff S_{v_1}, \ldots, S_{v_n} \text{ has a } k\text{-packing.}$$

As a consequence, we can reduce $INDSET$ to SP by considering a machine that takes (G, k) as input and returns $(S_{v_1}, \ldots, S_{v_n}, k)$, where G has vertexes $\{v_1, \ldots, v_n\}$. The machine M constructs a set S_{v_i} simply by inspecting the i-th row of the adjacency matrix of G and taking as S_{v_i} all v_js such that $A_{i,j} = 1$. Obviously, this can be done in polynomial time.

□