Calculabilitate si Complexitate

Cursul 1

- **Calculabilitate:** ce pot calcula cu memorie si timp oricat de mare
  - **Complexitate**: ce pot calcula eficient, cu resurse limitate
- **Functia lui Ackermann**
  - Această funcție are o adâncime mare din punctul de vedere al *recursivității*, valoarea ei crește *foarte rapid* si este o **functie recursiva**, *nu recursiv primitiva*

```
ack(0,n) = n+1
ack(m,0) = ack(m-1,1)
ack(m,n) = ack(m-1, ack(m, n-1))
```

- **Definitia formala** a functiei Ackermann:

$$A(m,n) = \begin{cases} n+1 & \mathrm{m} = 0 \\ A(m-1,1) & \mathrm{m} > 0 \text{ si n} = 0 \\ A(m-1,A(m,n-1)) & \mathrm{m} > 0 \text{ si n} > 0 \end{cases}$$

Calculam $A(1,1)$ astfel $A(1,1) = A(0,A(1,0) = A(0,A(0,1)) = A(0,2) = 3$

- **Functia lui Collatz**
  - *Construim un şir de numere naturale in felul urmator:* $x_0 = n$*,* x_{i+1} *se obţine din* x_i *dupa urmatoarea regula: daca* x_i *e par atunci* x_{i+1} *e jumatatea lui* x_i*, altfel e* $3x_i + 1$
  - Conjectura lui Collatz afirma faptul cã pentru orice valoare initiala, şirul va atinge la un moment dat valoarea 1, dupa care repeta la infinit ciclul 4,2,1,4,2,1,...

```
def collatz(n):
  while (n > 1):
    if (n % 2 == 0):
      n = n / 2
    else:
      n = 3 * n + 1
```

- A are acelasi numar de elemente cu $B \Leftrightarrow \exists\ f : A \to B$ bijectiva
- The **Matiyasevich Theorem** states that it is not possible to solve the problem of determining whether a *given polynomial equation* has a *whole number solution*, using a computer program. More specifically, the theorem states that there is no algorithm that can take as input a polynomial equation and determine whether it has a solution in the integers.
  - Se da $p(x_1, x_2, ..., x_n)$ cu coeficienti intregi.
  - Vreau sa decid $\exists (x_1, x_2, ..., x_n) \in Z$ pentru $p(x_1, x_2, ..., x_n) = 0$

Cursul 2

- O **problema de decizie** este definita astfel: **INPUT -** $x \in \Sigma^*$, **OUTPUT -** $true/false$
- O functie $f : \mathbb{N}^k \to \mathbb{N}$, eventual partiala, poate fi considerata calculabila, daca exista o procedura efectiva (un algoritm) care, pornita pentru niste valori initiale $(n_1, n_2, ..., n_k) \in \mathbb{N}^k$ se opreste dupa un numar finit de pasi si produce un rezultat $f(n_1, n_2, ..., n_k) \in \mathbb{N}$

Mai jos sunt definite functiile primitiv recursive. Este vorba despre functii $f : \mathbb{N}^k \to \mathbb{N}$ cu un numar arbitrar de variabile. Principiul acestei definitii este urmatorul: intai sunt mentionate cateva functii simple care fac parte din aceasta clasa, apoi sunt mentionate operatiile cu functii la care este inchisa aceasta clasa sau, mai bine zis, care genereaza aceasta clasa.

Definitie: Functiile primitiv recursive se definesc in modul urmator.

- Toate functiile constante $f : \mathbb{N}^k \to \mathbb{N}$ date de $c(n_1, \ldots, n_k) = c$ sunt primitiv recursive.

- Toate proiectiile $\pi_m^k : \mathbb{N}^k \to \mathbb{N}$ date de $\pi_m^k(n_1, \ldots, n_m, \ldots, n_k) = n_m$ sunt primitiv recursive.

- Functia succesor $s : \mathbb{N} \to \mathbb{N}$ data de $s(n) = n + 1$ este primitiv recursiva.

- Orice compunere de functii primitiv recursive este o functie primitiv recursiva. Mai exact, daca $f : \mathbb{N}^k \to \mathbb{N}$ si functiile $g_1, \ldots, g_k : \mathbb{N}^m \to \mathbb{N}$ sunt primitiv recursive, atunci functia $h : \mathbb{N}^m \to \mathbb{N}$ data de:

$$h(n_1, \ldots, n_m) := f(g_1(n_1, \ldots, n_m), \ldots, g_k(n_1, \ldots, n_m))$$

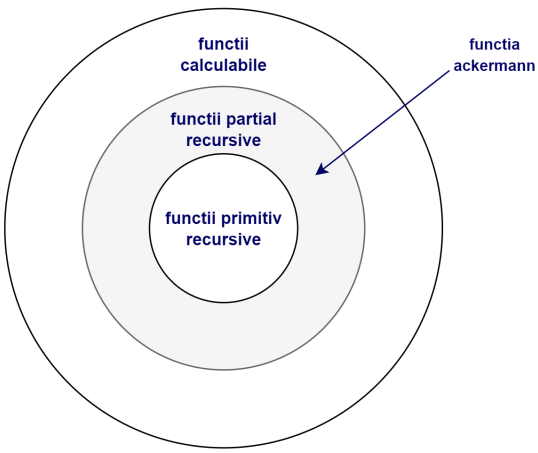
este primitiv recursiva.

- Orice functie obtinuta prin operatia de recursie primitiva din functii primitiv recursive este primitiv recursiva. Mai exact, daca $f : \mathbb{N}^k \to \mathbb{N}$ si $g : \mathbb{N}^{k+2} \to \mathbb{N}$ sunt functii primitiv recursuve, atunci functia $h : \mathbb{N}^{k+1} \to \mathbb{N}$ data de:

$$\begin{aligned} h(0, n_1, \ldots, n_k) &= f(n_1, \ldots, n_k) \\ h(n+1, n_1, \ldots, n_k) &= g(h(n, n_1, \ldots, n_k), n, n_1, \ldots, n_k) \end{aligned}$$

este primitiv recursiva. □

- $f : \mathbb{N} \to \mathbb{N}$ se numeste **primitiv recursiva** daca pot sa o obtin din functiile de baza prin *compunere* si *recursie primitiva*
- O functie se numeste **partial recursiva** daca se poate obtine din functiile de baza prin *compunere*, *recursie primitiva*, *minimizare*
- O functie $f : \mathbb{N} \to \mathbb{N}$ este **partial recursiva** doar daca f este *calculabila de o masina Turing*



- A **primitive recursive function** is a type of mathematical function that is defined using a set of basic operations, such as addition and composition of functions. These functions are considered "**primitive**" because they form the building blocks for constructing more complex functions. They are also considered "**recursive**" because they can be *defined in terms of themselves*.
  - In simple terms, a primitive recursive function is a mathematical function that can be built up from *simple operations*, and that can *call itself as part of its calculation*.

Cursul 3

- O **masina Turing** determinsta cu o banda este un tuplu $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ unde
  - Z este o multime finita de stari
  - Σ este alfabetul de input
  - $\Gamma \supset \Sigma$ este alfabetul de lucru
  - $\delta : Z \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ este functia de tranzitie
  - $z_0 \in Z$ este starea initiala
  - $\square \in \Gamma \backslash \Sigma$ este simbolul blank
  - $E \subset Z$ este multimea de stari finale
- O banda infinita in ambele sensuri contine inputul, care este un cuvant $w \in \Sigma^*$. Un cap de citire si scriere se afla pe prima litera a inputului iar masina se afla in starea initiala z_0

$$\ldots \square \square \square \boxed{a} \, b \, c \, 0 \, 1 \, 1 \, 0 \, \square \square \square \ldots$$

- Daca la un moment dat masina se afla in starea z si citeste litera a, se evalueaza functia de tranzitie $\delta(z, a) = (z', b, x)$. Deci masina va inlocui a cu b, va trece in starea z_0 si va face un pas la dreapta, la stanga sau va ramane pe loc (R, L, N)
- **Church's thesis** states that every effectively *computable functio*n can be computed by a Turing machine. This means that, for any task that can be performed by a computer program, there exists a corresponding **Turing machine** that can perform the same task.
- A **Universal Turing machine** is a theoretical model of a computer that was proposed by the mathematician Alan Turing in 1936. It is considered to be the theoretical foundation of modern computing.
  - In simple terms, a Universal Turing machine is a hypothetical machine that can perform any computation that can be described by a set of rules. It is called "universal" because it can perform any computation, given the appropriate set of rules, without having to be specifically designed for a particular task.
- A **Recursively Enumerable Se**t is a set of strings that can be *generated by a Turing machine*. This means that there is a Turing machine that can write out all the strings in the set, and that can *determine* whether a given string *is in the set.*
  - O ***multime recursiv enumerabila*** poate fi calculata de o Masina Turing. Multimea nevida A este recursiv enumerabila daca si numai daca este **semidecidabila:**

$$f(x) = \begin{cases} 1 & x \in A \\ nedefinit & x \notin A \end{cases}$$

  - O ***multime recursiva*** poate fi calculata de o Masina Turing care se *opreste* ***intotdeauna***. Multimea A este recursiva daca si numai daca este **decidabila:** (?)

$$f(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

Cursul 4

- In computability theory, an **undecidable problem** is a decision problem for which it is proved to be impossible to construct an algorithm that *always* leads to a correct yes-or-no answer
- The **halting problem** asks whether it is possible to *write a program or algorithm* that can take as input a *description of another program* and **determine whether that program will halt its execution** or run forever.
    - The problem was first formalized by Alan Turing in 1936, and he showed that the halting problem is **undecidable**. This means that there is *no general algorithm* that can solve the halting problem for *all possible inputs*.
- Other undecidable problems: *Wang Tiles, Self-Reference Problem*

Cursul 5

- **The Busy Beaver problem** asks to find the *maximum number of steps* that a Turing machine can take before halting, given a *fixed number of states*. The problem can be formalized as follows:
    - Given a Turing machine with n states, what is the ***maximum number of steps*** that the machine can take before halting, when starting with an empty tape?
    - The Busy Beaver problem is an example of a **optimization problem**, where the goal is to find the best possible solution. The problem is also **undecidable**, meaning that there is no general algorithm that can solve the problem for all possible inputs.

Cursul 6

- **Complexitate computationala:**
    - Un **algoritm eficient** are complexitatea $O(n^k)$ unde $k = 2, 3, ...$
    - *Backtracking* are complexitatea $O(2^n)$ deci **nu e eficient**
- Complexitate ***polinomiala*** vs ***exponentiala***
    - Polinomiala: $O(n^k)$ algoritm eficient
    - Exponentiala: solutie *usor de verificat* dar *greu de gasit*

Cursul 7

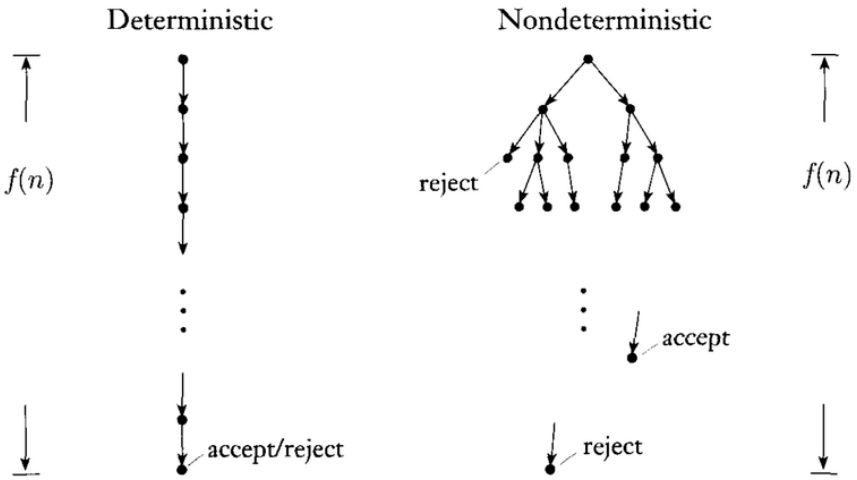
- **The P versus NP problem** is one of the most famous open problems in the theory of computation, and it asks whether two types of computational problems *are equivalent or no*t.
    - P (stands for "**polynomial time solvable**") is a class of computational problems that can be solved in a time proportional to some *polynomial function* of the size of the input. In other words, the running time of a P problem grows *relatively slowly* with the size of the input.
    - NP (stands for "**nondeterministic polynomial time**") is a class of computational problems for which a solution *can be verified in polynomial time,* but **finding the solution** may take

exponential time.

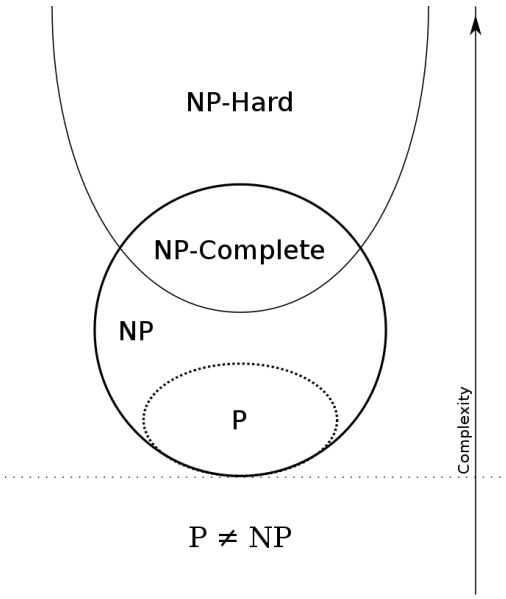
- The P versus NP problem asks whether every problem in NP can be solved in polynomial time (that is, **whether P = NP**), or if some NP problems require exponential time to solve

$$\mathrm{P} \subseteq \mathrm{NP} \subsetneq \text{Primitiv Recursiv} \subsetneq \text{Decidabil} \subsetneq \text{Semidecidabil}.$$

- **Masini Turing** *deterministe* vs *non-deterministe*



- **NP-hard** and **NP-complete** are terms used in computer science to describe the difficulty of solving *computational problems*.
  - **NP-hard** refers to a problem that is *at least as hard* as the hardest problems in NP. In other words, if a problem is NP-hard, it cannot be solved in polynomial time unless P = NP.
  - **NP-complete** is a class of problems that are both **NP-hard and in NP**, meaning that they are the hardest problems in NP. If any one NP-complete problem can be solved in polynomial time, then *all NP problems can be solved in polynomial time*, which would imply P = NP.



- Fie A NP-completa atunci $A \in P \iff P = NP$

- Fie $A \subseteq \Sigma^*$ si $B \subseteq \Gamma^*$ doua multimi. Se spune ca A este polinomial reductibil la B si se scrie $A \leq_p B$ daca exista o functie totala, calculabila in timp polinomial, $f : \Sigma^* \to \Gamma^*$ astfel incat pentru orice $x \in \Sigma^*$ are loc $x \in A \iff f(x) \in B$
- **Lema**: Daca $A \leq_p B$ si $B \in NP$, atunci $A \in NP$.
- **Lema:** Daca $A \leq_p B$ si $B \in P$, atunci $A \in P$.

Cursul 8

- **The Boolean Satisfiability Problem** (SAT) is a *decision problem* that asks whether a given boolean expression can be satisfied, or made true, by assigning values (either true or false) to its variables.
  - Given a **boolean formula** composed of variables and logical operators (e.g. AND, OR, NOT), the *SAT problem* asks whether it is possible to assign values to the variables in such a way that the *entire formula evaluates to true*.
  - The SAT problem is an **NP-complete** problem.
- The **3-SAT** problem is a specific type of the SAT. It is a decision problem that asks whether a given boolean formula with **exactly three literals** per clause can be satisfied, or made true, by assigning value to its variables. The 3-SAT problem is an **NP-complete** problem.
- The **2-SAT** problem is a **P problem**
- **Integer Linear Programming** (ILP) is a type of linear programming optimization problem where some or all of the variables are restricted to integer values. The goal of ILP is to find an optimal solution (i.e., one that maximizes or minimizes a given objective function) subject to a set of linear constraints. The ILP problem an **NP-complete** problem
  - Se pot demonstra relatiile **SAT** $\leq_p$ **3-SAT** si **3-SAT** $\leq_p$ **ILP**

Cursul 9

- **The Independent Set Problem** is a classical problem in *graph theory* and computational complexity. Given a graph, an independent set is a set of vertices such that no two vertices are adjacent, meaning that there is no edge connecting them. The independent set problem asks for the maximum size of an independent set in a given graph.
  - More formally, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the independent set problem is to find a subset of vertices $S \subseteq V$ such that no two vertices in S are adjacent, and $|S|$ is as large as possible.
  - The IS problem is an **NP-complete** problem.
- **The Clique Problem** is a classical problem in graph theory and computational complexity. Given a graph, a clique is a set of vertices such that **every two vertices are adjacent**. The clique problem asks for the *maximum size of a clique* in a given graph.
  - More formally, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the clique problem is to find a subset of vertices $S \subseteq V$ such that every two vertices in S are adjacent, and $|S|$ is as large as possible.

    - The CLIQUE problem is an **NP-complete** problem.
- **The Vertex Cover Problem** is a classical problem in graph theory and computational complexity. Given a graph, a vertex cover is a set of vertices such that every edge in the graph is **incident to at least one vertex** in the set. The vertex cover problem asks for the *minimum* size of a vertex cover in a given graph.
    - More formally, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the vertex cover problem is to find a subset of vertices $S \subseteq V$ such that every edge in E is incident to at least one vertex in S, and $|S|$ is as small as possible.
    - The vertex cover problem is an **NP-complete** problem.
- Se pot demonstra relatiile **3-SAT** $\leq_p$ **IS, IS** $\leq_p$ **CLIQUE** si **IS** $\leq_p$ **VC**

Cursul 10

- The **Davis–Putnam–Logemann–Loveland (DPLL)** algorithm is a complete and efficient algorithm for solving the **Boolean Satisfiability Problem (SAT)**.
    - The DPLL algorithm works by *dividing the problem into smaller* subproblems and using a combination of *backtracking* and *heuristics* to find a solution. The algorithm starts by setting one of the variables to either true or false, and then uses the constraints to propagate the effects of that assignment to other variables.
    - If the current assignment leads to a contradiction, the algorithm backtracks to a previous decision and tries a different assignment. If all variables have been assigned and the constraints are satisfied, the algorithm returns the **solution**. If no solution is found, the algorithm returns that the problem is **unsatisfiable**.
    - Its *worst-case time* complexity is **exponential**, but in practice it performs well.
- **The Conflict-Driven Clause Learning (CDCL)** algorithm is an efficient algorithm for solving the SAT problem. It is an improvement over the DPLL algorithm and is widely used in SAT solvers.
    - The CDCL algorithm works almost the same is DPLL, the **key difference** being the way in which the algorithm handles **contradictions**. When a contradiction is detected in CDCL, the algorithm identifies the cause of the contradiction and learns a *new constraint*, called a **conflict clause**, that *prevents* the same *contradiction* from happening again. This new constraint is then added to the *set of constraints* and used to guide the search.
    - Its *worst-case time* complexity is **exponential**, but in practice it performs well.

Cursul 11

- **Quantified Boolean Formulas** (QBF) is a type of mathematical formula used to represent the *satisfaction* of a *boolean function* in the context of computational complexity theory. In QBF, each variable is quantified, either **universally** or **existentially**. and the truth of the formula depends on the truth values of the quantified variables.
    - *Example:* $\forall x \; \exists y \; (x$ OR $y)$. In this QBF formula, the variable x is universally quantified $(\forall x)$ and the variable y is existentially quantified $(\exists y)$. The formula states that for all values of x, there exists a value of y such that *(x OR y)* is **true**.

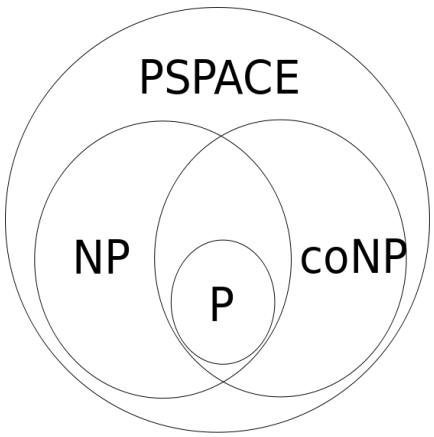
- The problem is **NP-complete**, meaning that it is at least as hard as the hardest problems in NP, and that there is no algorithm that can solve it in polynomial time unless P = NP.

Cursul 12

- **Co-NP** is the *class of decision problems* that are the **complement of problems in NP**. In other words, a problem is in Co-NP if and only if its complement is in NP. A decision problem is a problem where the answer is either "yes" or "no", and **Co-NP problems** are problems where the answer is **"no"** if and only if a corresponding **NP problem has a "yes"** answer.
  - Formally, a problem is in Co-NP if there is a polynomial-time *verifiable proof* that a proposed solution is *not correct*. In other words, for a Co-NP problem, there exists an algorithm that can check in polynomial time if a given candidate solution **is not valid.**
  - Many problems that are not known to be NP-complete are known to be in Co-NP, which means that if $P \neq NP$, then these problems **cannot be solved in polynomial** time either.

Cursul 13

- **PSPACE** is a class of problems in computational complexity theory that are solvable by algorithms that use a polynomial amount of space
  - The *amount of time* used by such an algorithm can be *exponential* in time, but the **amount of memory** used must be **polynomial**
- **NPSPACE** is a class of problems that are solvable by algorithms that use a **polynomial** amount of **nondeterministic space**
  - It includes problems that are solvable using a polynomial amount of memory on a *nondeterministic* **Turing machine.**
  - *NPSPACE* is considered to be a larger class of problems than NP and PSPACE



- The current best evidence suggests that **PSPACE** and **NPSPACE are not equa**l, but this has not been rigorously proven. One reason why PSPACE and NPSPACE might not be equal is that NPSPACE algorithms can use *nondeterminism* to *guess the solution* to a problem, which can potentially make it **easier to solve** some problems than with **deterministic** algorithms.
- **Savitch's theorem** is a result that *relates the amount of memory* used by an algorithm to the *running time* of that algorithm. It is particularly useful for studying problems in the PSPACE class, which consists of problems that can be solved using a polynomial amount of memory on a **deterministic Turing machine**.
  - The theorem implies that if a problem can be solved using a **polynomial amount of memory**, then it can also be solved using a **logarithmic amount of memory**, which is much more efficient.