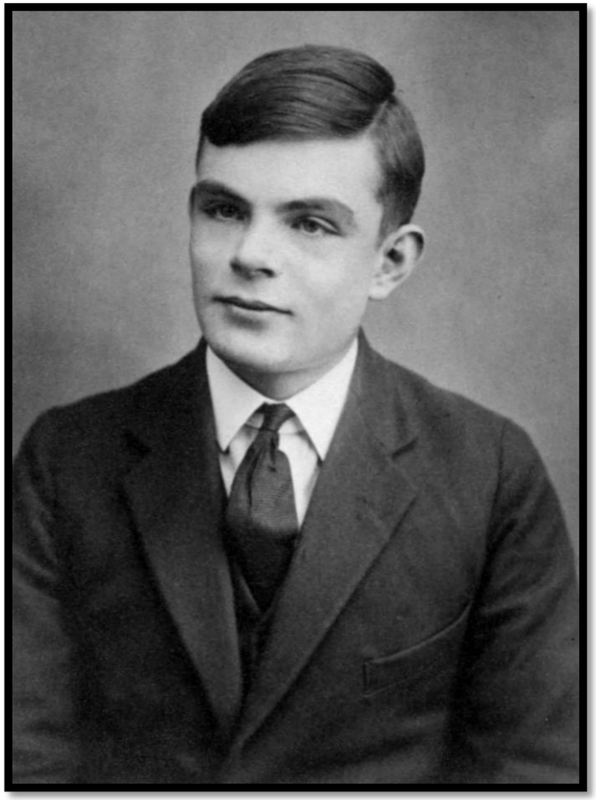
Computability and Complexity

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Can computers solve all problems?



Alan Turing

Model for Computation

Turing Machine

- An *infinite* **tape** divided into cells.
- A **head** that can read or write symbols on the tape, and move the tape left or right *one cell at a time*.
- A **state register** storing current state of the machine, among *finitely* many states.
- A *finite* **table of instructions**:
  - Given current state and current read symbol:
    Either erase or write a symbol;
    Move the head (left, right, or remain stationary);
    Assume the same or a new state.

Can computers solve all problems?

For each problem,
there exist a TM to solve it?

Decision Problem

- Problems that expect a **YES** or **NO** answer.
- An **instance** of a problem conceptually contains two parts:
  - Instance description;
  - The question itself.

- **Example:** Given a graph G, a pair of nodes (u, v), an integer k, is every path between (u, v) of length at least k?
- **Example:** Given a multiset S, is there a way to partition S into two subsets of equal sum?

Optimization vs Decision

- In an **optimization problem**, among all feasible solutions, we find one that *maximizes* (or *minimizes*) a given *objective*.
- **Example:** Given a graph G, a pair of nodes (u, v), what is the length of the shortest path between (u, v)?
- If we have an efficient algorithm for a decision problem, then we can usually solve the corresponding optimization problem efficiently, and vice versa.
- **Recall:** Given a graph G, a pair of nodes (u, v), an integer k, is every path between (u, v) of length at least k?
- **Another example**: chromatic number vs k-colorable.

Can computers solve all problems?

For each decision problem,
there exists a TM to solve it?

Turing Machine

- An *infinite* **tape** divided into cells.
- A **head** that can read or write symbols on the tape, and move the tape left or right *one cell at a time*.
- A **state register** storing current state of the machine, among *finitely* many states.
- A *finite* **table** of instructions.

- Informally, we say a TM solves (decides) a decision problem if for *each* instance of the problem, within *finite* steps, the TM *correctly* outputs "yes" or "no" and then *halts*.

Can computers solve all problems?

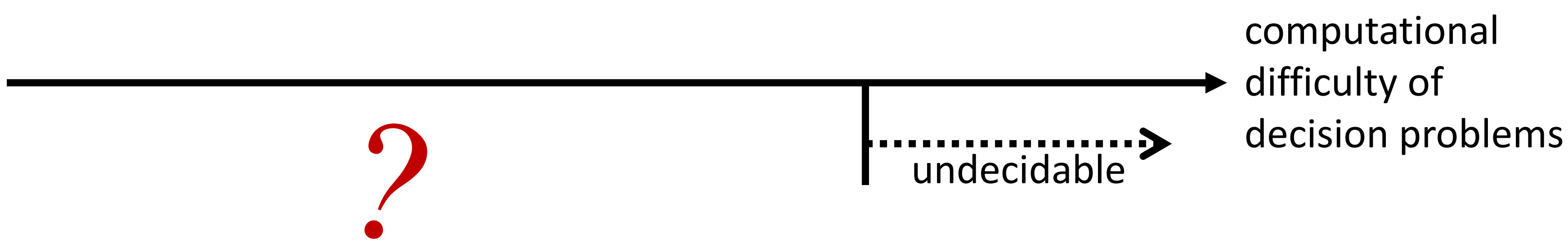
For each decision problem,
there exists a TM to decide it?

NO!

The Halting Problem

Given a computer program s and input x, will $s(x)$ ever halt?

No TM can decide the halting problem!



How *fast* can computers solve a problem?

For a given decision problem,
how *fast* can a TM decide it?

The Class **P**

- Consider a decision problem $\mathcal{P}$, let I be an instance of $\mathcal{P}$.
- Let $|I|$ denote the length of I under, say, binary encoding.
- An algorithm $\mathcal{A}$ for $\mathcal{P}$ is polynomially bounded, if the runtime of $\mathcal{A}$ is $(|I|)^{O(1)}$ for all I.
- **P** is the set of decision problems each of which has a polynomially bounded algorithm.
- **P** is the set of decision problems each of which can be decided by some TM within polynomial time.

- Most (but not all) problems we have studied so far are in **P**.

Some notes on **P**

- **P** contains the set of so-called tractable problems.
- So problems with $\Theta(n^{100})$ time algorithms also tractable?!
- Being in **P** doesn't mean a problem has efficient algorithms.
- Nonetheless:
  - Problems not in **P** are definitely expensive to solve.
  - Problems in **P** have "closure properties" for algorithm composition.
  - The property of being in **P** is independent of computation models.

A note on size of input

- Recall decision problem $\mathcal{P} \in \mathbf{P}$ if there exists an algorithm that can solve $\mathcal{P}$ in $(|I|)^{O(1)}$ time for every instance I of $\mathcal{P}$.

IsPrime(n):

```
for (i=2 to n-1)
  if (n%i == 0)
    return false
return true
```

- This algorithm has poly-n runtime, so $\mathbf{Primes} \in \mathbf{P}$?
- **No!** The size of the input is $O(\log n)$ with binary encoding.
- Indeed $\mathbf{Primes} \in \mathbf{P}$, but proved with a different algorithm…

Su

SubsetSumDP(X,T):

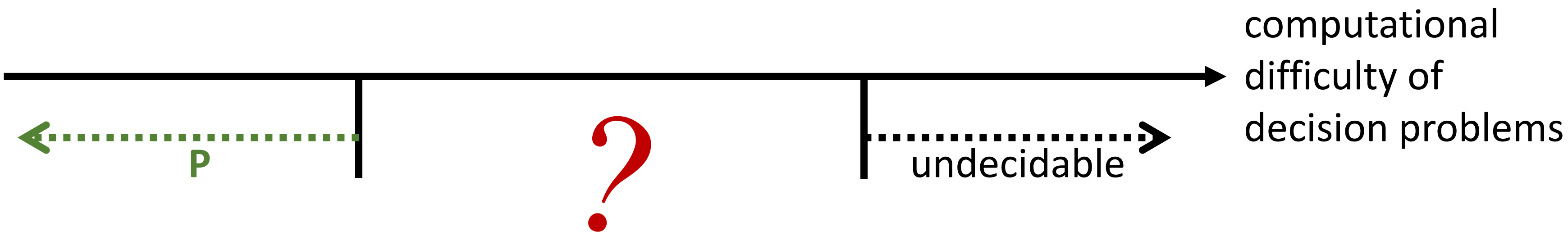
```
ss[n,0] = True
for (t=1 to T)
  ss[n,t] = (X[n]==t)?True:False
for (i=n-1 downto 1)
  ss[i,0] = True
  for (t=1 to X[i]-1)
    ss[i,t] = ss[i+1,t]
  for (t=X[i] to T)
    ss[i,t] = Or( ss[i+1,t], ss[i+1,t-X[i]] )
return ss[1,T]
```

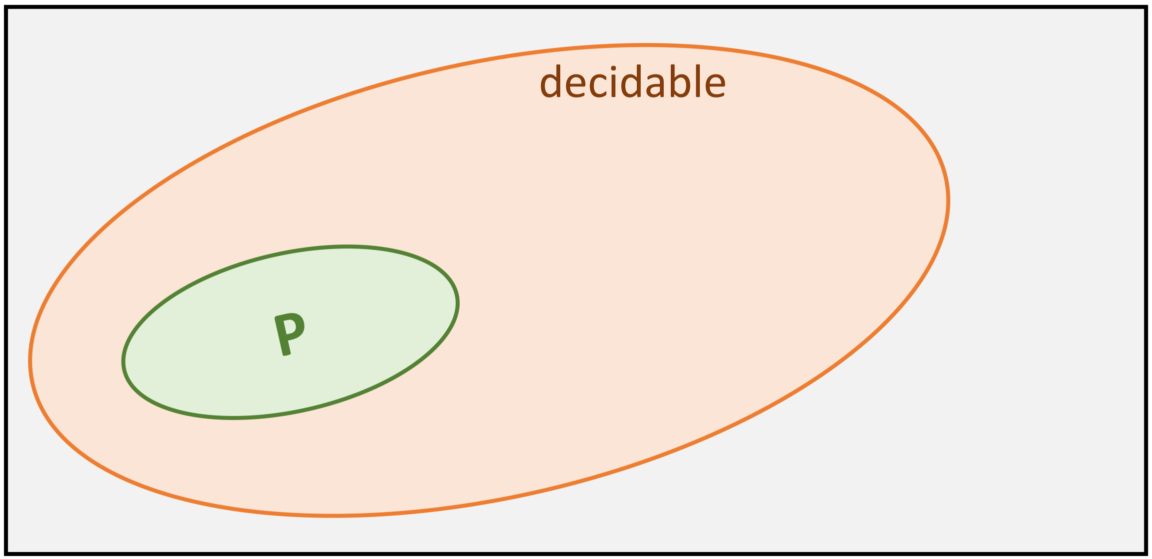
Runtime is $O(nT)$

- **Pro** rs, can ?
- **Ste**
  -
- **Step 2: Recursively define the value of an optimal solution.**
  - Let $ss(i,t) = true$ iff instance "$X[i \cdots n], t$" has a solution.
  - $$ss(i,t) = \begin{cases} true & \text{if } t = 0 \\ ss(i+1,t) & \text{if } t < X[i] \\ false & \text{if } i > n \\ ss(i+1,t) \vee ss(i+1, t - X[i]) & \text{otherwise} \end{cases}$$
- **Step 3: Compute the value of an optimal solution (Bottom-Up).**
  - Build an 2D array $ss[1 \cdots n, 0 \cdots T]$
  - Evaluation order: bottom row to top row; left to right within each row.

Subset Sum

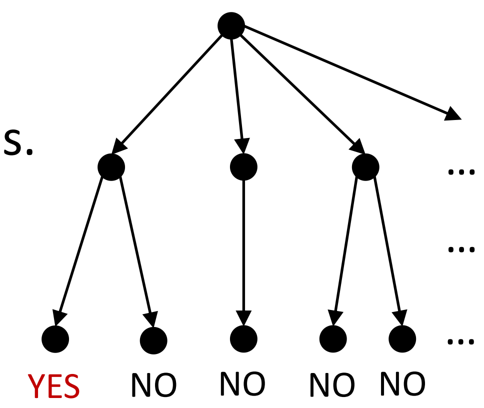
- **Problem:** Given an array $X[1 \cdots n]$ of n positive integers, can we find a subset in X that sums to given integer T?
- **Simple solution:** recursively enumerates all 2^n subsets, leading to an algorithm costing $O(2^n)$ time.
- **Dynamic programming:** costing $O(nT)$ time.
- Both algorithms are ***not*** polynomial time algorithms!


computational
difficulty of
decision problems
P
?
undecidable


decidable
P

Non-deterministic Turing Machine

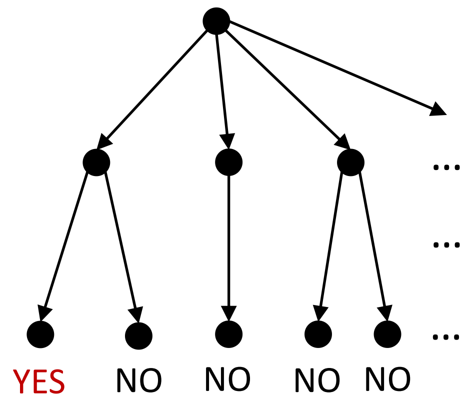
- An *infinite* **tape** divided into cells.
- A **head** that can read or write symbols on the tape, and move the tape left or right *one cell at a time*.
- A **state register** storing current state of the machine, among *finitely* many states.
- A *finite* **table** of instructions:
  - Given current state and current read symbol, choose to execute an action among *many* actions.
  - **E.g.:** if currently in state 3 and read symbol X: write a Y, move left, and switch to state 5, **or** write an X, move right, and stay in state 3.
- An NTM M on input x *returns* "yes" iff *some* execution of $M(x)$ halts with "yes".



The Class **NP**

- An NTM M on input x *returns* "yes" iff *some* execution of $M(x)$ halts in "yes" state.
- Informally, we say an NTM solves (decides) a decision problem $\mathcal{P}$ in time $T(n)$ if for *each* instance I of $\mathcal{P}$, within $T(|I|)$ steps, the NTM *correctly* *returns* "yes" or "no".
- I.e., within $T(|I|)$ steps, ≥ 1 branch "yes", or all branches *not* "yes".

- **NP** is the set of decision problems each of which can be decided by some NTM within polynomial time.
- **NP** means "non-deterministic polynomial time."



The Class **NP**, Take Two

- **Non-deterministic Algorithm:**
- A *free*, *non-deterministic* "guessing" phase.
  (Guess a proof, usually a solution.)
- **E.g.:** Consider the "k-colorable" problem,
  so $x = (G, k)$, and cer is a coloring scheme.
- A deterministic "verification" phase.
  (Verify the correctness of the proof.)
- **E.g.:** Verify cer is a valid $\leq k$-coloring of G.
- An output step.

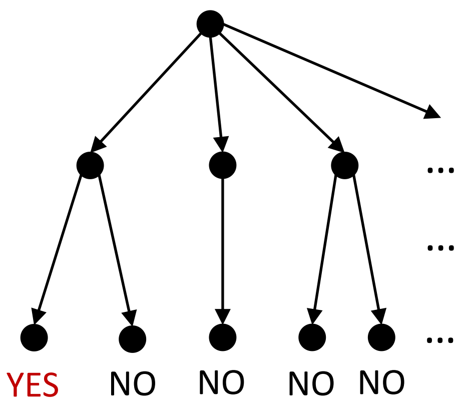
NonDetAlg(x):

```
cer = GenRndCertFree()
flag = Verify(cer,x)
if (flag == 1)
  Output("yes")
```

- **return value** of a non-deterministic algorithm $\mathcal{A}$ on input x is "yes" iff *some* execution of $\mathcal{A}(x)$ *outputs* "yes".
- Non-deterministic algorithm $\mathcal{A}$ for $\mathcal{P}$ is **polynomially bounded** if:
  for each "yes" instance I of $\mathcal{P}$, $\mathcal{A}(I)$ *returns* "yes" in $(|I|)^{O(1)}$ time.
- **NP** is the set of decision problems that have polynomially bounded non-deterministic algorithms.

Different faces of **NP**

The NTM Approach



NP is the set of decision problems that can be decided by NTM within polynomial time.

The Non-deterministic Algorithm Approach

NonDetAlg(x):

```
cer = GenRndCertFree()
flag = Verify(cer,x)
if (flag == 1)
  Output("yes")
```

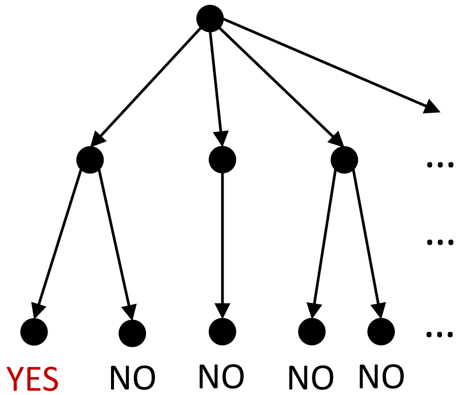
NP is the set of decision problems that have polynomially bounded non-deterministic algorithms.

NP is the set of decision problems that "yes" instances have short proofs that are efficiently verifiable.

SAT: A Problem in **NP**

- Given a Boolean formula ϕ in CNF, is ϕ satisfiable?
- **Example:** $\phi = (x_1 \lor x_2) \land (x_3 \lor \overline{x_1}) \land (x_2 \lor \overline{x_1} \lor \overline{x_2}) \land (x_4)$

The NTM Approach



Each branch is a truth assignment for a Boolean variable.

The Non-deterministic Algorithm Approach

NonDetAlg(x):

```
cer = GenRndCertFree()
flag = Verify(cer,x)
if (flag == 1)
  Output("yes")
```

Each cer contains a truth assignment for all Boolean variables.

Correctness of truth assignment can be verified in polynomial time.

NP is the set of decision problems that "yes" instances have short proofs that are efficiently verifiable.

$\mathbf{P} \subseteq \mathbf{NP}$

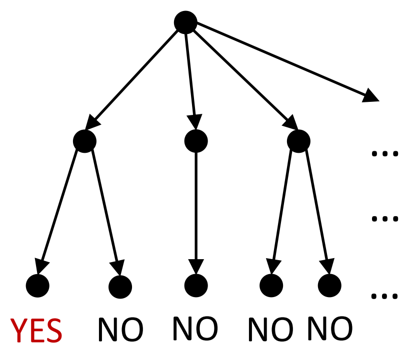
- **P** is the set of decision problems that have polynomially bounded algorithms.
- **P** is the set of decision problems that can be decided by (deterministic) TM within polynomial time.

- **NP** is the set of decision problems that have polynomially bounded non-deterministic algorithms.
- **NP** is the set of decision problems that can be decided by NTM within polynomial time.

- Any algorithm is also a special non-deterministic algorithm, any TM is also a special NTM.

The big question: $\mathbf{P} \neq \mathbf{NP}$?

- Most people believe $\mathbf{P} \neq \mathbf{NP}$.
- Informally, NTM and non-deterministic algorithm allows *exponential* "trials" within *polynomial* time.

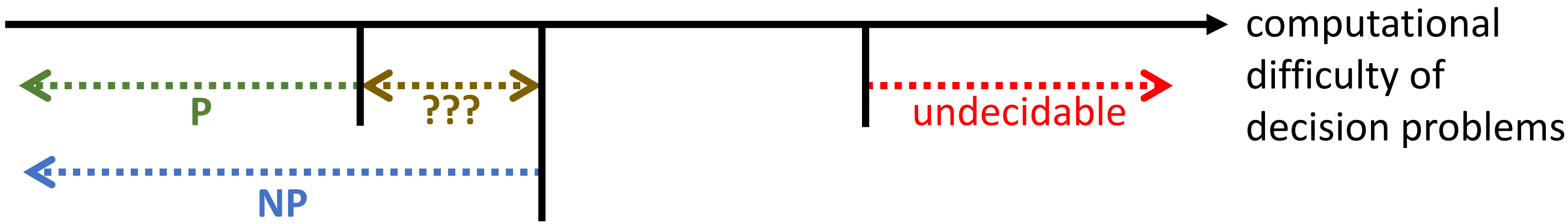


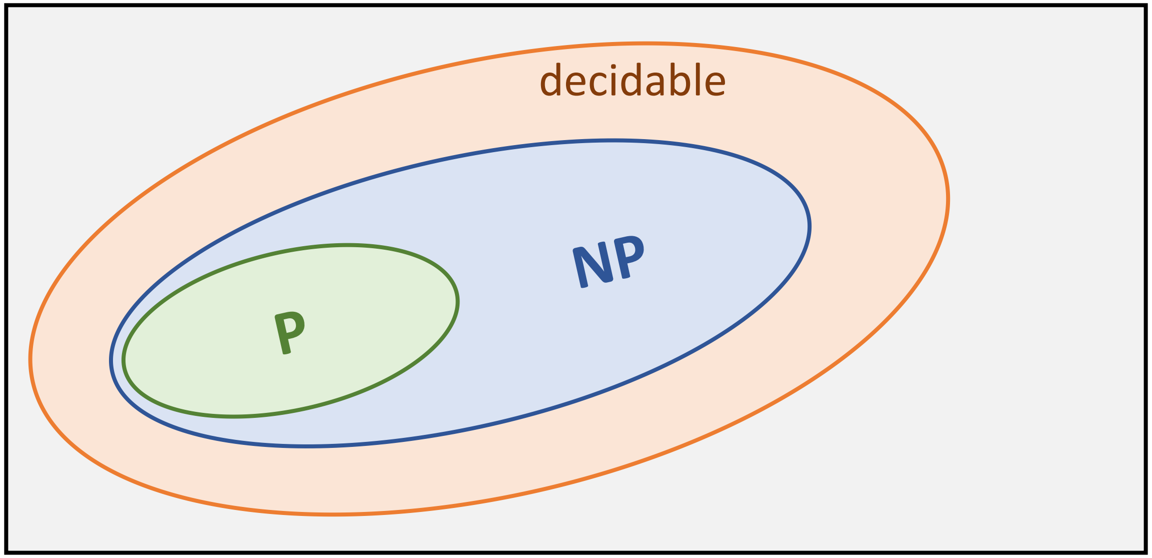
NonDetAlg(x):

```
cer = GenRndCertFree()
flag = Verify(cer,x)
if (flag == 1)
  Output("yes")
```

- **P** is the set of decision problems efficiently solvable.
- **NP** is the set of decision problems efficiently verifiable.
- **Solving a problem *should* be harder than verifying an answer?!**
- Yet we haven't found any $\mathcal{P} \in \mathbf{NP}$ while $\mathcal{P} \notin \mathbf{P}$.

If indeed $\mathbf{P} \neq \mathbf{NP}$ ...

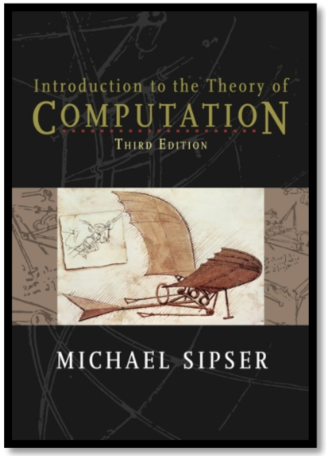


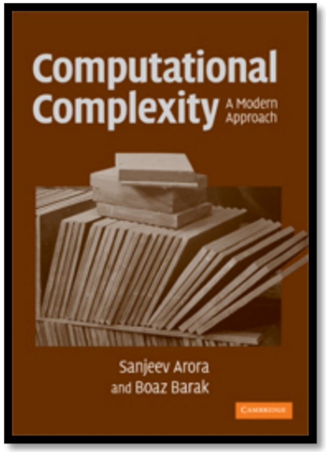


Reading

- [CLRS] Ch.34 (34.1-34.2)

- Other great books:
- ***Michael Sipser***, Introduction to the Theory of Computation (3ed)
- ***Arora*** *and* ***Barak***, Computational Complexity: A Modern Approach





(Very) Brief Review

- **Some important data structures**
  - list, stack, queue, heap, graph, …
  - hash tables, search trees, disjoint sets, …

Use computers to efficiently solve practical problems.
(Analytical and problem-solving skills.)

- **Basic algorithm design and analysis techniques**
  - induction, asymptotic notations, amortized analysis, …
  - recursion, divide-and-conquer, greedy, dynamic programming
- **Basic complexity theory**
  - upper bounds and lower bounds (decision tree, adversary argument)
  - computability and complexity (P, NP)