**P-class of programs that can be solved in polynomial time.**

*NP* - non-deterministic polynomial problem. Once you have a guess for this problem you have a feasible algo. \(c(x,y)\) that checks if \(y\) is a solution for the problem given \(x\).

**NP-hard**: Any NP problem can be reduced to a problem from this class. It is harder than any NP problem.

**NP-complete**: A NP-hard problem that is also NP.

2. **Product**: This is of class P because it takes polynomial time.
   - This is of class NP because once we have a guess we can check for solution in poly time.
   - If \(P \neq NP\), as believed, this problem is not NP-hard nor NP-complete.

3. **Interval comp**: This problem is NP-hard because SAT can be reduced to it.
   - This problem is not P because it takes exponential time (at least) for now.
   - This problem is not NP because it is not enough to only find one solution, so an exhaustive search must be performed.
   - Since it is not NP \(\Rightarrow\) not NP-hard, we proved that is NP-hard in class.
4. \((x_1 \lor x_2) \land (\neg x_1 \lor \neg x_3) \land (x_1 \lor x_2 \lor \neg x_3)\)

3-coloring

Solution:
\(x_1, x_3, x_2\)

5. Clique

6. Subset sum

Each is one coin
7) Interval computations
\((x_1 \lor x_2) \land (\neg x_1 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3)\)

\(7a \rightarrow 1-a\)
\(a \land b \rightarrow a \lor b\)
\(a \lor b \rightarrow (1-(1-a) \cdot (1-b)\)
\(x_1 \lor a \in [0,1]\)
\(x_2 \lor b \in [0,1]\)
\(x_3 \lor c \in [0,1]\)

\[ F = (1-(1-a)(1-b))(1-a+c)(1-(1-a)(1-b)(c)) \]

8) Product of \(n\) numbers:

To see if it belongs to \(NC\), we need to verify that this algorithm can be solved in a polynomial number of processors in polylog time. In this case for example if \(n=8\)

\[
\begin{align*}
& a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7 \quad a_8 \\
1 \quad \text{mult} \quad \text{mult} \quad \text{mult} \quad \text{mult} \quad \text{mult} \quad \text{mult} \quad \text{mult} \\
2 \quad \text{mult} \quad \text{mult} \\
3 \quad \text{mult} \\
4 \quad \text{processors}
\end{align*}
\]

We can solve the problem with \(\frac{n}{2}\) processors in \(\log_2 n\) timesteps therefore it belongs to \(NC\)!

9) \(NC \subseteq P\)? Yes, we can serialize the parallel computations and still take polynomial time.

\(NC = P\)? We don't know. To verify this, a \(P\)-hard problem such as linear programming should be feasibly parallelized but no proof or disproof has been found yet.
In this case, the computation is limited by the speed of the communication. Assuming that the communication is as fast as the speed of light, we have: 

\[ R = c \cdot T_{par}(n) \] 

is the radius of the sphere where we can fit processors that can communicate with each other. The volume is: 

\[ V = \frac{4}{3} \pi R^3 \] 

The max number of processors 

\[ T_{seq}(n) \leq N_{proc} \cdot T_{par}(n) \] 

\[ T_{seq}(n) \leq \text{const} \cdot (T_{par}(n))^4 \] 

This is the fastest assuming Euclidean physics.

This argument of using the speed of light as the maximum transfer speed is used in the proof of NP-hardness of SAT to show that any given cell depends on a finite polynomial number of neighbors and that there is only a finite polynomial number of states. Because this states can be represented in a binary way, it is possible to show that to compute SAT we need \( 2^{\text{poly}(n)} \).

The physics are based on Euclidean arguments. We state that \( V \leq C \) and that \( V = \frac{4}{3} \pi R^3 \), and because information can't travel faster, we can only use polynomial number of processors which are not enough to solve exponential-time problems.

Non-Euclidean physics can solve this for example:

- Lobachevsky geometry allows us to fit an exponential number of processors in the sphere which would allow to compute exponential time algs. in poly time.

- If speed can be faster than c, time travel is possible. One approach would be to leave a computer performing the computation and sending the result back in time. The other would be to exploit causality by having random number generators try to find a solution and trying to provoke very low probability events (much lower than finding the correct solution) by traveling back in time and trying to disrupt causality.
My project was to implement a Rough-Set based algorithm to predict execution time in multicore machines. The main part of the algorithm was:

1) Discretize the values and assign intervals
2) Create discernibility matrix (identify attributes which cause differences in decision values)
3) Reduce discernibility matrix using set absorption and deletion.
4) Predict the execution time only with reduced records.

One project was a new approach to identify and stop SQL injection. The structure of the source code is analyzed and transformed into Context Free Grammar, which allows to check if the input receive by the user contains malicious statements that try to modify or obtain unauthorized data.