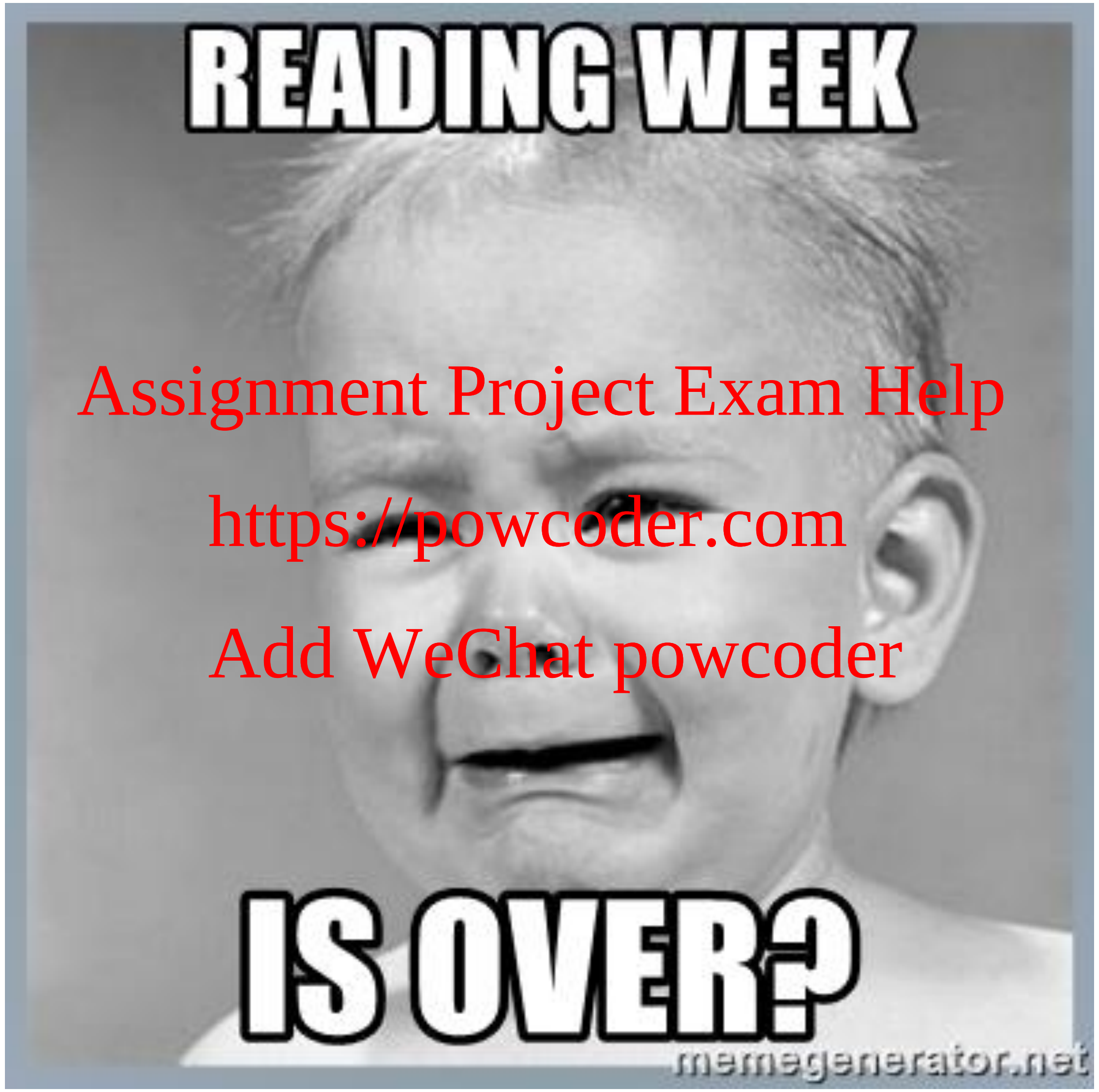
READING WEEK
IS OVER?
memegenerator.net

Arithmetic and Incompleteness

Finalizing the computability half of the course

Theory of Arithmetic

- The theory Th(ℕ) of all the facts about the structure of natural numbers is LIFE

- Naturally there is a desire to capture it through a manageable set of axioms

- By manageable I mean finite, or just computable

- By capture I mean axiomatize

- Sadly, this isn't possible (Gödel's Incompleteness Theorem)

Peano Axioms

- A suggested axiomatization for Th($\mathbb{N}$)

- From those axioms one can deduce (using a formal proof) many facts about the natural numbers

- … but not every fact

Gödel’s First Incompleteness

- Within the language of arithmetic, Gödel used his numbering tricks to make sentences speak about themselves (self reference)

- The idea is to create a formula $P(x, y)$ using $0, +, \times, (,), s, \rightarrow, \neg, \ldots$ such that y is the Gödel number of a proof in PA of the sentence whose Gödel number is x

- Look now at this sentence: $\neg\exists y P(e, y)$ where $e = gn(\neg\exists y P(e, y))$

- **It** says e (myself), not provable

- **We** see (as outsiders to PA) that it is true, but PA does not

Gödel's Second Incompleteness

- Gödel decided to play more with his numbering trick and created a sentence that speaks about PA (about the system from within the system)

- The sentence said: PA is consistent.

- Consis(PA): $\neg\exists y P(gn(\neg(0 = 0)), y)$ (there is no proof of $0 \neq 0$)

- Then Gödel showed that: PA $\nvdash$ Consis(PA)

- In other words, PA cannot prove its own consistency

Generalizability of the Incompleteness Theorems

- All those proofs of Gödel just required that the system is powerful enough to express arithmetic

- So, he was able to prove similar facts about, e.g., set theory

- $0 = \emptyset,\ 1 = \{\emptyset\},\ 2 = \{\emptyset, \{\emptyset\}\}, \dots, n = \{0, 1, \dots, n-1\}$

In philosophical terms

- A system which is powerful (enough to describe arithmetic) does not have a decidable list of axioms from which every fact would follow

- Imagine yourself creating a manageable (finite or computable) list of rules (laws) from which everything in your system of interest should follow.

- Unless the system is very weak, we can’t

Peace out Computability

Theory of Complexity

Inside what computers can do

Computation

- Formality produced models of computation: Turing Machines, Recursive Functions
- Other weaker models with restricted memory: Finite Automata, Pushdown Automata
- Turing Machines are a much more accurate model of a general purpose computer
- Church-Turing thesis connects real-world with theory
- Formality enable us to tell what computers **can’t** do
- Formality made concepts like randomness tangible
- I would like you to take a look at Sipser’s book

Complexity Analysis

- Formality does not only help us tell what computers can’t do, it also allows a general rigorous way to discuss computation resources (time and space)

- What is **efficient** computation? Turing Machines formalize efficiency and enable **measuring** it in a standard way

- Time as the number of steps (or transitions). Space as the number of tape cells used.

Complexity Measures

- Time Measure: $t(i,x) = \min\{s\colon \varphi_{i,s}(x)\downarrow\}$
- Space Measure:

$$M(i,x) = \begin{cases} \mathit{The\ number\ of\ cells\ visited\ by\ the\ reading} & \\ \mathit{head\ while\ computing}\ \varphi_{i,s}(x) & \text{if}\ \ \varphi_{i,s}(x)\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

- Those are examples of *complexity measures*

- A *complexity measure* is a more general concept (check Blum Axioms)

Polynomial Time Computability

- A function f is polynomial time computable if:

1. There is e such that $f = \varphi_e$
2. There is a polynomial $p(n)$ such that $t(e, x) \leq p(|x|)$ for every binary string input x

- Such a function is called *tractable, or efficiently computable*

We should think in Turing Machine terms

- Since the concepts we are discussing now are mechanical, we switch our terminology from p.c. functions to TMs

- We will work with TMs that halt on all inputs (total). In other words, all our TMs will be deciders

- $time(\mathrm{M}, x)$ = The number of steps M takes to accept/reject input x

Determinism vs Nondeterminism

- When a TM is in a given state and reads the next input symbol, we know what the next state will be (determined)
- In a **nondeterministic** machine, several choices may exist for the next state at any point

- Transition function (Deterministic)

$$\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

- Transition function (Nondeterministic)

$$\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$$

In some references: $\delta: Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$

Deterministic vs Nondeterministic

- Deterministic is a special case of Nondeterministic
- However, every Nondeterministic TM can be simulated by a Deterministic one (why? Hint: breadth-first search)

Time Complexity

- Let M be a deterministic TM. The ***running time*** or ***time complexity*** of M is the **function** $f: \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of size n

$$\begin{aligned} f(n) &= \max\{s : M(x) \text{ halts in exactly } s \text{ steps}, |x| = n\} \\ &= \max\{time(M, x) : x \in \Sigma^n\} \end{aligned}$$

- So, for all input strings x of length n, $M(x)$ halts **within** $f(n)$ steps

- We say M runs in time $f(n)$, or that M is an $f(n)$ time TM

Asymptotic Analysis (O notation)

- Running time is often a complex expression
- We usually are only interested in estimating it
- Example: if the running time is $f(n) = 6\,n^3 + 2\,n^2 - 20\,n + 45$, then we describe the running time as $O(n^3)$
- Generally, we write $f(n) = O(g(n))$ if
$$\exists c\ \exists n_0\ \forall n \geq n_0,\ f(n) \leq c\ g(n)$$
- $g(n)$ is said to be an *asymptotic upper bound*

Example: The sorting problem

- Input: a sequence of n numbers $a_1, a_2, \ldots, a_n$
- Output: a reordering $a'_1, a'_2, \ldots, a'_n$ of $a_1, a_2, \ldots, a_n$ such that

$$a'_1 \le a'_2 \le \cdots \le a'_n$$

Idea:

Look at a_2. If $\le a_1$, move it before a_1. So we obtain $a_2, a_1, \ldots, a_n$. Else, leave the ordering as it is, look at a_3, and compare it with a_2

.....

- This is known as *insertion sorting*

Clarification with numbers:

Input: 5,2,4,6,1,3

(a) 5,2,4,6,1,3 (At most 1 step)
(b) 2,5,4,6,1,3 (At most 2 steps)
(c) 2,4,5,6,1,3 (At most 3 steps)
(d) 2,4,5,6,1,3
(e) 1,2,4,5,6,3
(f) 1,2,3,4,5,6

- Total number of steps in worst-case scenario = 1+2+…+6
- In general, with input of size n, it will be $\frac{n(n+1)}{2} = O(n^2)$

Complexity Classes

- For any function $f\colon \mathbb{N} \to \mathbb{R}^+$, and $n \in \mathbb{N}$:

$$TIME\big(f(n)\big) = \{L : L \text{ is a language decidable by some TM that runs in worst case time } O(f(n))\}$$

$$SPACE\big(f(n)\big) = \{L : L \text{ is a language decidable by some TM that runs in worst case space } O(f(n))\}$$

The Class P

- $P = \{L: L \text{ is a language decidable by some polytime TM}\}$

- Note that $P = \bigcup_k TIME(n^k)$

Polytime Reducibility

- $A \leq_p B$ if $A \leq_m B$ via an m-reduction f which is polytime

- Fact: If B is decidable in polytime, and $A \leq_p B$, then A is also decidable in polytime