CMIMC 2019

Power Round

INSTRUCTIONS

1. Do not look at the test before the proctor starts the round.
2. This test consists of several problems, some of which are short-answer and some of which require proofs, to be solved within a time frame of **60 minutes**. There are **80 points** total.
3. Answers should be written and clearly labeled on sheets of blank paper. Each numbered problem should be *on its own sheet.* If you have multiple pages, number them as well (e.g. 1/3, 2/3).
4. Write your team ID on the upper-right corner and the problem and page number of the problem whose solution you are writing on the upper-left corner on each page you submit. Papers missing these will not be graded. Problems with more than one submission will not be graded.
5. Write legibly. Illegible handwriting will not be graded.
6. In your solution for any given problem, you may assume the results of previous problems, even if you have not solved them. You may not do the same for later problems.
7. Problems are not ordered by difficulty. They are ordered by progression of content.
8. No computational aids other than pencil/pen are permitted.
9. If you believe that the test contains an error, submit your protest in writing to Doherty 2302 by the end of lunch.

In this power round, we will explore approximation algorithms and use them to analyze Sudoku-solving algorithms.

Contents

1 Approximation Algorithms [40 points]

In this section we explore the viability of approximating so-called "hard" problems in polynomial time.

1.1 Background Information [10 points]

1.1.1 Languages [3 points]

Much of Theoretical Computer Science hinges on getting around hard problems: ones we cannot solve efficiently. To explore this, let's define some concepts.

- Define an *alphabet* Σ to be a set of characters, for example $\Sigma = \{a, b\}$ or $\{0, 1, 2\}$. Throughout this document, we will restrict our attention to finite alphabets.
- For Σ an alphabet, the set Σ^* is the set of all **finite** concatenations of characters in Σ. For example, if $\Sigma = \{a\}$, then
$$\Sigma^* = \{\varepsilon, a, a^2, a^3, \dots\} = \{\varepsilon\} \cup \{a^n : n \geq 1\},$$
where ε is the empty word.
- A *language* L over some alphabet Σ is a subset of Σ^*. L is not necessarily finite!

Problem 1.1 (3 points)

Give English descriptions for the following languages. [1 pt each]

(a) $L = \{0^n 1^n : n \in \mathbb{N}\}$

(b) $M = \{ww^r : w \in \Sigma^*\}$

(c) The language recursively defined as follows: $\varepsilon \in O$ and if $u, v \in O$ then $1u0v, 0u1v \in O$.

(a) The strings which start with some number of 0s and end with the same number of 1s, and have nothing in between.

(b) Even length palindromes.

(c) The set of strings with an equal number of 0s and 1s.

1.1.2 Complexity Classes [4 points]

We have the tools now to define (at a high level) classes of languages.
Define P (shorthand for polynomial time) as the set of all languages L for which there exists an algorithm A such that

- the running time of A on a string $w \in L$ is polynomial in the length of w;
- A accepts on input w if and only if $w \in L$.

For example, the language $L = \{0^{2n} : n \in \mathbb{N}\} \subseteq \{0\}^*$ is in P. Here is a polynomial time algorithm which checks to see if a string w is in L.

Algorithm 1 An algorithm to test membership in L.

```
procedure CHECKL($x$)
    len ← length of $x$
    count ← 0
    for $i = 1$ to $i =$ len do
        count ← 1 − count
    if count = 0 then
        return true
    else
        return false
```

Notice the requirement of being polynomial time is in the **length** of n. This means that if n is an integer, its length is defined to be the number of binary digits in its binary expansion. If the input is a list, its length is defined as the sums of the lengths of the elements.

Now we define NP (shorthand for non-deterministic polynomial time) as the set of all languages for which there exists a polynomial time *verifier* in the length of the input. In other words, we might not be able to solve the problem in polynomial time, but given some "witness" to the current input, we can check if that input really is in the language in polynomial time. For example, the following is a verifier for the question if a number is composite:

Algorithm 2 Testing compositeness of an integer.

```
procedure VERIFIER($n, k$)
    if $k > 1$ and $k < n$ and $n \bmod k = 0$ then
        return true
    else
        return false
```

Notice that the modulo operation is polynomial time. Furthermore, note that there exists a k such that VERIFIER(n, k) returns true if and only if n is composite. This implies that $\mathsf{INTFACT} \in \mathsf{NP}$.
It turns out these complexity classes are deeply related; in fact, there is a 1 million dollar prize open (Millenium Problems) for proving or disproving the equality $\mathsf{P} = \mathsf{NP}$! Although this is hard, at least one of the directions is easy.

Problem 1.2 (4 points)
Prove that $\mathsf{P} \subseteq \mathsf{NP}$.

Let $L \in \mathsf{P}$ be arbitrary and let A be a polynomial time algorithm solving L. Consider the following verifier:

```
def V(n,k):
  return A(n)
```

Then if $n \in L$, $A(n)$ will accept as desired. So if $n \in L$, all $k \in \Sigma^*$ act as witnesses. On the other hand, if $n \notin L$ there cannot be any witnesses under which this accepts as k is not used in the algorithm. Now note that A is polynomial time, so V must also be polynomial time. Hence we are done.

1.1.3 Hard problems [3 points]

It may seem as if NP is a complexity class which should cover almost everything. However, we can go further. Define the class of NP-*hard* problems as the collection of problems which are at least as hard as anything in NP. More formally, if we could solve any of these problems, then we could use them as subroutines to solve any other problem in NP.

We can then define the class NP-*complete* as the set of all languages which are both in NP and are NP-hard. It is not an obvious result that such problems exist, but they do, and we will not prove this here for time and scope reasons.[1]

Here are a few NP-complete problems.

- 3COL: Given a graph G, can its vertices be colored in 3 colors such that no two adjacent vertices are given the same color?
- COL: Given a graph G and a positive integer k, can its vertices be colored in k colors such that no two adjacent vertices are given the same color?
- SUBSET–SUM: Given a list L of integers and an integer t, does there exist a $S \subseteq L$ such that
$$\sum_{s \in S} s = t?$$
- PARTITION: Given a list L of integers, does there exist a set $S \subseteq L$ with the property that
$$\sum_{s \in S} s = \sum_{s \in L \setminus S} s?$$

Problem 1.3 (3 points)

Let's get some practice with these problems.

(a) [0.5] Explain (informally!) why COL is NP-complete using 3COL.

(b) [0.5] Explain why SUBSET–SUM is NP-complete using PARTITION.

(c) [2] Explain why PARTITION is NP-complete using SUBSET–SUM.

[1] The main idea is to encode the idea of computing a problem itself into a decision problem.

(a) One is a strict generalization of the other, so if we could solve COL we would have a general algorithm to solve 3COL.

(b) Let $2L$ denote the list formed by doubling every element in L, and x denote the sum of the elements in L. Then note that if we can find a subset of $2L$ with sum x, we can solve PARTITION. Of course, this goes both directions.

(c) Let x denote the sum of the elements in L. Then consider $L' = L \cup \{2t - x\}$. We claim that $L \in \text{SUBSET–SUM} \iff L' \in \text{PARTITION}$.

For the forward direction, let S be the sublist with sum t. Since the sum of elements in L' is $2t$, S has sum t so $L' \in$ PARTITION.

For the backward direction, note that one of the partitions does not contain our added $2t - x$ element, so this partition must have sum t and is only made up of elements from L.

1.2 Graph Approximations [30 points]

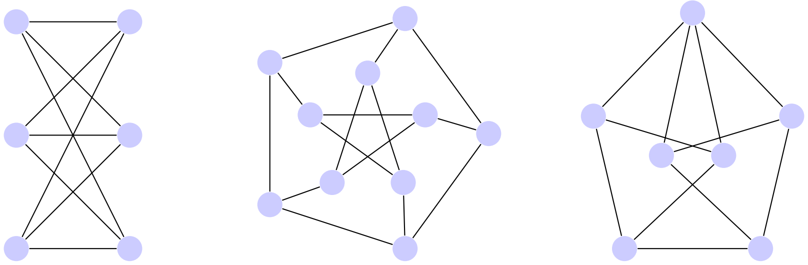
Although some problems are currently too "hard" to be exactly solved with our current knowledge, this does not mean we should shrug and give up, as these problems appear frequently in applications. It may still be reasonable to find efficient algorithms which get "close enough" to the exact answer. It turns out this is a viable approach and leads to a whole area of computer science research dubbed *Approximation Algorithms.*

1.2.1 A background in coloring [3 points]

Let's familiarize ourselves with graphs and colorings before getting started. Let $G = (V, E)$ be a graph, where V is its set of vertices and $E \subset \binom{V}{2}$ is its set of edges. Now we assign a color to each vertex. Call an edge *bichromatic* if the two vertices it is connected to have different colors. We say a graph is *k-colorable* if there exists a way to assign colors to all of the vertices with k colors such that every edge is bichromatic. Then let $\chi(G)$ be the smallest value of k such that G is k-colorable; we call this number the *chromatic number* of G.

Problem 1.4 (3 points)

Determine the chromatic number of each of the following graphs. No justification is required. [1 pt each]



(a) This is a bipartite graph ($K_{3,3}$) with chromatic number 2.

(b) This is the Petersen graph and has chromatic number 3.

(c) This is the Moser Spindle and has chromatic number 4.

1.2.2 Approximating Colorings [27 points]

In this section, we are not really considering decision problems anymore. Instead, we are considering *optimization* problems: ones in which we are trying to minimize or maximize some quantity. For example,

suppose we know that a graph is 3 colorable, but do not know the coloring. Then we could try to minimize the number of colors we have to use to color this graph in polynomial time.
Let I be an instance of a minimization problem: for example, it could be a graph given for the 3COL problem. We have a function OPT : instance → response which gives us the "best possible" solution for the given instance. Further let A denote our (maybe) suboptimal algorithm. For all reals $\alpha > 1$, we say that A is an *α-approximation algorithm* if

$$A(I) \leq \alpha \cdot \mathsf{OPT}(I) \quad \text{for all instances } I.$$

We will use this notion of approximation to analyze some more coloring algorithms.
The following problems provide useful lemmas for the upcoming approximations.

Problem 1.5 (3 points)

For any vertex v in a graph H, let the *degree* of v be the number of vertices w such that $\{v, w\}$ is an edge in H. Let G be an arbitrary graph with maximum degree Δ. Prove that $\chi(G) \leq \Delta + 1$.

We color greedily, starting from an arbitrary vertex. Assume for sake of contradiction that there exists a vertex which we have to color with color $\Delta + 2$. Then this means that its neighbors inhabit all the colors $1 \leq c \leq \Delta + 1$, impossible since there are at most Δ neighbors.

Problem 1.6 (2 points)

Prove that any 2-colorable graph can be colored exactly (1-approximation) in polynomial time. *One way to do this is to provide a high-level exact algorithm and explain why it is correct.*

Color greedily. Since the graph is guaranteed to be 2-colorable, we know that all neighbors of a vertex must be the other color, and there are no other options.

Now we can get a pretty good approximation for 3COL. Here is the algorithm:

Algorithm 3 Coloring a graph with chromatic number 3.

```
procedure 3Color($G$)
    colors ← $\varnothing$
    while $\deg(G) \geq \sqrt{n}$ do
        $v$ ← vertex of largest degree
        color $v$ with some color $c$, add $c$ to colors
        2-color neighbors of $v$, add these colors to colors
        delete $v$ and its neighbors from $G$
    Finish using the algorithm from Problem 1.5
```

Problem 1.7 (6 points)

Now we can prove the guarantees of the above algorithm.

(a) [2] Prove that 3color is a $\frac{4}{3}\sqrt{n}$-approximation algorithm.

(b) [4] Improve to a $\sqrt{n}$-approximation by slightly modifying the given algorithm.

(a) The notation is the tricky part: we just want to show this uses at most $4\sqrt{n}$ colors. Note that in each iteration of the loop we use 3 colors, and since there can be at most $n/\sqrt{n} = \sqrt{n}$ nodes with $\Delta(v) \geq \sqrt{n}$, this step uses $3\sqrt{n}$ colors. Now since the remaining graph has $\Delta(G) < \sqrt{n}$, this will use at most $\sqrt{n}$ colors. Hence in total we use $4\sqrt{n}$ as desired.

(b) The idea is to just not color v until the end, as the remaining G will have these as isolated vertices. Then we will only use $2\sqrt{n}$ colors while looping, giving $3\sqrt{n}$ in total.

We can actually improve further, by choosing the function $2\sqrt{n}$ instead of $\sqrt{n}$. This uses $2\sqrt{2n}$ colors.

In fact, we can actually construct a general algorithm for COL, assuming we know $\chi(G)$. First, for ease of notation set $f(n,k) = n^{1-1/(k-1)}$. Now we construct the following recursive algorithm when $2 \leq k < \log n$.

Algorithm 4 Coloring a graph with chromatic number k.

```
procedure KColor($G, k$)
    colors ← ∅
    while $\deg(G) \geq f(n,k)$ do
        $v$ ← vertex of largest degree
        $H$ ← neighbors of $v$
        call KColor($H, k-1$), add all used colors to colors
        color $v$ with some color $c$, add $c$ to colors
    Finish using the algorithm from Problem 1.5
```

We have a recursive base case of $k = 2$, where from problem 1.6 we know that the coloring is exact.

Problem 1.8 (6 points)

Let $2 \leq k < \log n$. Prove that KColor$(\cdot, k)$ is an $(2\left\lceil n^{1-1/(k-1)} \right\rceil)$-approximation algorithm.
It may be helpful to recall Jensen's Inequality: if $f : \mathbb{R} \to \mathbb{R}$ is a concave function, then for any real numbers $x_1, \ldots, x_k$,

$$\frac{f(x_1) + f(x_2) + \cdots + f(x_k)}{k} \geq f\left(\frac{x_1 + x_2 + \cdots + x_k}{k}\right).$$

Again, notation is tricky. We want to show that we use at most $2k\left\lceil n^{1-1/(k-1)}\right\rceil$ colors.
We proceed by induction. Note that this is trivially true for $k=2$. Now assume it is true for $2 \leq k' \leq k$, we show $k+1$.
Note first that the loop is executed at most $n^{1/k}$ times, and let the number of executions be m. Further let the subgraphs H have sizes $h_1, h_2, \ldots, h_m$, and note that the sum of these sizes is at most n. Now we have

$$\begin{aligned}
\sum_{i=1}^{m} 2k\left\lceil h_i^{1-1/(k-1)}\right\rceil &\leq 2k\left(m+\sum_{i=1}^{m} h_i^{1-1/(k-1)}\right) \\
&\leq (2k+1)m\frac{\sum_{i=1}^{m} h_i^{1-1/(k-1)}}{m} \\
&\leq (2k+1)m\left(\frac{n}{m}\right)^{1-1/(k-1)} \\
&= (2k+1)n^{1-1/(k-1)}m^{1/(k-1)} \\
&= (2k+1)n^{1-1/(k-1)}n^{1/k(k-1)} \\
&= (2k+1)n^{1-1/k} \\
&\leq (2k+1)\left\lceil n^{1-1/k}\right\rceil
\end{aligned}$$

In this proof, we used Jensen's since $f(n,k)$ is a concave function in terms of n. Furthermore, we are justified in the second step since $k < \log n$.
Finally, the greedy stage takes another $\left\lceil n^{1-1/k}\right\rceil$ colors so the final total is $(2k+2)\left\lceil n^{1-1/k}\right\rceil = 2(k+1)\left\lceil n^{1-1/k}\right\rceil$ as desired.

Unfortunately, the above algorithms require prior knowledge of the chromatic number of the input G. Here is an algorithm which does not require this assumption.

Algorithm 5 Coloring a graph with arbitrary chromatic number.

```
procedure COLOR?($G$)
    colors $\leftarrow \varnothing$
    $W \leftarrow$ neighbors of $G$
    while $|W| > 0$ do
        $U \leftarrow W$
        $c \leftarrow$ some color not in colors
        while $|U| > 0$ do
            $v \leftarrow$ some vertex in $U$ with minimal degree in $G$
            color $v$ with $c$, remove $v$ and its neighbors from $U$
        add $c$ to colors
```

Problem 1.9 (10 points)
Prove that COLOR? is a $(3n\log k)/(k\log n)$-approximation algorithm, where k is the actual chromatic number.

Again we just want to show that we use at most $3n/\log_k n$ colors. Note that by PHP there exists an independent set of size at least n/k when the graph is k-colorable. Hence there exists a vertex of degree at most $n(1-1/k)$ in G.

We claim that the inner while loop is executed at least $\log_k |U|$ times. Consider the first iteration of this inner loop, and let $m = |U|$. Then we will remove at most $m(1-1/k)$ vertices from U, leaving behind m/k. From there, we must iterate the process $\lfloor \log_k m \rfloor$ times to bring m to 0.

Now we split the algorithm into two parts. The first is while $|U| > n/\log_k n$ and the second is the other half. Note that no matter what we cannot use more colors than there are vertices, so in the second section we use up to $n/\log_k n$ colors.
In the first section, since $|U| > n/\log_k n$, then

$$\log_k |U| > \log_k n - \log_k \log_k n > \frac{1}{2} \log_k n$$

so each color covers at least $\frac{1}{2} \log_k n$ vertices. Hence we use at most $2n/\log_k n$ colors. Adding the two halves together gives the desired conclusion.

In fact, we can combine the previous two results to show that there exists a $3n(\log\log n)^2/(\log n)^3$ approximation for COL, but for time purposes we omit this.

2 Game solving [40 points]

In this section we use our approximation results and develop more to "solve" the game of Sudoku.

2.1 Basic Sudoku! [18 points]

2.1.1 Defining Sudoku [1 points]

Consider the standard game of Sudoku. The most common variant is played on a 9 × 9 grid, which we will later generalize. The game board starts off with some squares filled with numbers from 1 through 9. The goal of the game is to fill in the remaining squares with numbers in that same range so that: no two numbers in the same row, column, or gridded-off 3 × 3 boxes are the same. An example of a Sudoku board is shown below.

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Problem 2.1 (1 point)

Solve the above Sudoku board. *This is 1 point for a reason; don't spend all your time on this.*

This is filler text.

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

2.1.2 Bad Sudoku [4 points]

We can extend the notion of Sudoku to $n^2 \times n^2$ boards for arbitrary n. To solve these bigger Suduku boards, we make the key observation that Sudoku is really a graph coloring problem.

Let S be a Sudoku board, represented as a graph where two nodes are connected if they are in the same box, row, or column. Currently S is *partially colored*: that is, some of the nodes already have a color assigned to them. We can get around this, however. Construct a new graph S' as follows.

- Add n^2 new nodes to S, and label them with the colors $1, 2, \ldots, n^2$.
- Fully connect these new nodes amongst themselves.
- For each $1 \leq k \leq n^2$, connect all colored nodes with color k to all of the newly constructed nodes which do not have value k.

Problem 2.2 (2 points)
Prove that if S' has an n^2 coloring then there is a valid Sudoku solution (up to permutation of the numbers $1, 2, \ldots, n^2$).

Note that our original partial solution is unharmed: if there is an n^2 coloring then the partial solution must up to permutation be preserved since they are connected to every other color. Then, indeed S will have a valid solution as we have preserved the partial coloring and all the properties of a Sudoku board.

Great! Now we've transformed S into an instance (S', n^2) of COL. Applying the algorithm from Problem 1.8 gives a $6n^4 \log n/(4n^2 \log n) = \frac{3}{2}n^2$-approximation. This is a nontrivial result ... right?

Problem 2.3 (2 points)
Prove that any algorithm is an n^2-approximation.

First, note that any Sudoku grid must require at least n^2 distinct numbers (you can't use two of the same number in the same row, for example). There are only n^4 squares in the original Sudoku board, so giving each a distinct number is a n^2-approximation on its own.

So, our result really isn't the best.

2.1.3 Sudoku Board Approximation [13 points]

Now we create an algorithm for Sudoku.

Problem 2.4 (5 points)
Prove that there exists a 3-approximation algorithm for Sudoku. Here we want an explicit algorithm in addition to a proof. High level pseudocode is acceptable.
Note: You may not use the next problem in the solution to this one.

Since every node in the original Sudoku grid has at mot $3n^2 - 2n - 1 < 3n^2$ connections, we can simply use Problem 1.5.

This is very good! It's a constant factor approximation algorithm. Can we do better?

Problem 2.5 (8 points)
Indeed we can. Exhibit a k-approximation algorithm for some $k < 3$. The number of points you get depends on how close k is to 1.
Note: It is not enough to give an algorithm with an approximation ratio $3 - o(1)$.

First we show the following lemma.

Lemma. *An empty Sudoku grid can be colored exactly in polynomial time.*

Proof. We present a strategy. Color the first row with colors $1, 2, \ldots, n^2$ in order. Now split this up into blocks by n. Cyclically shift this for the next row: $n+1, \ldots, n^2, 1, \ldots, n$, and continue cyclically shifting for the first n rows. Then for the $n+1$ row do $2, 3, \ldots, n^2, 1$ and repeat the process. It is easy to see that this correctly fills the Sudoku board. □

Now we can do a 2-approximation: use the numbers $n^2+1, \ldots, 2n^2$ to fill an empty Sudoku grid as per the lemma, and then fill in the occupied slots from the partial solution.

2.2 Are you convinced? [22 points]

It turns out that try as we might, Sudoku is NP-complete. However, as brilliant CMIMC contestants, you think you've figured out a polynomial time solution to Sudoku!
The contest organizers are not convinced. They want you to demonstrate that your solution works, but you don't want them to know your solution. Is there a way for you to convince them of this fact? Surprisingly, the answer is yes!

2.2.1 Defining interactive proving [5 points]

In this section, we will discuss what it means to be an interactive proof. To do so, we define the computational class IP (shorthand for interactive polynomial time).
The key idea behind this computational class is that there are two parties: the prover P and the verifier V. In our setting, you (the contestants) are the provers and we (the organizers) are the verifiers. P wishes to demonstrate a proof of a fact to V. They do this via a protocol known as an *interactive proof system*. In this system, V queries P while preserving the following properties:

- **Completeness**: If P is trying to prove a true fact, an honest V will always accept P's proof.
- **Soundness**: If P is trying to prove a false fact, an honest V will reject P's proof except with a small probability.

From here, IP is defined as having a polynomially bounded V with access to randomness and an unbounded P, where V can query P a polynomial number of times.
Let's do an example. Assume you and a friend have 2 balls: one of which is blue and the other of which is red. Your friend is blindfolded and is convinced that the balls are the same color. You wish to prove that this is not the case. The following protocol is agreed upon:

1. Your friend puts both balls behind their back.
2. They show one of the balls to you and then put it back behind their back.
3. With $\frac{1}{2}$ probability, they swap the ball with the other and show the ball to you.
4. You answer if the ball is switched or not.
5. Repeat as long as your friend is unconvinced.

Let's prove the required two properties.

- **Completeness**: If you know the color of the ball, of course you know if it has been switched.
- **Soundness**: If they are the same color, then you will guess if it is switched or not with probability $\frac{1}{2}$. After repeating many times, the probability you are correct every time is negligible.

Problem 2.6 (5 points)

Prove that $\mathsf{NP} \subseteq \mathsf{IP}$.

Note: You may not reference problem 2.8 in any way: this would be circular reasoning from the exposition after that problem.

Let $L \in \mathsf{NP}$ be arbitrary. V sends an input n to P. P sends back a k, and V runs the verifier $A(n, k)$ and accepts iff this verifier accepts. Let's then show the two properties:

- **Completeness:** If $n \in L$, then there exists a witness k by definition of NP.
- **Soundness:** If $n \notin L$, then for all witnesses $k \in \Sigma^*$ we know that $A(n, k)$ will reject. Hence P cannot fool V with any probability of error.

2.2.2 Zero Knowledge Proving [5 points]

Now we extend the definition to include zero knowledge. In essence, we say that such an interactive proof is *zero knowledge* if there exists a simulator S from V's side which would be able to output the same transcript of interaction between P and V indistinguishably from any other transcript. In less technical terms, V only learns that there is a proof, but does not learn of the proof itself.

As an example, recall the ball example from the previous section. It also satisfies the Zero-Knowledge property: Since your friend knows if they switched the ball or not, they can recreate the whole transcript but do not learn if the balls are of different colors or not.

Problem 2.7 (5 points)

Recall the $\mathsf{3COL}$ problem. We demonstrate the following zero knowledge proof scheme for this (which shows some amazing results, as we prove later).

1. P creates a random permutation of the colors $\{1, 2, 3\}$ and assigns them the vertices of the graph G. P then commits to this coloring of the graph and seals it in an envelope (or other form, we have no preference).
2. V asks for an edge $(u, v) \in E$, the edge set of the original graph.
3. P reveals the colors a, b assigned to u and v and V verifies that they are different.
4. This is repeated (from the beginning) until V is satisfied or finds an error.

Your tasks:

(a) [1] Prove **Completeness**.

(b) [2] Prove **Soundness** and demonstrate an exact error bound for one iteration of this proof.

(c) [2] Prove **Zero-Knowledge** by showing that the transcript seen by V is indistinguishable from one made by randomly choosing distinct colors a, b.

(a) If the graph is indeed 3-colorable, then for any query V may have, the edge will be bichromatic.

(b) If the graph is not 3-colorable, then P is unable to 3-color the edges. Then there must be at least one edge with the same color assigned to both sides. In one iteration of the protocol then, the probability that V asks for such an edge is $\geq \frac{1}{|E|}$.

(c) Since every iteration the colors are permuted, we can emulate this by just choosing two distinct colors at iteration, as desired.

Remark. We could make this a non-zero knowledge proof by simply revealing the coloring of the graph.

This problem used the idea of a *bit commitment*: P can seal bits in an envelope which is given to V and can only be opened in the presence of V.
In fact, since 3COL is NP-complete and NP $\subseteq$ IP, this means that under the assumption that we have unbreakable encryption, all problems in NP have zero knowledge proofs![2]

2.2.3 Do you even Sudoku? [12 points]

As discussed in the previous section, there exists a zero knowledge proof of Sudoku, and we can probably find one using some reduction from 3COL. There's no fun in that, and who's to say that it won't have abysmal soundness error?
To do this, we need a notion of physical security. This was alluded to earlier by putting bits in an envelope. Physical security means securing information in the real world instead of through some cryptographic function (we could have done the previous problem using a *one way function*, a function which is easy to compute but difficult to invert).
Consider the following physical protocol for Sudoku, which uses playing cards.

1. P puts three playing cards face down on each square except the filled in ones, which are placed face up.
2. V makes piles for every row, column, and region by choosing one of the three playing cards for each.
3. P shuffles the cards.
4. V turns over each card and verifies that each packet is distinct and made of the numbers from 1 through n^2.

Problem 2.8 (8 points)

Prove **Soundness** and demonstrate that the probability of error is at most $\frac{1}{9}$.

[2] In fact, under the same assumptions all problems in IP have zero knowledge proofs.

Obviously, the probability of error is at most $\frac{1}{3}$. If the Sudoku! board has no solution, then P cannot put three of the same card on every square. They can only play to one of V's randomized choices for which playing card to take from each square. Hence the probability P gets away with it is at most $\frac{1}{3}$.

Now we analyze this a bit more: intuitively it seems that if P does this in one place then they must do it in another as well. Assume there are exactly 2 places where P cheats: squares a and b. Note that a must have (x, x, y) and b must have (y, y, x) for P to have any chance of succeeding, where x and y are arbitrary distinct integers between 1 and 9. Let's split into cases.

- If a and b share no characteristics (column, row, region) then these function as separate entities each with probability at most $\frac{1}{3}$ of success.
- Suppose a and b share exactly one characteristic, which we can assume is the row. Imagine P and V playing an alternate version of this game in which the cards at squares a and b are removed. Then there are two blank cards left in the common row pile and one blank card left in the four remaining piles. The two blank cards in the common row pile are x and y. If the x comes from b and the y comes from a, then this occurs with probability $\leq \frac{1}{9}$ automatically. If the x comes from a and the y comes from b, then this occurs with probability ≤ 49; but now there is at most a $\frac{1}{4}$ probability of fitting the remaining four cards correctly, and this obtains the desired bound.

  If the column and subgrid differ for both a and b, again we have a $\frac{1}{9}$ probability of success.

  In any other scenario, P cannot win.
- Suppose a and b share two characteristics. We assume this is the row and subgrid. Then if a needs a y card and b needs an x card for the column in order to succeed, this has a $\frac{1}{9}$ probability of success.

  If a needs x and b needs y, then P needs to cheat on more places to make this happen, but we assumed there are exactly 2 places.

  Otherwise P cannot win.

If there are more than 2 places P cheats, we can note that there are at most n values which commonly share the same 2 characteristics. So, if we have any pair of places which share less than 2 characteristics we are done with our analysis. Otherwise, we can simply enhance the above analysis to m values which share two characteristics.

This is a rather good algorithm then. But can we do better?

Problem 2.9 (4 points)

Indeed we can. It's now your turn! Give an interactive proof scheme for Sudoku (using physical security) with **0 probability of error**.

We create a notion of duplication: the idea is that P in front of V must "triplicate" a playing card for each square. Then there cannot be any probability of error.

Let's reflect on what we've done in this power round. We've showed some approximations for graph coloring and bounded how well we can solve Sudoku. You've also found a way to convince suspicious verifiers that you have a correct solution.
But now that you've demonstrated your algorithm works, those pesky verifiers want to actually verify that your polynomial proof and correctness proof of your algorithm are legit. The answer is yes and leads into a field of mathematics and computer science known as *proof theory.* It is possible to mechanically verify your proofs provided they are written in the correct format.[3]

[3] It is fun to note that most theorem provers are built on the NP-complete SAT problem, so checking a proof that P = NP would be a bit circular.