1. (3 points) Either prove that the following problem is NP-complete or show that it belongs to P by giving a polynomial time algorithm:

   - **Input:** An undirected graph $G$.
   - **Question:** Does $G$ have a proper colouring with three colours $R, G, B$ that assigns the colour $B$ to exactly one vertex?

   **Solution:** The problem is in P as there is a polytime algorithm that solves it. Note that the problem is basically asking to see if it is possible to turn $G$ into a bipartite graph after deleting only one vertex (the vertex with colour $B$). We can have a for-loop that goes over all the vertices, and for each one, we will check if $G$ is bipartite after removing that vertex. Checking whether a graph is bipartite (equivalently 2-colourable) can be done in polynomial time using DFS or BFS.
2. (8 points) Consider the triangle elimination problem. We are given an undirected graph \( G = (V, E) \), and want to find the smallest possible set of vertices \( U \subseteq V \) such that deleting these vertices removes all the triangles (i.e. cycles of length 3) from the graph. For each one of the following algorithms, either show that it is a 3-factor approximation algorithm, or give an example to show that it is not.

**Algorithm I:**
- While there is still a triangle \( C \) left in \( G \):
  - Delete all the three vertices of \( C \) from \( G \)
- EndWhile
- Output the set of the deleted vertices

**Algorithm II:**
- While there is still a triangle \( C \) left in \( G \):
  - Delete one arbitrarily chosen vertex of \( C \) from \( G \)
- EndWhile
- Output the set of the deleted vertices

**Algorithm III:**
- While there is still a triangle left in \( G \):
  - Delete a vertex that is in the largest number of triangles
- EndWhile
- Output the set of the deleted vertices

**Algorithm VI:**
- Write an Integer Program for this problem with constraints \( x_u + x_v + x_w \geq 1 \) for every triangle \( (u, v, w) \) in \( G \) (and constraints \( x_u \in \{0, 1\} \)).
  - Solve the LP relaxation of this Integer Program (with constraints \( 0 \leq x_u \leq 1 \)).
- EndWhile
- Output the set of the vertices \( u \) with \( x_u \geq \frac{1}{3} \).

**Solution:** Algorithm I is a 3-factor approximation algorithm. If the algorithm detects \( m \) triangles, then it removes \( 3m \) points. These \( m \) triangles are disjoint, and thus the optimal solution must remove at least 1 from each. Hence \( \text{opt} \geq m \), which implies output = \( 3m \leq 3 \times \text{opt} \).

**Solution:** Algorithm II is not a 3-factor approximation algorithm. Consider four triangles all sharing one vertex \( v \). In this case the optimal answer is 1 (just removing \( v \)), while the algorithm might remove 4 different vertices. \( 4 \not\leq 3 \times 1 \).

**Solution:** Algorithm III is not a 3-factor approximation algorithm. This one is difficult. We start with constructing a bipartite graph \( H \). Let \( L \) be a set of size \( r = k! = 2 \times 3 \times \ldots \times k \) for some large \( k \) to be determined later (we want \( r \) to be divisible to \( 2, \ldots, k \)). Let \( R_2 \) be a set of size \( r/2 \), \( R_3 \) be a set of size \( r/3 \), etc, and finally \( R_k \) be a set of size \( r/k \) (there is no \( R_1 \)).

For each \( 2 \leq j \leq k \), divide the \( r \) vertices in \( L \) into \( r/j \) equal parts of size \( j \), and connect each vertex in \( R_j \) to all the vertices in one of these parts (so no two vertices in the same \( R_j \) have any common neighbours in \( L \)).
Let us look at some of the properties of $H$. First note that $H$ is bipartite, and every vertex in $R_j$ is of degree $j$ (it is connected to $j$ vertices in $L$). On the other hand every vertex in $L$ is of degree $k - 1$ (it is connected to one vertex in each $R_j$ for $j = 2 \ldots, k$).

Now we finally construct $G$ by “gluing” a separate triangle on top of each edge of $H$. In other words, for each each $e = uv$ of $H$, we add a new vertex $w_e$ and connect both $u$ and $v$ to $w_e$.

Note that removing all the vertices of $L$ from $G$ will kill all the triangles. So $Opt \leq |L| = r$. Let us see how Algorithm III performs on this graph. Initially

- Every vertex in $R_j$ is in $j$ triangles (for each edge incident to it in $H$).
- Every vertex in $L$ is in $k - 1$ triangles.
- Every new vertex $w_e$ is only in 1 triangle.

So our algorithm will pick the vertices in $R_k$ and delete them one by one. This will remove all of $R_k$. After this every vertex in $L$ is going to be in $k - 2$ triangles, so it will start removing the vertices in $R_{k-1}$ which are in $k - 1$ triangles, etc. So our algorithm will remove all the vertices in $R_2 \cup R_3 \cup \ldots \cup R_k$, which is of size

$$\frac{r}{k} + \frac{r}{k-1} + \ldots + \frac{r}{2} = (1/2 + 1/3 + \ldots + 1/k)r.$$  

Remember that $Opt \leq r$, but if we take $k$ to be large (say $k = 3^{100}$) we will have that

$$(1/2 + 1/3 + \ldots + 1/k)r \geq 100r.$$  

Basically this algorithm is not a $C$-factor approx algorithm for no $C = O(1)$.

**Solution:** Algorithm VI is a 3-factor approximation algorithm. First note that since for every triangle $uvw$, we have $x_u + x_v + x_w \geq 1$, at least one of $u, v, w$ will be assigned a value $\geq 1/3$, and thus we will select at least one vertex from every triangle. In other words, the algorithm outputs a proper set that removes all the triangles.

Why is the solution at most 3 times the optimal? Because the linear program is a relaxation of the integer program that solves the actual problem (with constraints $x_u \in \{0, 1\}$ for all $u$) we have that

$$Opt(LP) \leq Opt(problem).$$

On the other hand the output is at most 3 times $Opt(LP)$. This is because

$$Output = \sum \overline{x}_u \leq 3 \sum x_u = Opt(LP) \leq Opt(problem),$$

where $\overline{x}_u = 1$ if $x_u \geq 1/3$ and $\overline{x}_u = 1$ if $x_u < 1/3$.

3. (4 points) We are given a graph $G$ together with an ordering of the vertices of $G$ such that every vertex $v$ has at most 5 neighbours that appear before $v$ in that order (but $v$ can have many neighbours appear later in the order).

- Show that the vertices of $G$ can be properly coloured using 6 colours.

**Solution:** We can colour the vertices greedily following the ordering. That is we colour every vertex with a colour that is currently available for that vertex. Note that when we colour a vertex, it has at most five neighbours that are already coloured, so there is always at least one colour available for that vertex.
Next we want to colour the vertices of $G$ with 5 colours so as to maximize the number of edges that are properly coloured (that is they have different colours on their endpoints). Design a $\frac{14}{15}$-factor approximation algorithm for this problem.

**Solution:** We start with the 6 colouring that we have already found. Let $V_1, \ldots, V_6$ be the set of the vertices that are coloured by colours 1, $\ldots$, 6 respectively. We find the $i \neq j$ such that the number of edges between $V_i$ and $V_j$ is minimized. Since there are $\binom{6}{2} = 15$ different choices for $ij$, we know that the number of edges between $V_i$ and $V_j$ is at most $|E|/15$.

We combine $V_i$ and $V_j$ by colouring all the vertices in $V_j$ with the colour $i$ as well. This way we eliminated one colour but now the edges between $V_i$ and $V_j$ (and only these edges) are not properly coloured. So the total number of edges that are properly coloured is at least $|E| - |E|/15 = 14|E|/15$. So

$$Output \geq 14|E|/15 \geq 14Opt/15,$$

because $Opt$ cannot be larger that $|E|$.

4. Let $x$ be a string of length $n$ of 0’s and 1’s. Consider the following operations:

- $\text{del}(x, i)$ (for $1 \leq i \leq n$) deletes the $i$-th bit of the string $x$, and thus decreases its length to $n-1$.
- $\text{set}(x, i, b)$ (for $1 \leq i \leq n$ and $b \in \{0,1\}$) sets the $i$-th bit of $x$ to the bit $b$.
- $\text{insert}(x, i, b)$ (for $1 \leq i \leq n+1$ and $b \in \{0,1\}$) inserts $b$ after the $i-1$-th bit of $x$, and thus increases the length of $x$.

Let $a$ and $b$ be two strings 0’s and 1’s. Define the distance $d(a, b)$ to be the smallest number of operations required to convert $a$ to $b$.

(a) (1 point) Show that $d(a, b) = d(b, a)$.

**Solution:** “Insert” is the reverse of “delete”, and “set” can be reversed with “set”. So if we have a sequence of $k$ operations that converts $a$ to $b$, we can find a sequence of $k$ operations that can convert $b$ to $a$, and vice versa. We have $d(a, b) = d(b, a)$.

(b) (2 point) Explain briefly how $d(a, b)$ can be computed in polynomial time using dynamic programming.

**Solution:** This is basically the sequence alignment problem (See 6.6 of the book). We define $D[i, j] = d(a[1 \ldots i], b[1 \ldots j])$, and then we can find a recursive formula for this. See also https://web.stanford.edu/class/cs124/lec/med.pdf

(c) (4 points) We are given 3 strings $a, b, c$, and we want to find a fourth string $d$ that minimizes $d(a, d) + d(b, d) + d(c, d)$. Give a 4/3-approximation algorithm for this problem.

**Solution:** We try $d = a, d = b,$ and $d = c$ and output the best one (we can find the distances in polytime using the previous dynamic programming part). That is our output is the smallest of $d(a, b) + d(a, c)$, and $d(a, b) + d(b, c)$, and $d(c, a) + d(c, b)$ which is obviously at most their average

$$Output \leq \frac{1}{3} (d(a, b) + d(b, c) + d(a, b) + d(b, c) + d(c, a) + d(c, b)) = \frac{2}{3} (d(a, b) + d(a, c) + d(b, c)).$$

Let $d^*$ be the optimal solution. Obviously

$$Opt = d(a, d^*) + d(b, d^*) + d(c, d^*) = \frac{1}{2} (d(a, d^*) + d(b, d^*) + d(a, d^*) + d(c, d^*) + d(b, d^*) + d(c, d^*)).$$
But by triangle inequality, \( d(a, d^*) + d(b, d^*) \geq d(a, b) \), and \( d(a, d^*) + d(c, d^*) \geq d(a, c) \), and \( d(b, d^*) + d(c, d^*) \geq d(b, c) \). Hence

\[
Opt \geq \frac{1}{2} (d(a, b) + d(a, c) + d(b, c)).
\]

This and what we had above shows

\[
Opt \leq \frac{4}{3} \text{Output}.
\]

Fun fact: No better algorithm is known.