1. Asymptotic notation properties
   a. TRUE
   \[ g_1 \in O(f_1) \rightarrow \exists n_1, c_1 \text{ such that } g_1(n) \leq c_1 f(n) \text{ for all } n \geq n_1. \]
   \[ g_2 \in O(f_2) \rightarrow \exists n_2, c_2 \text{ such that } g_2(n) \leq c_2 f(n) \text{ for all } n \geq n_2. \]
   Thus, \((g_1 \cdot g_2)(n) = g_1(n) + g_2(n) \leq (c_1 + c_2)f(n) \text{ for all } n \geq \max(n_1, n_2).\)
   So, for \(c = c_1 + c_2, n_0 = \max(n_1, n_2),\)
   \((g_1 \cdot g_2)(n) \leq cf(n), \text{ for all } n \geq n_0.\)
   b. TRUE
   \[ g \in O(f) \iff \exists c, n_0 \text{ such that } g(n) \leq cf(n) \text{ for all } n \geq n_0 \]
   \[ \iff \exists c, n_0 \text{ such that } f(n) \geq \frac{1}{c}g(n) \text{ for all } n \geq n_0 \]
   \[ \iff \exists c', n_0 \text{ (with } c' = \frac{1}{c} \text{) such that } f(n) \geq c'g(n) \text{ for all } n \geq n_0 \]
   \[ \iff f \in \Omega(g). \]
   c. FALSE
   For \(f(n) = 2^n \ (f(\frac{n}{2}) = 2^n = (\sqrt{2})^n), \) we have \(2^n \not\in \Theta(\sqrt{2}^n).\)
   \(2^n \not\in \Theta(\sqrt{2}^n) \) since \(2^n \not\in O(\sqrt{2}^n).\)
   If it were, then \(\exists n_0, c \text{ such that } \frac{2^n}{\sqrt{2}^n} = \sqrt{2}^n \leq c \text{ for } n \geq n_0, \) which is impossible.
   d. TRUE
   If \(g \in o(f), \text{ then } g \in O(f).\)
   So, \(\exists c, n_0 \text{ such that } g(n) \leq cf(n) \text{ for all } n \geq n_0 \)
   Thus, \((f + g)(n) = f(n) + g(n) \leq (c + 1)f(n) \text{ for all } n \geq n_0, \) which implies \(f + g \in O(f).\)
   On the other hand, for all \(n, (f + g)(n) = f(n) + g(n) \geq f(n), \) so \((f + g) \in \Omega(f).\)
   Therefore, since \(f + g \in O(f) \text{ and } f + g \in \Omega(f), \text{ then } f + g \in \Theta(f).\)

2. Encodings
   a.

<table>
<thead>
<tr>
<th>Type</th>
<th>Encoding</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>11111111</td>
<td>7</td>
</tr>
<tr>
<td>Binary</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td>Ternary</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
b. The algorithm takes \( n \) steps. The running time of the algorithm must be given in terms of \( l \), the encoding length.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Running Time (( n ) steps) as a function of ( l )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>( l = n )</td>
<td>( O(l) )</td>
</tr>
<tr>
<td>Binary</td>
<td>( l = \lfloor \log_2 n \rfloor + 1 &gt; \log_2 n )</td>
<td>( O(2^l) )</td>
</tr>
<tr>
<td>Ternary</td>
<td>( l = \lfloor \log_3 n \rfloor + 1 &gt; \log_3 n )</td>
<td>( O(3^l) )</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>( l = \lfloor \log_{16} n \rfloor + 1 &gt; \log_{16} n )</td>
<td>( O(16^l) )</td>
</tr>
</tbody>
</table>

3. Turing machines, RAM programs and analysis of algorithms

a. Turing machine \( M_1 \):

![Diagram of Turing machine]

Note: The symbol written to the tape is omitted, since it is the same already there.

Complexity of Turing machine \( M_1 \):
\( M_1 \) reads symbols until the first “\_” is found, in which case it stops. Thus, the running time is \( O(n) \).

b. The RAM program is as follows, encoding the end of string or empty space in the Turing machine by \(-1\):
1. LOAD =1
2. STORE 1
3. READ ↑1 (at \( k \)th iteration \( r_1 = k \) here)
4. STORE 2 (\( r_2 := i_k \))
5. LOAD 1
6. ADD =2
7. STORE 1 (at \( k \)th iteration \( r_1 = k + 2 \) here)
8. LOAD 2
9. STORE \( \uparrow 1 \) \( r_{k+2} := r_2 \) (kth input value)
10. ADD 1
11. JZERO 16 (if \( r_0(= r_{k+1}) = -1 \) go to 16)
12. LOAD 1
13. SUB =1
14. STORE 1 \( r_1 = k + 1 \) here
15. JUMP 3 (go to 3 for next iteration)
16. LOAD =3
17. STORE 1 \( r_1 := 3 \) (the index of register holding the first input))
18. LOAD \( \uparrow 1 \) (start simulation or \( q_1,1 \))
19. SUB =1
20. JZERO 22 (check if symbol is 1)
21. JUMP 26 (if not go to \( q_1,\downarrow \))
22. LOAD 1
23. ADD =1
24. STORE 1 (head move to the right)
25. JUMP 34 (transition to start of state \( q_2 \))
26. LOAD \( \uparrow 1 \) (start simulation of \( q_1,\downarrow \))
27. SUB =-1
28. JZERO 30 (check if symbol is -1)
29. JUMP 52 (if not, something is wrong; reject)
30. LOAD 1
31. ADD =-1
32. STORE 1 (head moves to the left)
33. JUMP 50 (transition to \( q_{accept} \))
34. LOAD \( \uparrow 1 \) (start simulation of \( q_2,1 \))
35. SUB =1
36. JZERO 37 (check if symbol is 1)
37. JUMP 42 (if not go to \( q_2,\downarrow \))
38. LOAD 1
39. ADD =1
40. STORE 1 (move head to right)
41. JUMP 18 (transition to begining of state \( q_1 \))
42. LOAD \( \uparrow 1 \) (start simulation \( (q_2,\downarrow) \))
43. SUB =-1
44. JZERO 46 (check if symbol is -1)
45. JUMP 52 (if not, something wrong! reject)
46. LOAD 1

3
47. \( \text{ADD} = -1 \)
48. \( \text{STORE} \ 1 \) (move head to the left)
49. \( \text{JUMP} \ 50 \) (transition to \( q_{\text{accept}} \))
50. \( \text{LOAD} = 1 \) (start simulation of \( q_{\text{accept}} \))
51. \( \text{HALT} \ (r_0 = 1) \)
52. \( \text{LOAD} = 0 \) (start simulation of \( q_{\text{reject}} \))
53. \( \text{HALT} \ (r_0 = 0) \)

Complexity of the RAM program:
The RAM program simulates the states of the Turing machine; for each state of the Turing machine, the RAM program takes at most 16 steps (steps 18-33 for \( q_1 \), 34-49 for \( q_2 \), 50-51 for \( q_{\text{accept}} \), 52-53 for \( q_{\text{reject}} \)), so the simulation takes \( O(n) \) steps. The initial loop that reads the \( n \) inputs also takes \( O(n) \), so the total time is \( O(n) \).

4. Multitape Turing machines
   a. Idea for 2-tape machine:

   1. Scan tape 1 all the way to the right, keeping tape 2 put, by moving it to the left. Note that, after reading the first symbol, it is substituted by another symbol: if the first symbol is 1, substitute it by ! and if it is 0, substitute it by @.

   2. Scan tape 1 right to left while copying its symbols to tape 2 from left to right (mark first symbol of tape 2 too). See states \( q_3, q_4 \).

   3. Move head of tape 2 to its beginning while leaving tape 1 put (by chosing to move left, while at the leftmost position). See state \( q_5 \).

   4. Scan both tapes from left to right, checking that symbols in each tape are the same. If at any point the symbols differ, reject. If both tapes reach their end always with equal symbols, accept. See state \( q_6 \).
b. Idea for 1-tape machine:
Repeat the process:

- Read the current symbol on the tape. If it is an $x$, then accept. Otherwise, write $x$ on top of it. Compare this symbol with the last one before _ or before an $x$. (If there are no more non-$x$s, accept.)

- If different, reject; otherwise, write $x$ on top of the last symbol.

- Go left until an $x$ is found. Move right.
5. \textbf{P, NP and co-NP.}

\textbf{a.} In all cases below since $L_1 \in \textbf{P}$, $L_2 \in \textbf{P}$, we know there exists a polynomial time algorithm $A_1$ that decides $L_1$ and $A_2$ that decides $L_2$. The proofs below show how to build $A_3$ which decides the appropriate language $L_3$ in polynomial time.

\textbf{a1.} $L_3 = L_1 \cup L_2$

Given $x$, $A_3$ will return 1 if and only if $A_1(x) = 1$ or $A_2(x) = 1$ (clearly polynomial time).

\textbf{a2.} $L_3 = L_1 \cap L_2$

Given $x$, $A_3$ will return 1 if and only if $A_1(x) = 1$ and $A_2(x) = 1$ (clearly polynomial time).
a3. $L_3 = L_1 L_2$
A works as follows on string $x$ of length $n$.
for each of the $n + 1$ prefixes $x_0$ of $x$ do

let $x_1$ be such that $x = x_0 x_1$
if $A_1(x_0) = 1$ and $A_2(x_1) = 1$ then return 1
end for
return 0

A runs in polynomial time since its complexity is in $O(n(T_1(n) + T_2(n)))$
where $T_i(n)$ is the running time of algorithm $A_i$, which was assumed to be
polynomial in $n$.

a4. $L_3 = \overline{L_1}$ (assuming $L_1$ in P)
Given $x$, $A_3$ will return 1 if and only if $A_1(x) = 0$ (clearly polynomial time).

b. Let $L \in P$, we must show that $L \in \text{co-NP}$. By part a we know that $\overline{L} \in \text{P} \subseteq \text{NP}$. By definition of co-NP, $\overline{L} \in \text{co-NP}$. But $\overline{L} = L$. Therefore, $L \in \text{co-NP}$.

c. This is equivalent to showing that $\text{P} = \text{NP}$ implies $\text{NP} = \text{co-NP}$.
Assume $\text{P} = \text{NP}$.
Let $L \in \text{co-NP}$. We know that $\overline{L} \in \text{NP}$, but since $\text{NP} = \text{P}$, we conclude that $L \in \text{P}$. By part a we know that $L = \overline{L} \in \text{P}$. Thus $\text{co-NP} \subseteq \text{P}$. By part b, we know that $\text{P} \subseteq \text{co-NP}$. Thus $\text{P} = \text{co-NP}$, and since $\text{P} = \text{NP}$, we conclude that $\text{NP} = \text{co-NP}$.

6. Decision problems vs. optimization problems
($\Rightarrow$)
Assume that LONGESTPATHLENGTH can be solved in polynomial time; let $A$
be a polynomial time algorithm for LONGESTPATHLENGTH. We will provide
an algorithm $A'$ for deciding LONGESTPATH.
Algorithm $A'$:
Input: $< G, u, v, k >$
max $k \leftarrow A(G, u, v)$;
if $k > \max k$ then return 0 else return 1
Since $A$ runs in polynomial time and $A'$ simply calls $A$ and does some extra
constant number of steps, then $A'$ runs in polynomial time.

($\Leftarrow$)
Assume that \texttt{LongestPath} can be decided in polynomial time; let \( A' \) be a polynomial time algorithm that decides \texttt{LongestPath}. We will provide an algorithm \( A \) that solves \texttt{LongestPathLength}.

\textbf{Algorithm A:}

\begin{itemize}
  \item \textbf{Input:} \(<G, u, v>\)
  \item \textbf{Output:} The size of the longest path between \( u \) and \( v \) or \(-1\)
  \item if there is no path between them
    \begin{enumerate}
      \item \( k \leftarrow n \) // Number of vertices in \( G \)
      \item while \((k \geq 0) \) and \((A'(G, u, v, k) = 0)\) do
        \begin{enumerate}
          \item \( k \leftarrow k - 1 \)
        \end{enumerate}
      \end{enumerate}
      \item return \( k \);
    \end{enumerate}
\end{itemize}

At most \( n + 2 \) calls of \( A' \) are performed, plus a polynomial number of steps.
Since \( A' \) runs in polynomial time, \( A \) runs in polynomial time.

\section{Proving NP-completeness}

Algorithm \( B \) decides \texttt{ShortestPath} (there are many algorithms possible):

\textbf{Algorithm B:}

\begin{itemize}
  \item \textbf{Input:} \(<G, u, v, k>\)
  \item \textbf{Output:} 1/0, depending on whether there exists a simple path
    \begin{itemize}
      \item from \( u \) to \( v \) in \( G \) having at most \( k \) edges
    \end{itemize}
  \item Let \( V \) be the vertex set and \( E \) be the edge set of \( G \). Let \( n = |V| \).
  \item Let \( M \) be an array on the vertex set.
    \begin{enumerate}
      \item Initialize \( M \) with 0 in all positions
      \item \( M[u] \leftarrow 1 \)
      \item \( \text{count} \leftarrow 0 \)
      \item while \(((\text{count} < k) \) and \((\text{count} < n) \) and \((M[v] = 0))\) do
        \begin{enumerate}
          \item for all \((x \in V)\) do
            \begin{enumerate}
              \item if \((M[x] = 1)\) then
                \begin{enumerate}
                  \item for all \((y \in V \text{ with } \{x, y\} \in E)\) do
                    \begin{enumerate}
                      \item \(M[y] = 1\)
                    \end{enumerate}
                \end{enumerate}
            \end{enumerate}
          \end{enumerate}
        \end{enumerate}
      \item \( \text{count} \leftarrow \text{count} + 1 \)
    \end{enumerate}
  \end{itemize}
  \item \text{if} \((M[v] = 1)\) then return 1 else return 0
\end{itemize}

\textbf{Correctness:} It is easy to see that at the end of the while loop, all vertices
at distance \textbf{count} of \( u \) are marked. We run the loop at most until \textbf{count} = \( k \). Thus, the algorithm only returns 1 if \( v \) was reached within distance \( k \), and correctly decides \textsc{ShortestPath}.

\textbf{Polynomial time:} The running time will depend on the data structure used to represent \( G \). Assume an adjacency matrix is used to store \( G \). Let \( n = |V| \) and \( m = |E| \).

- Initialization of \( M \): \( O(n) \)
- \textbf{while} loop runs at most \( n \) times. Its embedded \textbf{for} loop runs \( n \) times, and the other embedded \textbf{for} loop runs at most \( n \) times. Thus, the \textbf{while} loop runs in \( O(n^3) \).
- The final test runs in \( O(1) \).

So the total running time is \( O(n^3) \), which is polynomial in the length of the input.

Note: If one applies Dijkstra’s algorithm for \textsc{ShortestPath}, this algorithm could be done in \( O(n \log n) \).

\textbf{b.} We show that \textsc{LongestPath} is \textsc{NP}-complete.

\textbf{Step 1:} \textsc{LongestPath} \( \in \text{NP} \)

1. An algorithm \( A(x, y) \) to verify \textsc{LongestPath} is described here.

   - The certificate \( y \) for the algorithm will be a sequence of vertices in the graph.
   - The verifier will check that the vertex sequence \( y \) satisfies the following conditions:
     - \( y \) has at least \( k \) vertices
     - \( y \) has no repeated vertices (since the path must be simple)
     - the first vertex of \( y \) is \( u \) and the last one is \( v \)
     - there is an edge connecting each pair of contiguous vertices in the sequence \( y \)

   The verifier returns 1 if and only if all of these conditions are satisfied.

2. This algorithm is a verifier:
• If \( < G, u, v, k > \) \( \in \text{LONGESTPath} \), then the longest simple path corresponds to a sequence of vertices \( y \) with no more than \( n \) vertices (so the certificate is short). For this \( y \), \( A(x, y) = 1 \) since the longest path has length at least \( k \).

• If \( < G, u, v, k > \not\in \text{LONGESTPath} \), then there exists no vertex sequence corresponding to a simple path with length at least \( k \). So for any \( y \), one of the conditions checked by the algorithm will be violated, and this \( A(x, y) = 0 \).

3. The algorithm runs in polynomial time, since the conditions can easily be checked in polynomial time.

**Step 2:** We will show that
\[ \text{HAMPath} \leq_p \text{LONGESTPath} \]

**Step 3:** We will describe a polynomial time algorithm \( F \) that computes a function \( f : \{0,1\}^* \rightarrow \{0,1\}^* \) such that

\[ < G, u, v > \in \text{HAMPath} \text{ if and only if } f(< G, u, v >) \in \text{LONGESTPath} \]

**Algorithm \( F \):**

Input: \( < G, u, v > \)

Let \( n \) be the number of vertices in \( G \)

return \( < G, u, v, n > \)

**Step 4:** We now prove that \( < G, u, v > \in \text{HAMPath} \text{ if and only if } f(< G, u, v >) = < G, u, v, n > \in \text{LONGESTPath} \).

Note that a hamiltonian path from \( u \) to \( v \) is a path passing through every vertex of \( G \) exactly once. So, a hamiltonian path from \( u \) to \( v \) is a simple path from \( u \) to \( v \) with \( n \) vertices (and vice versa).

If \( < G, u, v > \in \text{HAMPath} \) then there exists a hamiltonian path \( P \) from \( u \) to \( v \) with \( n \) vertices. So \( P \) is a simple path with \( n \) vertices, which implies \( < G, u, v, n > \in \text{LONGESTPath} \).

If \( < G, u, v > \not\in \text{HAMPath} \) then there exists no hamiltonian path from \( u \) to \( v \) in \( G \). So, every simple path from \( u \) to \( v \) in \( G \) has at most \( n - 1 \) vertices. So, \( < G, u, v, n > \not\in \text{LONGESTPath} \).

**Step 5:** We show that algorithm \( F \) runs in polynomial time.
The algorithm does nothing but reading the input and outputting what was on the input and the number \( n \) which is smaller than the input size. So the algorithm runs in linear time on the input size.