Problem 1

Solution: Let $M_L$ be the Turing machine that recognizes $L$. This means that on every $w \in L$, $M_L$ accepts, and on every $x \notin L$, $M_L$ either rejects or never halts.

Note that $\Sigma^*$ is a countable set. Let $x_1, x_2, x_3, \ldots$ denote an ordering of all strings in $\Sigma^*$. For example, one can order strings in increasing order of length, and strings with the same length can be ordered lexicographically.

Note also that the set $\mathbb{N} \times \mathbb{N}$ is countable (where $\mathbb{N}$ is the set of natural numbers). Let $(i_1, j_1), (i_2, j_2), (i_3, j_3), \ldots$ denote an ordering of $\mathbb{N} \times \mathbb{N}$. For example, one can order the pairs in increasing order of the sum of two co-ordinates, and pairs with the same sum can be ordered in increasing order of the first co-ordinate.

Define the required machine $M$ as follows:

For $k = 1, 2, 3, \ldots$ do:

- Let $(i_k, j_k)$ be the $k^{th}$ pair in the ordering of $\mathbb{N} \times \mathbb{N}$.
- Simulate the machine $M_L$ on string $x_{i_k}$ for $j_k$ steps.
- If $M_L$ accepts, then print the string $x_{i_k}$ on the output tape, and print the symbol #.

Clearly, $M$ prints only those strings that are accepted by $M_L$, i.e. the strings in $L$. On the other hand, for any $w \in L$, $w$ is accepted by $M_L$ in (say) $t$ steps. Suppose $w = x_i$ in the ordering of $\Sigma^*$. When the machine $M$ works on the pair $(i, t)$ (it will, eventually), it prints $x_i$ on the output tape.

Problem 2

Solution: It is clear that Set-Cover $\in \text{NP}$, as an NTM can decide whether $\langle S = \{S_1, \ldots, S_m\}, k \rangle \in \text{Set-Cover}$ by nondeterministically guessing a subcollection $\{S_{i_1}, \ldots, S_{i_k}\}$ of size $k$, and verifying whether $\bigcup_{j=1}^k S_{i_j} = \bigcup_{j=1}^m S_j$.

To show that Set-Cover is $\text{NP-Complete}$, we give a polynomial-time reduction from Vertex-Cover to Set-Cover, as follows:

On input a Vertex-Cover instance $\langle G = (V, E), k \rangle$:

1. Let $U = E$, that is, the universe $U$ is the set of edges in $G$.

2. For each vertex $v \in V$ in $G$, define $S_v = \{(u, v) : (u, v) \in E\}$. That is, $S_v$ is the set of all edges incident with $v$.

3. Let $S = \{S_v : v \in V\}$. That is, the collection $S$ consists of $S_v$ for every vertex $v \in V$.

4. Output $\langle S, k \rangle$. 


Clearly the reduction takes polynomial time. We now show that the reduction is correct, that is, 
\((G, k) \in \text{Vertex-Cover}\) if and only if \((S, k) \in \text{Set-Cover}\).

If \(\{v_1, \ldots, v_k\}\) is a vertex cover in \(G\), then \(\bigcup_{i=1}^{k} S_{v_i} = E = U\), and thus \(\{S_{v_1}, \ldots, S_{v_k}\}\) is a set cover in \(S = \{S_v : v \in V\}\). Conversely, if \(\{S_{v_1}, \ldots, S_{v_k}\}\) is a set cover in \(S\), then \(\bigcup_{i=1}^{k} S_{v_i} = E = U\), and thus \(\{v_1, \ldots, v_k\}\) is a vertex cover in \(G\).

We therefore conclude that Set-Cover is \textbf{NP-Complete}.

\textbf{Problem 3}

\textbf{Solution to Part 1:} Suppose that \(P = NP\). Then there is a polynomial-time algorithm \(A\) that decides 3-SAT. We now describe an algorithm \(B\) that actually finds a satisfying solution to any given 3-SAT instance \(\varphi\) that is satisfiable by invoking algorithm \(A\) \(n\) times, where \(n\) is the number of variables in \(\varphi\). Therefore, if \(A\) runs in polynomial-time, then \(B\) runs in polynomial-time.
**Algorithm B:**  
On input $\varphi(x_1, \ldots, x_n)$:

1. Run algorithm $A$ on $\varphi$ to decide whether $\varphi$ is satisfiable. If not, then output NO and halt.  
   If $\varphi$ is satisfiable, then the rest of the algorithm finds a satisfying assignment in $n$ iterations, as follows.

2. Define formulas $\varphi_0(x_2, \ldots, x_n) = \varphi(0, x_2, \ldots, x_n)$ and $\varphi_1(x_2, \ldots, x_n) = \varphi(1, x_2, \ldots, x_n)$. That is, $\varphi_0$ and $\varphi_1$ are the resulting formulas after $x_1$ is substituted by constants 0 and 1 respectively. If $\varphi$ is satisfiable, then clearly at least one of $\varphi_0$ and $\varphi_1$ must be satisfiable, as in any satisfying assignment $x_1$ is assigned either 0 or 1. Thus, in the first iteration, first run algorithm $A$ on $\varphi_0$ to decide whether $\varphi_0$ is satisfiable, and if so, set $a_1 = 0$; else $\varphi_1$ must be satisfiable, and set $a_1 = 1$. Assign $x_1 = a_1$, and repeat the above for $\varphi_{a_1}$ until all variables have been assigned. That is:

3. In general, in the $i$-th iteration, with $a_1, \ldots, a_{i-1}$ already assigned to $x_1, \ldots, x_{i-1}$ in the first $i-1$ iterations so that $\varphi_{a_1, \ldots, a_{i-1}}(x_i, \ldots, x_n) = \varphi(a_1, \ldots, a_{i-1}, x_i, \ldots, x_n)$ is satisfiable, set
   
   $\varphi_{a_1, \ldots, a_{i-1}, 0}(x_{i+1}, \ldots, x_n) = \varphi(a_1, \ldots, a_{i-1}, 0, x_{i+1}, \ldots, x_n),$
   
   and
   
   $\varphi_{a_1, \ldots, a_{i-1}, 1}(x_{i+1}, \ldots, x_n) = \varphi(a_1, \ldots, a_{i-1}, 1, x_{i+1}, \ldots, x_n).$

   Then as above, at least one of $\varphi_{a_1, \ldots, a_{i-1}, 0}$ and $\varphi_{a_1, \ldots, a_{i-1}, 1}$ must be satisfiable. Thus, first run algorithm $A$ on $\varphi_{a_1, \ldots, a_{i-1}, 0}$ to decide whether it is decidable, and if so, set $a_i = 0$; else $\varphi_{a_1, \ldots, a_{i-1}, 1}$ must be satisfiable, and set $a_i = 1$.

4. Repeat the above process until all variables $x_1, \ldots, x_n$ have been assigned, and output the assignment $x_1 = a_1, \ldots, x_n = a_n$.

If $\varphi$ is not satisfiable, then algorithm $B$ outputs NO at the beginning. If $\varphi$ is satisfiable, then the assignment $x_1 = a_1, \ldots, x_n = a_n$ found by $B$ satisfies $\varphi$ as explained in the description of algorithm $B$. The claimed polynomial running time of $B$ can be easily verified.

**Solution to Part 2:** Define the language

$$\text{MAX-3-SAT} = \{\langle \varphi, k \rangle : \varphi \text{ is in 3-CNF and } \exists \text{ an assignment that satisfies } k \text{ clauses of } \varphi \}.$$

Clearly MAX-3-SAT $\in \text{NP}$, as an NTM can decide whether $\langle \varphi, k \rangle$ $\in$ MAX-3-SAT by nondeterministically guessing an assignment and verifying whether it satisfies $k$ clauses of $\varphi$. Therefore if $\textbf{P} = \textbf{NP}$, then there is a polynomial-time algorithm $C$ that decides MAX-3-SAT. We now construct the following algorithm $D$ that finds an assignment that satisfies the maximum number of clauses in a given $\varphi$ using this algorithm $C$. Algorithm $D$ uses essentially the same technique as algorithm $B$ does.
Algorithm $D$: 

On input $\varphi(x_1, \ldots, x_n) = C_1 \land \cdots \land C_m$, where $m$ is the number of clauses in $\varphi$:

For $k = m$ downto 0:

1. If $k = 0$, then output any assignment and halt. Else,

2. Run algorithm $C$ on $\langle \varphi, k \rangle$ to decide whether there is an assignment that satisfies $k$ clauses of $\varphi$. If $C$ outputs NO, then go to the next iteration. Else (if $C$ outputs YES), we find such an assignment as follows:

3. Set $\varphi_0(x_2, \ldots, x_n) = \varphi(0, x_2, \ldots, x_n)$ and $\varphi_1(x_2, \ldots, x_n) = \varphi(1, x_2, \ldots, x_n)$ as in algorithm $B$. Then at least one of $\varphi_0$ and $\varphi_1$ has an assignment that satisfies at least $k$ clauses. Thus first run algorithm $C$ on $\langle \varphi_0, k \rangle$, and if $C$ accepts, set $a_1 = 0$; else set $a_1 = 1$. Repeat this for $\varphi_{a_1}$ in a way similar to algorithm $B$, until all variables have been assigned.

4. Output $x_1 = a_1, \ldots, x_n = a_n$ and halt.

It is not hard to see that algorithm $D$ finds an assignment that satisfies the maximum number of clauses of a given formula $\varphi$, and it takes polynomial time provided that $C$ runs in polynomial time.

Problem 4

Solution: We show that Subset-Sum is a special case of Knapsack. Consider special instances of Knapsack where the volumes and costs are the same, i.e. $v_i = c_i \forall i$, and the volume bound equals the target cost, i.e. $B = t$. The Knapsack problem asks whether there exists a set $S \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in S} c_i \geq t \text{ and } \sum_{i \in S} v_i \leq B \quad (1)$$

which is same as asking whether there exists $S$ such that

$$\sum_{i \in S} v_i \geq t \text{ and } \sum_{i \in S} v_i \leq t$$

which is same as asking whether there exists $S$ such that

$$\sum_{i \in S} v_i = t$$

which is an instance of Subset-Sum.

Therefore, since Subset-Sum is a NP-hard problem, so is Knapsack. On the other hand, Knapsack is in NP (guess the set $S$ and verify whether Condition (??) is satisfied). Hence Knapsack is NP-complete.

Problem 5