1. Prove that the class P is closed under (a) intersection and (b) concatenation.
For a) show if L1, L2 ∈ P then L1 ∩ L2 ∈ P. For b) show that if L1, L2 ∈ P then L1 ◦ L2 ∈ P, where L1 ◦ L2 = {x1x2 | x1 ∈ L1, x2 ∈ L2}.

ANS:

- **Intersection**. Let $L_1, L_2 \in P$. We want to show that $L_1 \cap L_2 \in P$. Because $L_1 \in P$ then there exists a TM M_1 with time complexity $O(n^{k_1})$ for some constant k_1. Because $L_2 \in P$ then there exists a TM M_2 with time complexity $O(n^{k_2})$ for some constant k_2. We construct a decider M with polynomial time complexity deciding $L_1 \cap L_2$:

```
M = "On input x:
    1. Run M1 on input x
    2. If M1 accepted, then run M2 on input x, else reject.
    3. If M2 also accepted, then accept, else reject."
```

In the worst case M will run both M_1 and M_2, in which case it uses $O(n^{k_1}) + O(n^{k_2})$ steps. Let $k = \max(k_1, k_2)$. We then see that M has time complexity $O(n^k)$ and hence $L(M) = L_1 \cap L_2 \in P$. (Note that we omitted the details where a copy of the string x for the machine M_2 is stored while the machine M_1 is computing; this can be done e.g. on a second tape but we know that all deterministic variants of Turing machines are polynomial-time equivalent.

• **Complement.** The same construction as for decidable languages (see e.g. slide 13 in Lecture 3). We simply swap the accept and reject state. The running time of the modified machine does not change.

- **Concatenation**. We want to show that if $L_1, L_2 \in P$ then $L_1 \circ L_2 \in P$. Assume so that $L_1 \in P$ and that $L_2 \in P$. By definition, this means that there exist deciders M_1 and M_2 such that M_1 is a decider for L_1 with time complexity $O(n^{k_1})$ and M_2 is a decider for L_2 with time complexity $O(n^{k_2})$ for some constants k_1 and k_2.

The concatenation $L_1 \circ L_2$ is defined as

$$L_1 \circ L_2 = \{x_1 x_2 \mid x_1 \in L_1, x_2 \in L_2\}$$

The decider for $L_1 \circ L_2$ must, given an input x, try to find a partition of x into $x_1 x_2$ such that $x_1 \in L_1$ and $x_2 \in L_2$. Here is the decider:

```
"On input $x = a_1 \dots a_n$
  1. For $i = 0$ to $n$ do
       i. Let $x_1 = a_1 \dots a_i$ and $x_2 = a_{i+1} \dots a_n$. (By agreement $a_1 \dots a_0 =$ and $a_{n+1} \dots a_n =$ ).
       ii. Run $M_1$ on the input $x_1$.
       iii. Run $M_2$ on the input $x_2$.
       iv. If both $M_1$ and $M_2$ accepted, then accept
  2. If no choice of $x_1$ and $x_2$ led to acceptance, then reject"
```

We must now show that the decider has polynomial time complexity. The main loop of the decider is traversed at most (n+1)-times. If we run M_1 on a substring of x, this will take at most $O(n^{k_1})$ steps. Similarly, running M_2 on a substring of x will take at most $O(n^{k_2})$ steps. Consequently, a single traversal of the loop body uses no more than $O(n^{k_1}) + O(n^{k_2}) = O(n^k)$ steps, where $k = \max(k_1, k_2)$. The whole decider thus uses $(n + 1) \cdot O(n^k) = O(n^{k+1})$ steps. Hence $L(M) = L_1 \circ L_2 \in P$.

2. Every week someone manages to "prove" that P = NP or that P ≠ NP. Last week a very famous professor published the following proof that P ≠ NP using the language HAM-PATH = { <G,u,v> : there is a Hamiltonian path from u to v in graph G}.
Proof:
Consider the following decider for HAM-PATH:
"On input <G, s, t>:
1. Generate all possible permutations of nodes from G.
2. If one of these permutations (sequences of nodes) forms a Hamiltonian path, then accept.
3. Otherwise reject."

Because there are n! different permutations of nodes to examine, the algorithm clearly does not run in polynomial time. Therefore we have proved that HAM-PATH has exponential time complexity and this means HAM-PATH ≠ P. Because we know that HAM-PATH ∈ NP, we conclude that P ≠ NP.

Describe the error in the above proof.

ANS:

It is true that the suggested algorithm for HAMPATH does not run in polynomial time, but from this fact we cannot conclude that the language HAMPATH does not belong to the class P. There can still be other (faster) algorithms for HAMPATH that run in polynomial time; we simply did not exclude this possibility by presenting one particular algorithm with an exponential running time. To conclude that HAMPATH $\notin$ P we would have to show no algorithm for HAMPATH has a polynomial time complexity.

3. Show that the HAM-PATH problem is NP-complete. (You may assume that you know that HAM-CYCLE is NP-complete.)

ANS:

Again, observe that given a sequence of vertices it is easy to check in polynomial time if the sequence is a hamiltonian path, and thus the problem is in NP.

We reduce from the hamiltonian cycle problem. Let $G = (V, E)$ be a graph. The reduction transforms graph G into G' as follows. We pick an arbitrary vertex $v \in V$ and add a new vertex v' that is connected to all the neighbors of v. We also add new vertices u and u' so that u is adjacent to v and u' is adjacent to v'. This reduction clearly takes a polynomial time.

To complete the proof, we have to show that G has a hamiltonian cycle if and only if G' has a hamiltonian path. Now if there is a hamiltonian cycle $(v, v_2, \ldots, v_n, v)$ in G, then $(u, v, v_2, \ldots, v_n, v', u')$ is a hamiltonian path in G'. On the other hand, if there is a hamiltonian path in G' , its endpoints have to be u and u', because these have only on neighbor and thus cannot be in a middle of the path. Thus, the path has form $(u, v, v_2, \ldots, v_n, v', u')$ and we have that $(v, v_2, \ldots, v_n, v)$ is a hamiltonian cycle in G.

4. Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

ANS:

Let k be the number of subroutines. The algorithm starts out with an input data of size n. Each subroutine takes (some of) the available data as an input, performs some steps, then returns some amount of data as an output. Every time a subroutine returns, the output accumulates the amount of data the algorithm has access to. Any or all of this data may then be given as an input to the next subroutine. Since each subroutine runs in polynomial time, the output must also have a size polynomial in the size of the input. Let d be an upper bound on the degree of the polynomials. Then there is a function $p(n) = n^d + c$, where c is a constant, such that p(n) is an upper bound for the size of the output of any subroutine when given an input of size n.

Let $n_0 = n$ and $n_i = n_{i-1} + p(n_{i-1})$ for all $1 \le i \le k$. We show by induction that n_i is an upper bound for the amount of data available to the algorithm after the ith subroutine call. The base case is trivial. Assume the claim holds for $i - 1$ and let n^0 be the exact amount of data available before the ith call. Then we have $n_i \leq n' + p(n')$, since the ith call accumulates the amount of the data by at most $p(n')$. Since p is increasing, by assumption we have $n' \leq n_{i-1}$ and $p(n') \leq p(n_{i-1})$, from which the claim follows.

We use induction again to show that each n_i is polynomial in n. The base case is again trivial. Assume n_{i-1} is polynomial in n. Since the composition of two polynomials is also polynomial, we have that $p(n_{i-1})$ is polynomial in n. Since also the sum of two polynomials is polynomial, we have that $n_{i-1} + p(n_{i-1}) = n_i$ is polynomial in n. Therefore n_k, which is an upper bound for the amount of data after the final subroutine, is also polynomial in n, and the time must also be polynomial.

For the second part, observe that if we have a subroutine whose output is always twice the size of its input, and we call this subroutine n times, starting with input of size 1 and always feeding the previous output back into the subroutine, the final output will have size 2^n. This means that the algorithm will take exponential time.