```
def minimizeFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
      = {(q, a): r for (q, a, r) in fsa.R}
    dist = {(q, r) for q in fsa.Q for r in fsa.Q if q != r and (q in fsa.F) != (r in fsa.F)]
    done = False
    while not done:
        done = True #; print(dist)
        for q in fsa.Q:
            for r in fsa.Q:
                if q != r and (q, r) not in dist and any(((q, u) in  ) != ((r, u) in  ) or \
                    ((q, u) in  ) and (( [(q, u)],  [(r, u)]) in dist) for u in fsa.T):
                    dist |= {(q, r)}; done = False #; print('adding', q, r)
    Q  = {set({q} | {r for r in fsa.Q if (q, r) not in dist}) for q in fsa.Q}
    R  = {(q , u, r ) for q  in Q  for r  in Q  for u in fsa.T if any((q, u, r) in fsa.R for q
    q 0 = {q  for q  in Q  if fsa.q0 in q }.pop()
    F  = {q  for q  in Q  if (q  & fsa.F) != set()}
    return FiniteStateAutomaton(fsa.T, Q , R , q 0, F )
```

This code implements a function `minimizeFSA` that takes a finite state automaton (FSA) as input and returns a minimized version of the input FSA.

A finite state automaton (FSA) is a mathematical model of computation that consists of a finite set of states, a set of transitions between states, and a set of final or accepting states.

The code uses a well-known algorithm for minimizing FSAs, which is based on the observation that two states are equivalent if and only if they are indistinguishable with respect to the language recognized by the automaton. The algorithm works by partitioning the set of states into sets of equivalent states and constructing a new automaton with a smaller number of states, where each set of equivalent states is represented by a single state. The algorithm is implemented in the code using a combination of set operations and graph search.

The code returns a new FSA object with a minimized number of states.

The time complexity of the algorithm implemented by this code is $O(n^2)$, where n is the number of states in the input FSA. The algorithm has a nested loop structure that iterates over all pairs of states in the input FSA, which results in a quadratic time complexity.

The algorithm performs a graph search in which it builds a set of equivalent states by repeatedly adding pairs of states to the set. The size of the set of equivalent states is determined by the number of pairs of states that need to be checked and added, which is proportional to the number of states in the input FSA.

Therefore, the algorithm has a time complexity of $O(n^2)$, which means that the running time of the algorithm increases quadratically with the number of states in the input FSA.

```
def parseFSA(fsa: str) -> FiniteStateAutomaton:
```

```
    fsa = [line for line in fsa.split('\n') if line.strip() != '']
    q0 = fsa[0] # first line: initial
    F = set(fsa[1].split()) # second line: final, final, ...
    R = set()
    for line in fsa[2:]: # all subsequent lines: "source symbol → target"
        l, r = line.split('→')
        R |= {(l.split()[0], l.split()[1], r.split()[0])}
    T = {r[1] for r in R}
    Q = {q0} | F | {r[0] for r in R} | {r[2] for r in R}
    return FiniteStateAutomaton(T, Q, R, q0, F)
```

This code defines a function `parseFSA` that takes a string representation of a finite state automaton (FSA) as input and returns a corresponding `FiniteStateAutomaton` object.

The function first splits the input string into lines and removes any empty lines. The first line of the input string is assumed to represent the initial state of the FSA and is stored as the variable `q0`. The second line of the input string is assumed to represent the final states of the FSA, which are stored as a set of strings `F`.

The subsequent lines of the input string are assumed to represent the transitions in the FSA, and are processed one by one. Each line is split into two parts, with the first part representing the source state and symbol and the second part representing the target state. The source state, symbol, and target state are extracted from the line and added as a tuple to the set of transitions `R`.

The set of symbols `T` is then constructed from the symbols used in the transitions. The set of states `Q` is constructed from the initial state, final states, and source and target states used in the transitions.

Finally, the function returns a `FiniteStateAutomaton` object constructed from the sets of symbols, states, transitions, initial state, and final states.

The time complexity of the `parseFSA` function is $O(n)$, where n is the number of transitions in the input string representation of the finite state automaton (FSA).

The function processes each line of the input string representation one by one and adds the extracted source state, symbol, and target state as a tuple to the set of transitions `R`. The set `R` has a time complexity of $O(1)$ for adding a new element. Therefore, the overall time complexity for processing all the transitions is $O(n)$, where n is the number of transitions in the input string representation.

The construction of the sets of symbols `T` and states `Q` from the transitions has a time complexity of O(n), as each transition is processed once. The construction of the `FiniteStateAutomaton` object has a time complexity of $O(1)$.

Therefore, the overall time complexity of the `parseFSA` function is $O(n)$.

```
def equivalentFSA(a: FiniteStateAutomaton, a : FiniteStateAutomaton, printMap = False) -> bo
    a = minimizeFSA(totalFSA(a))
    a  = minimizeFSA(totalFSA(a ))
      = {(q, u): r for (q, u, r) in a.R}
       = {(q, u): r for (q, u, r) in a .R}
    m, v = {a.q0: a .q0}, {a.q0}
    while v != set():
        if printMap: print(m)
        q = v.pop(); q  = m[q]
        for u in a.T:
            if ((q, u) in  ) != ((q , u) in  ): return False
            elif (q, u) in  : # (q , u) in
                r, r  =  [(q, u)],  [(q , u)]
                if r in m:
                    if m[r] != r : return False
                elif r  in m.values(): return False
                else: v.add(r); m[r] = r
    if printMap: print(m)
    return a .F == {m[q] for q in a.F}
```

This code determines if two finite state automatons (FSA) are equivalent.

The function first minimizes both input FSAs by calling `minimizeFSA` and `totalFSA` functions, which reduce the number of states and make the automatons deterministic respectively.

Then it creates a dictionary of transition functions for each FSA and checks if the transition functions are equivalent.

The algorithm uses a BFS-like approach to traverse the states and build a mapping from the states in the first FSA to the states in the second FSA. The algorithm returns False if it finds a state in one FSA that has a transition for a symbol where the corresponding state in the second FSA does not. The algorithm also returns False if it finds two states in the first FSA that map to the same state in the second FSA.

Finally, the function compares the sets of final states of both minimized FSAs and returns True if they are equivalent, False otherwise. The time complexity of this function is $O(|Q| * |\Sigma|)$, where Q is the number of states and Σ is the size of the alphabet.

The time complexity of the "equivalentFSA" function is $O(n^2)$, where n is the number of states in the finite state automata.

The bulk of the time spent by the function is on the while loop, which continues until the set "v" is empty. On each iteration of the loop, the code performs a constant amount of work (checking if a transition exists in both automata and checking if there is a conflict in the mapping). The size of the set "v" decreases by 1 on each iteration of the loop, so the while loop will run a maximum of n

times, where n is the number of states in the finite state automaton.

Additionally, the first two lines of the function call "minimizeFSA" on both automata, which has a time complexity of $O(n^2)$, so this contributes to the overall time complexity as well.

Therefore, the overall time complexity of the "equivalentFSA" function is $O(n^2)$.

```
def totalFSA(A: FiniteStateAutomaton, t = -1) -> FiniteStateAutomaton:
    T = set('abcdefghijklmnopqrstuvwxyz') # T is vocabulary, t is trap state
    R = A.R | {(q, a, t) for q in A.Q for a in T if all((q, a, r) not in A.R for r in A.Q)}
    if any(r == t for (q, a, r) in R): # transition to t exists
        Q = A.Q | {t}
        R = R | {(t, a, t) for a in T}
    else: Q = A.Q
    return FiniteStateAutomaton(T, Q, R, A.q0, A.F)

def renameFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    m, c = {}, 0
    for q in fsa.Q:
        m[q] = c; c = c + 1
    Q  = {i for i in range(c)}
    R  = {(m[q], u, m[r]) for (q, u, r) in fsa.R}
    q0 = m[fsa.q0]
    F  = {m[q] for q in fsa.F}
    return FiniteStateAutomaton(fsa.T, Q , R , q0, F )
```

This code defines two functions: `totalFSA` and `renameFSA` that work with finite state automata (FSA) represented as `FiniteStateAutomaton` objects.

The `totalFSA` function takes as input an FSA `A` and an optional parameter `t`. The function returns an FSA with the same vocabulary `T` as `A` (in this case, T is set to the alphabet of lowercase letters), but with some additional states and transitions. If no transition exists from any state `q` in `A` to any other state for some letter `a` in the vocabulary, then a transition from `q` to a trap state `t` for `a` is added. If the trap state `t` already exists in the transitions, the function does not add it to the set of states `Q`, but does add transitions from `t` to `t` for all letters in the vocabulary.

The `renameFSA` function takes as input an FSA `fsa` and returns a new FSA with the same transitions but with the states renamed to be integers starting from 0. This is done by creating a mapping `m` from the original states to integers, and then creating new sets of states `Q` , transitions `R` , initial state `q0`, and final states `F` based on this mapping.

The time complexity of the `totalFSA` function is `O(|Q| * |T|)`, where `|Q|` is the number of states in the finite state automaton (FSA) and `|T|` is the size of the vocabulary. This is because for each state in the FSA and for each symbol in the vocabulary, the function checks whether there is a transition for that state

and symbol, and if not, it adds a transition to the trap state.

The time complexity of the `renameFSA` function is `O(|Q|)`. This is because the function performs a single iteration over all states of the FSA, renaming them with consecutive integers, and storing the mapping in a dictionary. The calculation of the renamed transition relations and final states is also a single iteration over the original transition relations and final states, respectively.