```python
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 )
```

The `minimizeFSA` function takes a `FiniteStateAutomaton` object `fsa` as input and returns a new `FiniteStateAutomaton` object that represents a minimized version of the input automaton. The function works by iteratively merging pairs of states in the input automaton until no more merges can be performed.

The algorithm used for the minimization is based on the notion of distinguishing sequences. Two states `q` and `r` in the automaton are said to be distinguishable if there exists a string `u` such that the transitions from `q` and `r` on `u` lead to different sets of states. The algorithm starts by initializing a set `dist` with all pairs of states that are distinguishable (i.e., one of the states is accepting and the other is not, or vice versa).

The algorithm then iterates over all pairs of states in the automaton and checks if they are not already in `dist` and if there exists a string `u` such that the transitions from `q` and `r` on `u` lead to distinguishable sets of states. If this is the case, the pair `(q, r)` is added to `dist`. The loop over all pairs of states is repeated until no more pairs can be added to `dist`.

Once `dist` is finalized, the algorithm constructs a new set of states `Q'` by merging each pair of states in the automaton that are not in `dist`. Each element in `Q'` is a set of states from the original automaton that have been merged together. The transitions in the minimized automaton are constructed by considering all transitions in the original automaton and mapping the source and target states to the appropriate sets in `Q'`. The initial state and set of accepting states in the minimized automaton are obtained by finding the set in `Q'` that contains the initial state and all sets that contain at least one accepting state, respectively.

Overall, the `minimizeFSA` function implements a well-known algorithm for minimizing finite state automata, called the Hopcroft algorithm. This algorithm has a worst-case time complexity of O(n log n), where n is the number of states in the automaton. However, the actual running time of the algorithm depends on the structure of the input automaton and can be much faster in practice.

The `minimizeFSA` function has a time complexity of $O(|Q|^3)$, where $|Q|$ is the number of states in the input automaton, because it uses a variant of the Hopcroft algorithm, which has a time complexity of $O(|Q|^2|T|)$, where $|T|$ is the size of the input alphabet. In this implementation, $|T|$ is assumed to be constant, so the time complexity simplifies to $O(|Q|^2)$.

```python
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)
```

The `parseFSA` function takes a string `fsa` as input, which represents a finite state automaton in a specific text format, and returns a `FiniteStateAutomaton` object that represents the automaton. The function works by splitting the input string into lines, extracting the initial state, final states, and transitions, and constructing a `FiniteStateAutomaton` object from this information.

The first line of the input string is assumed to contain the initial state of the automaton. The second line is assumed to contain a space-separated list of final states. The remaining lines are assumed to represent transitions in the automaton, with each line containing the source state, input symbol, and target state separated by spaces and an arrow symbol (→).

The function first extracts the initial state and final states from the first two lines of the input string. It then iterates over all remaining lines and extracts the source state, input symbol, and target state for each transition. The resulting set of transitions is stored in the variable `R`.

The set of input symbols is computed by extracting the second component of each transition and taking the set of unique values. The set of all states in the automaton is computed by taking the union of the initial state, final states, and the source and target states of each transition.

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

Overall, the `parseFSA` function implements a simple parser for a text-based format for representing finite state automata. This format is not standard, and there are many different ways to represent finite state automata in text form, but the basic idea is to encode the automaton as a list of lines, where each line represents a transition in the automaton. The `parseFSA` function assumes a specific format for the input string, and may fail or produce incorrect results if the input string is not well-formed or does not conform to the assumed format.

The time complexity of `parseFSA` function is O(n), where n is the total number of characters in the input `fsa` string.

This is because the function simply reads each line of the input string once, and then processes each line by splitting it into substrings, performing set unions and creating tuples. These operations all take constant time. The overall time complexity is therefore linear in the size of the input.

```
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 is a Python function `equivalentFSA` that takes two `FiniteStateAutomaton` objects `a` and `a'`, and returns a boolean indicating whether they are equivalent (i.e., they accept the same language). The function first calls the `totalFSA` function on `a` and `a'` (which I'm assuming adds a sink state to each automaton), and then calls the `minimizeFSA` function on each resulting automaton to minimize them.

Next, the function constructs a mapping `m` from states in `a` to states in `a'`. The function initializes `m` with the initial state of `a` and `a'` (i.e., `a.q0` and `a .q0`), and initializes a set `v` with `a.q0`.

The function then enters a loop that continues until `v` is empty. In each iteration of the loop, the function removes a state `q` from `v` and its corresponding state `q ` from `m`. The function then checks whether `q` and `q ` agree on all outgoing transitions (i.e., for each symbol `u` in the alphabet of the automata, the two states have transitions labeled `u` to the same destination states or both have no transition labeled `u`). If `q` and `q ` do not agree on a transition, then the automata are not equivalent and the function returns `False`.

If `q` and `q` agree on all transitions, then the function examines each destination state `r` of `q` and its corresponding destination state `r` of `q` . If `r` has already been assigned a mapping in `m`, then the function checks whether the assigned mapping is to `r` . If not, then the automata are not equivalent and the function returns `False`. If `r` has not yet been assigned a mapping, then the function adds `r` to `v` and assigns `r` to `m[r]`.

Finally, the function checks whether the set of final states of `a'` is the image of the set of final states of `a` under the mapping `m`. If so, then the automata are equivalent and the function returns `True`. Otherwise, the automata are not equivalent and the function returns `False`.

The time complexity of the `equivalentFSA` function depends on the size of the input finite state automata `a` and `a` , as well as on the size of the output of the `minimizeFSA` function and the `totalFSA` function, which in turn depend on the input automata.

The `equivalentFSA` function uses the `minimizeFSA` and `totalFSA` functions, so its time complexity is at least $O(|Q|^2)$ for each input automaton. In addition, the function uses a breadth-first search algorithm to explore the state space of the two automata and check if they are equivalent. The time complexity of the BFS algorithm is $O(|Q|^2|T|)$, where $|T|$ is the size of the input alphabet. Therefore, the overall time complexity of the `equivalentFSA` function is $O(|Q|^3|T|)$.

```python
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}
    q 0 = m[fsa.q0]
    F  = {m[q] for q in fsa.F}
    return FiniteStateAutomaton(fsa.T, Q , R , q 0, F )
```

`totalFSA` is an algorithm that takes a finite state automaton (FSA) `A` as input and returns a new FSA `A'` that is "total". A total FSA is one in which there exists a transition from every state on every input symbol. If the input FSA `A` is already total, then `totalFSA` simply returns `A`.

If `A` is not total, `totalFSA` adds a new "trap" state `t` to `A` and creates transitions from `t` to itself on every input symbol that does not already have a transition defined in `A`. Then `totalFSA` returns the new FSA `A'` that includes the original transitions of `A` as well as the newly added transitions to `t`.

The time complexity of `totalFSA` is O(|T||Q|^2), where |T| is the size of the vocabulary and |Q| is the number of states in the FSA.

`renameFSA` is an algorithm that takes an FSA `fsa` as input and returns a new FSA with its states renamed to the integers 0 to |Q|-1, where |Q| is the number of states in `fsa`. This is done to make it easier to compare FSAs for equivalence, since the state names may not be the same across different FSAs.

The algorithm first creates a dictionary `m` that maps each state in `fsa` to a unique integer. Then it creates a new set of states `Q'` that are simply the integers from 0 to |Q|-1. The transitions in the new FSA `R'` are created by mapping each state in the original FSA `fsa` to its corresponding integer in `Q'`.

The time complexity of `renameFSA` is O(|Q| + |R|), where |Q| is the number of states in `fsa` and |R| is the number of transitions in `fsa`.'

The time complexity of `totalFSA` is O(|Q| * |T|^2), where |Q| is the number of states in the input automaton and |T| is the size of the vocabulary. This is because the function first iterates through all possible transitions (|Q| * |T|^2), and then checks whether a trap state needs to be added, which requires checking each state to see if it has a transition for every symbol in the vocabulary (|Q| * |T|). The overall time complexity is dominated by the first step, so the function runs in O(|Q| * |T|^2) time.

The time complexity of `renameFSA` is O(|Q|^2), where |Q| is the number of states in the input automaton. This is because the function creates a dictionary that maps each state to a new unique identifier, which requires iterating through all states once (|Q|), and then updating each transition in the automaton to use the new identifiers (another |Q| operations). The overall time complexity is dominated by the first step, so the function runs in O(|Q|^2) time.