**Algorithm for flight Path trace**

**Algo for Best Time complexity**

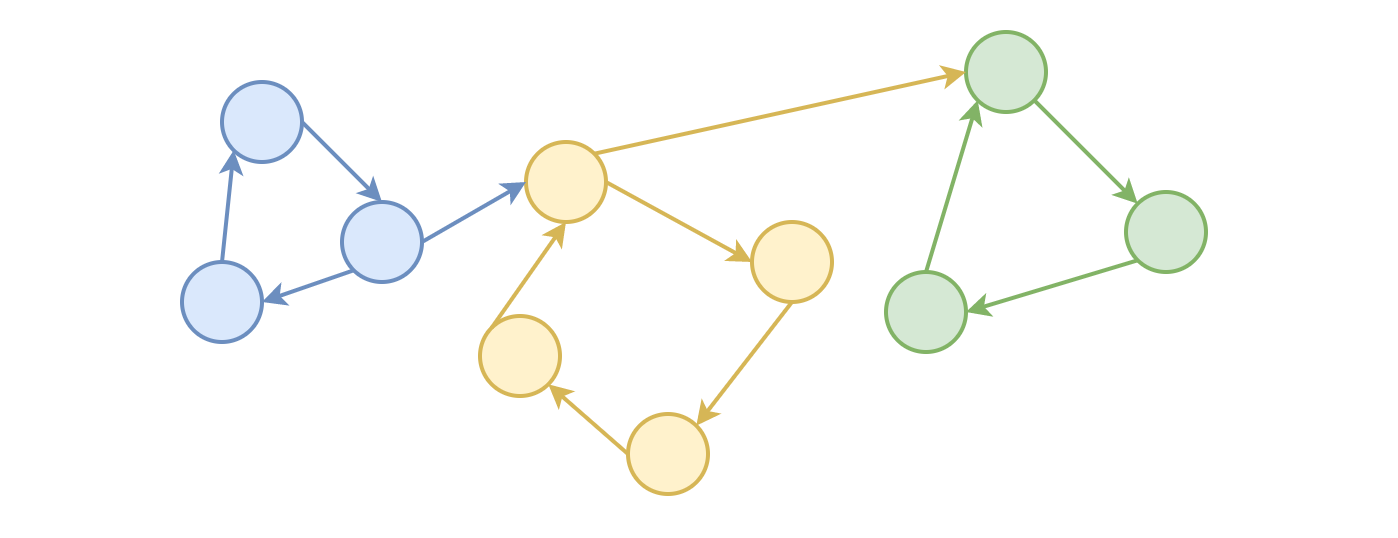
We have an array containing departure and arrival pairs. We assume this problem as a graph, where the vertices of the graph represent airports and edges represent a flight between the airports.

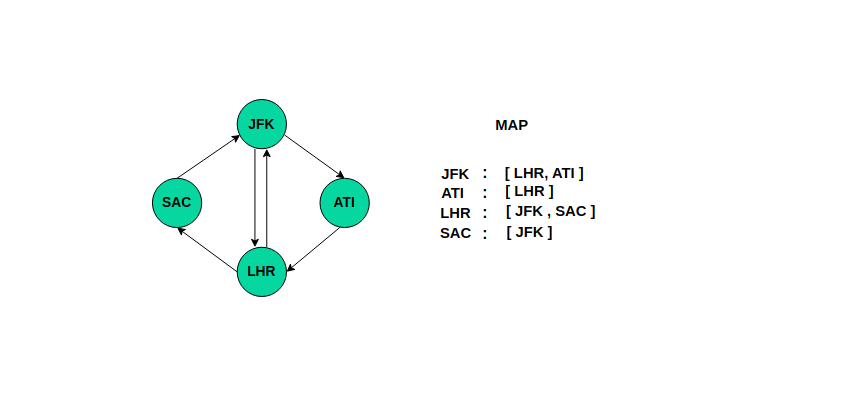
Let’s see an algorithm for the described problem below:

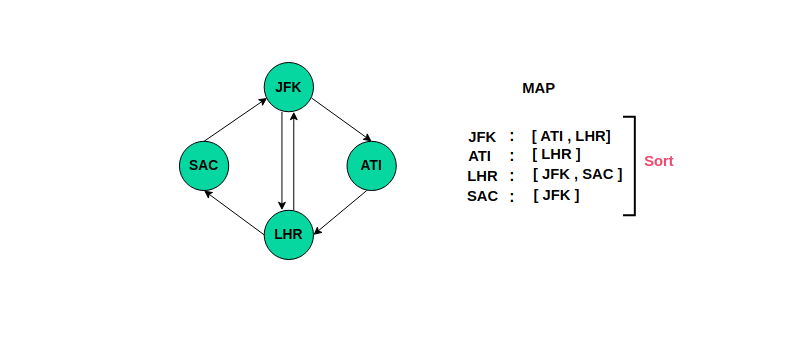
1. First, we create a hash map with keys for each departure point. The value for a given key is an alphabetically sorted list of all airports to which the passenger flew from the airport represented by the given key.
2. Next, we will append the start vertex to a string.
3. Then, we will perform a post-order depth-first search (DFS) from the start vertex where the visit operation comprises appending a visited vertex to a list of strings.
4. Once all vertices have been visited, reverse the list to get the result.

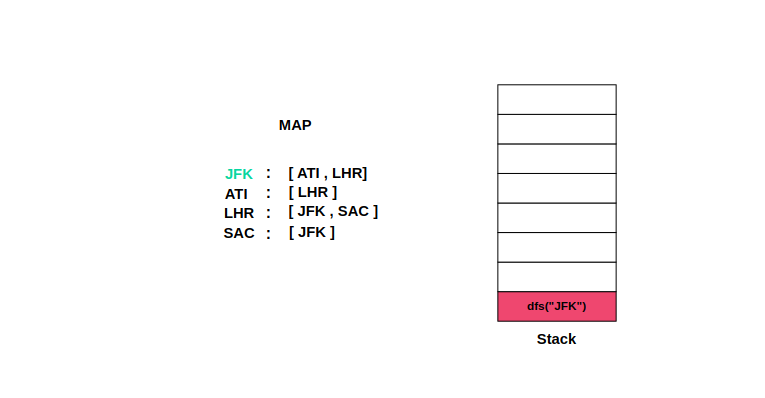
This way, we will get a valid itinerary that visits all the airports. We could assume the algorithm as the postorder depth-first search in a graph, with a fixed start vertex.

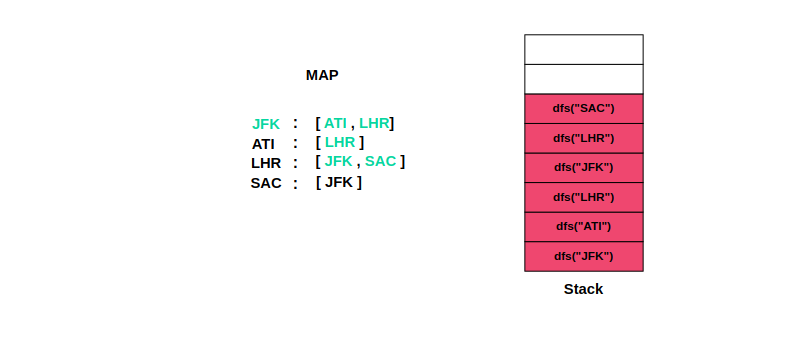
Let’s see the following illustration to understand this algorithm.

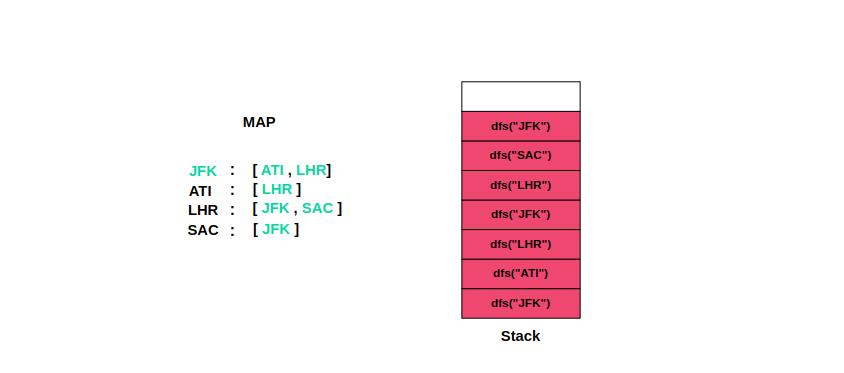
By connecting all the circles in the above process, we build the Eulerian cycle at the end.

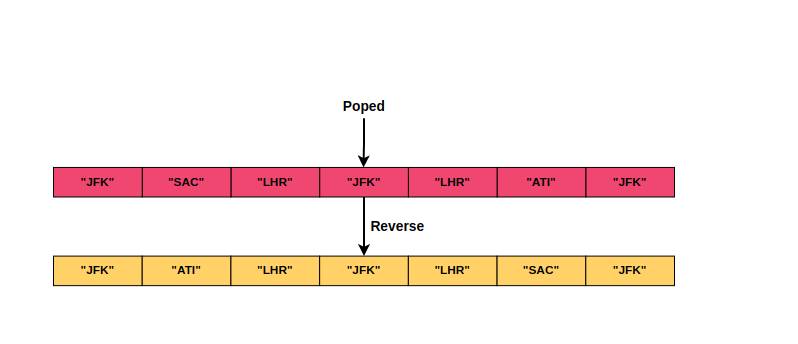
****

****

****

****

****



**Algorithm for Best Memory**

We have an array containing departure and arrival pairs. We assume this problem as a graph, where the vertices of the graph represent airports and edges represent a flight between the airports.

Let’s see an algorithm for the described problem below:

1. First, we sort the tickets slice. Then we create a hash map with keys as departure point and value as slice of airport numeric code and number of edges (number of destination airports from the source).

edges := make(map[string][]int)

In best time algorithm – I am using map with key as string and value as []string as compared to []int above which saves lot of memory.

m := make(map[string][]string, len(tickets)+1)

For map generation as after sorting tickets keys with same source will be consecutive, we can very easily count number of edges from the source, and repeat the process for other sources.

1. Next, during dfs we start from source airport, for each edge we take destination from ticket map.

if e, ok := edges[airport]; ok {

for e[0] < e[1] {

i := e[0]

e[0]++

visit(tickets[i][1], results, head, tickets, edges)

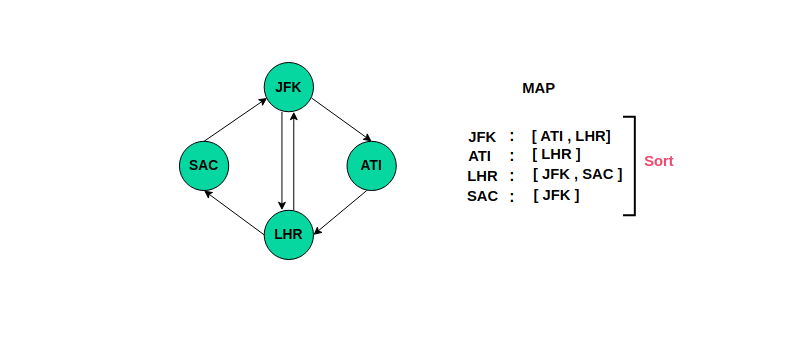
}

}

For each edge visited stack is formed as above, popping from stack occurs when all the edges of all the vertices have been traversed.

7. Once all edges have been visited, route array is fully populated after full execution of dfs function, flight path is returned.

**Additional function** Search\_without\_lexicographic is also added.



**This sorting step is avoided as we are only supposed to return source and final airport.**

**Complexity**

Time Complexity: O(∣E∣log∣V∣∣E∣​) where ∣E∣ is the number of edges (flights) in the input.

As one can see from the above algorithm, during the DFS process, we would traverse each edge once. Therefore, the complexity of the DFS function would be ∣E∣.

However, before the DFS, we need to sort the outgoing edges for each vertex. And this, unfortunately, dominates the overall complexity.

It is though tricky to estimate the complexity of sorting, which depends on the structure of the input graph.

In the worst case where the graph is not balanced, i.e. the connections are concentered in a single airport. Imagine the graph is of star shape, in this case, the starting airport would assume half of the flights (since we still need the return flight). As a result, the sorting operation on this airport would be exceptionally expensive, i.e. NlogN, where N=2∣E∣​. And this would be the final complexity as well, since it dominates the rest of the calculation.

Let us consider a less bad case, or an average case, where the graph is less clustered, i.e. each node has the equal number of outgoing flights. Under this assumption, each airport would have (2⋅∣V∣)∣E∣​ number of flights (still we need the return flights). Again, we can plug it into the NlogN minimal sorting complexity.. As a result, we have ∣V∣⋅(NlogN), where N=2⋅∣V∣∣E∣​. If we expand the formula, we will obtain the complexity of the average case as O(2∣E∣​log2⋅∣V∣∣E∣​)=O(∣E∣log∣V∣∣E∣​)

Space Complexity: O(∣V∣+∣E∣) where ∣V∣ is the number of airports and ∣E∣ is the number of flights.

In the algorithm, we use the graph, which would require the space of ∣V∣+∣E∣.

Since we applied recursion in the algorithm, which would incur additional memory consumption in the function call stack. The maximum depth of the recursion would be exactly the number of flights in the input, i.e. ∣E∣.

As a result, the total space complexity of the algorithm would be O(∣V∣+2⋅∣E∣)=O(∣V∣+∣E∣).