

## Metro trip frequencies

Some metro systems (e.g. the Shanghai one) use electronic cards to regulate access. Since cards have an ID it is possible to track frequencies of metro trips for statistical purposes.

Function `startTrip(cardID, stationID)` simulates the start of a trip at a certain station for a certain card

Function `endTrip(cardID, stationID)` simulates the end of a trip at a certain station

Function `mostFrequentTrip` returns the trip (a pair `stationFrom – stationTo`) most frequently used in the metro system

Function `mostFrequentTripPerUser` returns the trip used most frequently by a certain user (`user == cardID`)

**Constraints:** minimize execution time for the `mostFrequentTrip`, `mostFrequentTripPerUser` functions.

Notes. Trip from A to B is different than from B to A

Ex

```
void main(){

    startTrip(1234, "People square" );    endTrip(1234, "Siping road" );
    startTrip(2234, "People square" );    endTrip(2234, "Siping road" );
    startTrip(5534, "People square" );    endTrip(5534, "Siping road" );
    startTrip(5534, "People square" );    endTrip(5534, "Siping road" );

    startTrip(1234, "People square" );    endTrip(1234, "Shanghai railroad" );
    startTrip(3334, "People square" );    endTrip(3334, "Shanghai railroad" );
    startTrip(5534, "People square" );    endTrip(5534, "Shanghai railroad" );

    mostFrequentTrip(); // returns People Square – Siping Road
    mostFrequentTripPerUser(5534); // returns People Square – Siping Road
    mostFrequentTripPerUser(3334); // returns People Square – Shanghai railroad

}
```

## Discussion of data structure and algorithms

Other solutions are possible and welcome, provided they are of lower or comparable complexity.

-----mostFrequentTrip()

```
struct trip {  
    char* startStation;  
    char* endStation;  
    int nTrips;  
}
```

*trips*: hash table

Key: concatenation of startStation-endStation

Value: struct trip

```
addTrip (cardId, fromStation, toStation){  
    search fromStation-toStation in trips, nTrips++ (complexity: constant)  
}
```

```
mostFrequentTrip(){  
    visit trips, search max(nTrips); (complexity: n1)  
}
```

Note:  $n1 \leq \text{number of possible trips} = \text{numberOfStations} * (\text{numberOfStations} - 1)$

overall computing the most frequent trip requires to record each trip (constant) and a search of maximum (n1)

-----mostFrequentTripPerUser()

```
struct user {  
    int cardId;  
    tripsOfUser;  
}
```

```
struct trip2 {  
    char* startStation;  
    char* endStation;  
    int nTimes;  
}
```

*tripsOfUser* : linked list of trip2, ordered descending on nTimes

*users*: hash table  
Key: cardId  
Value: struct user

```
addTrip(cardId, fromStation, toStation){  
  search cardId in hash table users (constant)  
  in user associated to cardId, search in tripsOfUser the trip fromStation-toStation,  
  increase nTimes++, keeping the list ordered by nTimes  
  complexity: n2 to search the list, constant to reorder  
  (the ordering is constant, because only 1 is added at a time,  
  so the trip will either remain in the same position, or shifted +-  
  1 position)  
}
```

```
mostFrequentTripPerUser(cardId){  
  search cardId in hash table users (constant)  
  in user associated to cardId, return first trip in tripsOfUser (constant)  
}
```

Note:  $n_2$  depends on number of different trips made by cardId, should be  $n_2 \ll n_1$

Note2: for tripsOfUser linked list better than array for the type of ordering (array would require shifting all elements before (or after) the element shifted)

Note3: another option is to keep tripsOfUser not ordered, then order it on demand when function mostFrequentTripPerUser is called. This option is less complex if mostFrequentTripPerUser is called seldom in comparison to addTrip(), and/or is called for a limited number of users compared to the total number of users.