Day 24 / 100 :

Topic - Strings

1 Problem statement: LRU Cache (easy)

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.
Implement the LRUCache class:
LRUCache(int capacity) Initialize the LRU cache with positive size capacity.
int get(int key) Return the value of the key if the key exists, otherwise return -1.
void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.
The functions get and put must each run in O(1) average time complexity.

Example 1:
Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);   // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2);   // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1);   // return -1 (not found)
lRUCache.get(3);   // return 3
lRUCache.get(4);   // return 4

Solutions :
Approach 1 - Brute Force

1. To implement an LRU cache, we need a data structure that allows us to store key-value pairs and provides fast access to both the key and value.
2. We can use a combination of a doubly linked list and a hash map to achieve this.
3. The doubly linked list will keep track of the order in which the keys are accessed, with the most recently used keys at the front and the least recently used keys at the end.

4. The hash map will store references to the nodes in the linked list, allowing us to quickly access and update the key-value pairs.
5. For the get operation, we will first check if the key exists in the hash map. If it does, we will retrieve the corresponding node from the linked list and move it to the front to indicate that it has been recently used. We will then return the value associated with the key. If the key does not exist, we will return -1.
6. For the put operation, we will first check if the key exists in the hash map. If it does, we will update the corresponding node's value and move it to the front of the linked list. If the key does not exist, we will create a new node with the key-value pair and insert it at the front of the linked list. If the cache capacity is exceeded, we will remove the least recently used node from the end of the linked list and remove its reference from the hash map.
7. The time complexity of the get and put operations will be O(1) on average, as all the operations involved (accessing nodes in the linked list, updating the hash map) can be done in constant time.

```
class LRUCache {
private:
    int capacity;
    list<pair<int, int>> cache;
    unordered_map<int, list<pair<int, int>>::iterator> cacheMap;

public:
    LRUCache(int capacity) {
        this->capacity = capacity;
    }

    int get(int key) {
        if (cacheMap.find(key) != cacheMap.end()) {
            // Key found, move it to the front of the cache
            auto iter = cacheMap[key];
            cache.splice(cache.begin(), cache, iter);
            return iter->second;
        }
        return -1; // Key not found
    }

    void put(int key, int value) {
        if (cacheMap.find(key) != cacheMap.end()) {
            // Key exists, update its value and move it to the front of the
cache
            auto iter = cacheMap[key];
            iter->second = value;
```

```
            cache.splice(cache.begin(), cache, iter);
        } else {
            // Key doesn't exist, insert a new key-value pair
            cache.push_front({key, value});
            cacheMap[key] = cache.begin();

            // Check if cache capacity is exceeded, and evict the least
recently used key
            if (cacheMap.size() > capacity) {
                int lruKey = cache.back().first;
                cacheMap.erase(lruKey);
                cache.pop_back();
            }
        }
    }};
```

Solutions :
Approach 2 - optimized Approach

1. We can further optimize the LRU cache implementation by using a combination of a doubly linked list and an ordered map (or ordered dictionary).
2. The ordered map is a data structure that maintains the keys in sorted order, which allows us to access the least recently used key efficiently.
3. The doubly linked list will store the key-value pairs in the order of their usage, with the most recently used pairs at the front and the least recently used pairs at the end.
4. The ordered map will store references to the nodes in the linked list, allowing us to quickly access and update the key-value pairs.
5. For the get operation, we will first check if the key exists in the ordered map. If it does, we will retrieve the corresponding node from the linked list and move it to the front to indicate that it has been recently used. We will then return the value associated with the key. If the key does not exist, we will return -1.
6. For the put operation, we will first check if the key exists in the ordered map. If it does, we will update the corresponding node's value and move it to the front of the linked list. If the key does not exist, we will create a new node with the key-value pair and insert it at the front of the linked list. If the cache capacity is exceeded, we will remove the least recently used node from the end of the linked list and remove its reference from the ordered map.
7. The time complexity of the get and put operations will be O(1) on average, as all the operations involved (accessing nodes in the linked list, updating the ordered map) can be done in constant time.

The ordered map implementation allows us to achieve the same functionality as the brute force solution but with a more efficient and concise code.

```
class LRUCache {
private:
    int capacity;
    list<pair<int, int>> cache;
    unordered_map<int, list<pair<int, int>>::iterator> cacheMap;

public:
    LRUCache(int capacity) {
        this->capacity = capacity;
    }

    int get(int key) {
        auto iter = cacheMap.find(key);
        if (iter != cacheMap.end()) {
            // Key found, move it to the front of the cache
            cache.splice(cache.begin(), cache, iter->second);
            return iter->second->second;
        }
        return -1; // Key not found
    }

    void put(int key, int value) {
        auto iter = cacheMap.find(key);
        if (iter != cacheMap.end()) {
            // Key exists, update its value and move it to the front of the
cache
            iter->second->second = value;
            cache.splice(cache.begin(), cache, iter->second);
        } else {
            // Key doesn't exist, insert a new key-value pair
            cache.push_front({key, value});
            cacheMap[key] = cache.begin();

            // Check if cache capacity is exceeded, and evict the least
recently used key
            if (cacheMap.size() > capacity) {
                int lruKey = cache.back().first;
                cacheMap.erase(lruKey);
                cache.pop_back();
            }
        }
    }};
```

2 Problem statement: Best Time to Buy and Sell Stock (easy)

You are given an array prices, where prices[i] is the price of a given stock on the ith day.
You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.
Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.

Solutions :
Approach 1 -

Intuition of this Problem:
The intuition behind the code is to keep track of the minimum stock value seen so far while traversing the array from left to right. At each step, if the current stock value is greater than the minimum value seen so far, we calculate the profit that can be earned by selling the stock at the current value and buying at the minimum value seen so far, and update the maximum profit seen so far accordingly.

By keeping track of the minimum stock value and the maximum profit seen so far, we can solve the problem in a single pass over the array, without needing to consider all possible pairs of buy-sell transactions. This makes the code efficient and easy to implement.

Approach::

- Initialize a variable n to the size of the input vector prices.
- Initialize variables maximumProfit and minStockVal to 0 and INT_MAX respectively.
- Initialize a loop variable i to 0.
- While i is less than n, do the following:
    a. Set minStockVal to the minimum value between minStockVal and the value of prices at index i.

b. If minStockVal is less than or equal to the value of prices at index i, set maximumProfit to the maximum value between maximumProfit and the difference between prices at index i and minStockVal.
c. Increment i by 1.

- Return maximumProfit.

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        int maximumProfit = 0, minStockVal = INT_MAX;
        int i = 0;
        while (i < n) {
            minStockVal = min(minStockVal, prices[i]);
            // whenever the price of current stock is
greater then then the stock value which we bought then
only we will sell the stock
            if (prices[i] >= minStockVal)
                maximumProfit = max(maximumProfit,
prices[i] - minStockVal);
            i++;
        }
        return maximumProfit;
    }
};
```