1 动态规划

1.1 基本框架

明确 base case -> 明确「状态」-> 明确「选择」 -> 定义 dp 数组/函数的含义。

按上面的套路走，最后的结果就可以套这个框架：

```
# 初始化 base case
dp[0][0][...] = base
# 进行状态转移
for 状态1 in 状态1的所有取值:
    for 状态2 in 状态2的所有取值:
        for ...
            for 选择 in 选择列表:
                dp[状态1][状态2][...] = 求最值(选择1, 选择2...)
```

1.1.1 416. 分割等和子集

给定一个**只包含正整数**的**非空**数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

注意:

1. 每个数组中的元素不会超过 100
2. 数组的大小不会超过 200

示例 1:

```
输入: [1, 5, 11, 5]

输出: true

解释: 数组可以分割成 [1, 5, 5] 和 [11].
```

示例 2:

```
输入: [1, 2, 3, 5]

输出: false

解释: 数组不能分割成两个元素和相等的子集.
```

- **解答**

```
class Solution {
private:
    vector<vector<bool>> memo; //dp表格
public:
    bool canPartition(vector<int>& nums) {
      int sum = 0;
      for (auto iter : nums) { sum += iter;};
```

```
      if (sum % 2 != 0) {
        return false;
      }
      int target = sum / 2;

      memo = vector<vector<int>>(nums.size() + 1,
              vector<int>(target + 1, -1)); //初始化memo
      // memo的basecase的情况
      for (int i = 0; i<= target; i++) {
        memo[0][i] = 0;
      }
      for (int i = 0; i <= nums.size(); i++) {
        memo[i][0] = 1;
      }

      return dp(nums.size() - 1, target, nums);
    }

    bool dp(int K, int target, vector<int>& nums)
    {
      if (memo[K + 1][target + 1] != -1) {
        return memo[K + 1][target + 1];
      }
      bool res;
      //这里是选择
      if (nums[K] > target) {
        res =  dp(K - 1, target, nums);
      } else {
         //状态转移
        res = dp(K - 1, target, nums) || dp(K - 1, target - nums[K], nums);
      }
      memo[K + 1][target + 1] = res;
      return res;
    }
};
```

```
// regex_search example
#include <iostream>
#include <string>
#include <unordered_map>
#include <regex>
using namespace std;

class Node {
public:
    int key, val;
    Node *next = nullptr;
    Node *prev = nullptr;
    Node(int k, int v) {
        this->key = k;
        this->val = v;
    }
};
class DoubleList {
    // 头尾虚节点
```

```
private:
    Node *head, *tail;
    // 链表元素数
    int size;

public:
     DoubleList() {
        head = new Node(0, 0);// 初始化双向链表的数据
        tail = new Node(0, 0);
        head->next = tail;
        tail->prev = head;
        size = 0;
        return;
    }

    // 在链表尾部添加节点 x，时间 O(1)
    void addLast(Node *x) {
        x->prev = tail->prev;
        x->next = tail;
        tail->prev->next = x;
        tail->prev = x;
        size++;
        return;
    }

    // 删除链表中的 x 节点（x 一定存在）
    // 由于是双链表且给的是目标 Node 节点，时间 O(1)
    void remove(Node *x) {
        x->prev->next = x->next;
        x->next->prev = x->prev;
        size--;
        return;
    }

    // 删除链表中第一个节点，并返回该节点，时间 O(1)
    Node* removeFirst() {
        if (head->next == tail)
            return nullptr;
        Node* first = head->next;
        remove(first);
        return first;
    }

    // 返回链表长度，时间 O(1)
    int GetSize() { return size; }
};
class LRUCache {
private:
    unordered_map<int, Node*> map;// key -> Node(key, val)
    DoubleList cache;
    int cap;// 最大容量

public:
    LRUCache(int capacity) {
        this->cap = capacity;
        map = unordered_map<int, Node*>();
        cache = DoubleList();
        return;
```

```
    }
    /* 将某个 key 提升为最近使用的 */
private:
    void makeRecently(int key)
    {
        Node* x = map.at(key);
        // 先从链表中删除这个节点
        cache.remove(x);
        // 重新插到队尾
        cache.addLast(x);
        return;
    }
    /* 添加最近使用的元素 */
    void addRecently(int key, int val) {
        Node* x = new Node(key, val);
        // 链表尾部就是最近使用的元素
        cache.addLast(x);
        // 别忘了在 map 中添加 key 的映射
        map.insert({key, x});
    }

        /* 删除某一个 key */
     void deleteKey(int key) {
        Node* x = map.at(key);
        // 从链表中删除
        cache.remove(x);
        // 从 map 中删除
        map.erase(key);
    }
/* 删除最久未使用的元素 */
 void removeLeastRecently() {
    // 链表头部的第一个元素就是最久未使用的
    Node* deletedNode = cache.removeFirst();
    // 同时别忘了从 map 中删除它的 key
    int deletedKey = deletedNode->key;
    map.erase(deletedKey);
}

public:
    int get(int key) {
    if (!map.count(key)) {
        return -1;
    }
    // 将该数据提升为最近使用的
    makeRecently(key);
    return map.at(key)->val;
}

void put(int key, int val) {
    if (map.count(key)) {
        // 删除旧的数据
        deleteKey(key);
        // 新插入的数据为最近使用的数据
        addRecently(key, val);
        return;
    }

    if (cap == cache.GetSize()) {
```

```
        // 删除最久未使用的元素
        removeLeastRecently();
    }
    // 添加为最近使用的元素
    addRecently(key, val);
}
};


int main ()
{

    return 0;
}

```