

# 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：

```
1 # 初始化 base case
2 dp[0][0][...] = base
3 # 进行状态转移
4 for 状态1 in 状态1的所有取值:
5     for 状态2 in 状态2的所有取值:
6         for ...
7             for 选择 in 选择列表:
8                 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:

```
1 输入: [1, 5, 11, 5]
2
3 输出: true
4
5 解释: 数组可以分割成 [1, 5, 5] 和 [11].
```

示例 2:

```
1 输入: [1, 2, 3, 5]
2
3 输出: false
4
5 解释: 数组不能分割成两个元素和相等的子集.
```

#### • 解答

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

```

8         if (sum % 2 != 0) {
9             return false;
10        }
11        int target = sum / 2;
12
13        memo = vector<vector<int>>(nums.size() + 1,
14                                   vector<int>(target + 1, -1)); //初始化memo
15        // memo的basecase的情况
16        for (int i = 0; i <= target; i++) {
17            memo[0][i] = 0;
18        }
19        for (int i = 0; i <= nums.size(); i++) {
20            memo[i][0] = 1;
21        }
22
23        return dp(nums.size() - 1, target, nums);
24    }
25
26    bool dp(int K, int target, vector<int>& nums)
27    {
28        if (memo[K + 1][target + 1] != -1) {
29            return memo[K + 1][target + 1];
30        }
31        bool res;
32        //这里是选择
33        if (nums[K] > target) {
34            res = dp(K - 1, target, nums);
35        } else {
36            //状态转移
37            res = dp(K - 1, target, nums) || dp(K - 1, target - nums[K], nums);
38        }
39        memo[K + 1][target + 1] = res;
40        return res;
41    }
42 };

```

```

1 // regex_search example
2 #include <iostream>
3 #include <string>
4 #include <unordered_map>
5 #include <regex>
6 using namespace std;
7
8 class Node {
9 public:
10     int key, val;
11     Node *next = nullptr;
12     Node *prev = nullptr;
13     Node(int k, int v) {
14         this->key = k;
15         this->val = v;
16     }
17 };
18 class DoubleList {
19     // 头尾虚节点

```

```

20 private:
21     Node *head, *tail;
22     // 链表元素数
23     int size;
24
25 public:
26     DoubleList() {
27         head = new Node(0, 0); // 初始化双向链表的数据
28         tail = new Node(0, 0);
29         head->next = tail;
30         tail->prev = head;
31         size = 0;
32         return;
33     }
34
35     // 在链表尾部添加节点 x, 时间 O(1)
36     void addLast(Node *x) {
37         x->prev = tail->prev;
38         x->next = tail;
39         tail->prev->next = x;
40         tail->prev = x;
41         size++;
42         return;
43     }
44
45     // 删除链表中的 x 节点 (x 一定存在)
46     // 由于是双链表且给的是目标 Node 节点, 时间 O(1)
47     void remove(Node *x) {
48         x->prev->next = x->next;
49         x->next->prev = x->prev;
50         size--;
51         return;
52     }
53
54     // 删除链表中第一个节点, 并返回该节点, 时间 O(1)
55     Node* removeFirst() {
56         if (head->next == tail)
57             return nullptr;
58         Node* first = head->next;
59         remove(first);
60         return first;
61     }
62
63     // 返回链表长度, 时间 O(1)
64     int GetSize() { return size; }
65 };
66
67 class LRUCache {
68 private:
69     unordered_map<int, Node*> map; // key -> Node(key, val)
70     DoubleList cache;
71     int cap; // 最大容量
72
73 public:
74     LRUCache(int capacity) {
75         this->cap = capacity;
76         map = unordered_map<int, Node*>();
77         cache = DoubleList();
78         return;
79     }
80
81     // 返回 key 对应的 value
82     int get(int key) {
83         if (map.find(key) != map.end())
84             return map[key]->val;
85         return -1;
86     }
87
88     // 将 key 对应的 value 放入缓存
89     void put(int key, int value) {
90         if (map.find(key) != map.end())
91             map[key]->val = value;
92         else if (map.size() == cap) {
93             // 如果满了, 删除最久远的没有用到的那个元素
94             Node* first = removeFirst();
95             if (first != nullptr)
96                 map.erase(first->key);
97         }
98         map[key] = new Node(key, value);
99         cache.addLast(map[key]);
100     }
101 };

```

```

78     }
79     /* 将某个 key 提升为最近使用的 */
80 private:
81     void makeRecently(int key)
82     {
83         Node* x = map.at(key);
84         // 先从链表中删除这个节点
85         cache.remove(x);
86         // 重新插到队尾
87         cache.addLast(x);
88         return;
89     }
90     /* 添加最近使用的元素 */
91     void addRecently(int key, int val) {
92         Node* x = new Node(key, val);
93         // 链表尾部就是最近使用的元素
94         cache.addLast(x);
95         // 别忘了在 map 中添加 key 的映射
96         map.insert({key, x});
97     }
98
99     /* 删除某一个 key */
100    void deleteKey(int key) {
101        Node* x = map.at(key);
102        // 从链表中删除
103        cache.remove(x);
104        // 从 map 中删除
105        map.erase(key);
106    }
107    /* 删除最久未使用的元素 */
108    void removeLeastRecently() {
109        // 链表头部的第一个元素就是最久未使用的
110        Node* deletedNode = cache.removeFirst();
111        // 同时别忘了从 map 中删除它的 key
112        int deletedKey = deletedNode->key;
113        map.erase(deletedKey);
114    }
115
116 public:
117     int get(int key) {
118         if (!map.count(key)) {
119             return -1;
120         }
121         // 将该数据提升为最近使用的
122         makeRecently(key);
123         return map.at(key)->val;
124     }
125
126     void put(int key, int val) {
127         if (map.count(key)) {
128             // 删除旧的数据
129             deleteKey(key);
130             // 新插入的数据为最近使用的数据
131             addRecently(key, val);
132             return;
133         }
134
135         if (cap == cache.GetSize()) {

```

```
136         // 删除最久未使用的元素
137         removeLeastRecently();
138     }
139     // 添加为最近使用的元素
140     addRecently(key, val);
141 }
142 };
143
144
145 int main ()
146 {
147
148     return 0;
149 }
150
```