

# 算法笔记

## 基础语法

### 基本模板

- ```

1  #include<bits/stdc++.h>           // 万能头文件
2  using namespace std;
3
4  using ll=long long;               // 简化long long 数据类型的声明
5  const int MOD=998244353;         // 取模值
6
7  void solve(){}                   // 实际解决方案
8
9  int main(){
10
11     ios::sync_with_stdio(false); // 加速外挂
12     cin.tie(nullptr);            // endl换为 '\n'
13
14     int T=1;
15     cin>>T;                       // 测试案例个数
16     while(T--){solve();}
17     return 0;
18 }
```

### 输入输出

- ```

1
2  cout<<fixed<<setprecision(n); // 指定输出小数位数
3
4  string str;
5  getline(cin,str); // 按行输入,包括空格
6
7  getline(cin,str,','); // 以char类型的字符','为分隔符进行输入
8  while(getline(cin,str,',')){
9      //.....
10 }
```
-

## 数据类型

### • 转换

```

1 | int num=c-'0'; // 字符char转int
2 | char c=num+'0'; // 整型int转char
3 | string str=to_string(num); // 整形int转string
4 | int num=stoi(str); // 字符串string转int
5 | long long num_ll=stoll(str); // 字符串string转long long
6 |
7 | char ch=toupper(c); // 小写转大写
8 | char ch=tolower(c); // 大写转小写

```

### • 判断

- 数值判断isdigit().
- 字符判断isalpha().
- 

## 数学函数

### • 基础函数

- 绝对值 abs().
- 开平方 sqrt().
- 最值 max(a,b), min(a,b).
- 最大公约数gcd(a,b), 最小公倍数lcm(a,b)

### • 模板函数

#### 1. is\_prime ( 素数判断 )

```

1 | bool isPrime(int n) {
2 |     if (n < 2) return false;
3 |     // i * i <= n 可能会溢出, 建议写成 i <= n / i
4 |     for (int i = 2; i <= n / i; i++) {
5 |         if (n % i == 0) return false;
6 |     }
7 |     return true;
8 | }

```

#### 1 | 2. \*\*Factorial\*\* ( 阶乘, 全排列 )

```

1 | const int MAX_N=.....;
2 | long long fact[MAX_N];
3 |
4 | void Factorial(long long n){
5 |     fact[0] = 1;
6 |     for (int i = 1; i < MAX_N; i++){
7 |         fact[i] = (fact[i - 1] * i) % MOD;
8 |     }
9 | }

```

#### 3. GCD,LCM ( 辗转相除法 )

```

1 // 递归版
2 long long gcd(long long a, long long b) {
3     return b == 0 ? a : gcd(b, a % b);
4 }
5 // 非递归版：避免栈溢出（处理极大数字时）
6 long long gcd_iter(long long a, long long b) {
7     while (b) {
8         a %= b;
9         swap(a, b);
10    }
11    return a;
12 }
13
14 long long lcm(long long a, long long b) {
15     if (a == 0 || b == 0) return 0;
16     // 注意：先除后乘可以防止 a * b 直接相乘导致的溢出
17     return (a / gcd(a, b)) * b;
18 }

```

#### 4. $(A^B) \% MOD$ (快速幂)

```

1 // 迭代写法
2 long long modpow(long long a, long long b){
3     long long res = 1;
4     while (b > 0){
5         if (b & 1) res = res * a % MOD; // 二进制最后一位为 1 的情况
6         a = a * a % MOD;
7         b >>= 1; // 幂次减半
8     }
9     return res;
10 }
11 // 递归写法
12
13 long long modpow(long long a, long long b){
14     if (b == 1) return a;
15     long long res = modpow(a, b >> 1);
16     res = res * res % MOD;
17     if (b & 1) return a * res % MOD;
18     return res;
19 }

```

◦ 补充: 费马小定理

## • 技巧结论

◦ 分数计算  $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$

```
1 sum += (1.0 / i); // 用 1.0 进行除法, 强制转化为浮点运算
```

◦ 约瑟夫环 (如: 星期1~7循环).

```

1 (index + Δstep) % n; // 顺时针循环
2 (index - Δstep % n + n) % n; // 逆时针循环

```

◦ 两点间距直接函数

```
1 hypot(x1-x2, y1-y2); // 返回两点间距
```

## • 位运算

- 奇偶判断

```
1 | (n&1)==1?"奇数":"偶数";
```

- 运算结论

```
1 | a+b=a|b --> a&b=0 // 推导
2 |
```

## STL容器

### 迭代器

- ```
1 | auto itA=v.begin(); // 指向容器第一个元素,*it表示实际存储的值
2 | auto itB=v.end(); //指向最后一个元素的下一个位置,*it报错
3 | for(auto it=v.begin();it!=v.end();it++);// 遍历
4 |
5 | auto itA=v.rbegin(); // 指向容器最后一个元素
6 | auto itB=v.rend(); //指向第一个元素的前一个位置
7 | for(auto it=v.rbegin();it!=v.rend();it++);// 逆序遍历
8 |
9 | min_element(v.begin(),v.end()); // 返回指向最小值的迭代器
10 | max_element(v.begin(),v.end()); // *max_element得到具体值
```

### algorithm

- **sort (排序)**

```
1 | sort(v.begin(),v.end()); // 升序排序
2 | sort(v.begin(),v.end(),greater<T>); // 降序排序
3 |
4 | sort(v.begin(), v.end(), [](int a, int b) {
5 |     return a > b;
6 | }); // 自定义排序
```

- **merge (合并)**

```
1 | //merge(iterator beg1, iterator end1, iterator beg2, iterator end2,
   | iterator dest);
2 | merge(v1.begin(), v1.end(), v2.begin(), v2.end(), vtarget.begin()); //
   | 合并两个序列
3 |
```

- **accumulate (累加和)**

```

1 //accumulate(iterator beg, iterator end, value);
2 int total = accumulate(v.begin(), v.end(), 0); // 计算容器元素累计总和
3

```

- **fill** ( 填充 )

```

1 //fill(iterator beg, iterator end, value);
2 fill(v.begin(), v.end(), val); //容器区间内元素填充为指定的值

```

- **count** ( 计数 )

```

1 // count(iterator beg, iterator end, value), 返回值为整数
2 int numX=count(v.begin(),v.end(),x); // 统计容器区间内值为 x的个数

```

## string

- 增删改查

```

1 strA+=strB; // 尾部拼接
2 str.insert(index,"A"); //指定索引index位置插入字符"A"
3
4 auto pos=str.find("abc"); // 查找str是否包含子串,pos为size_t类型,直接输出
   则为索引
5 if(pos!=string::npos){} // 表示查找成功的情况

```

- **reverse** ( 反转字符串 )

```

1 reverse(str.begin(),str.end()); // 无返回值

```

- **replace** ( 替换字符 )

```

1 replace(str.begin(),str.end(),'A','B'); // 将所有'A'替换为'B', 无返回值

```

- **transform** ( 转换大小写 )

```

1 transform(s.begin(), s.end(), s.begin(), ::tolower); // 全部转为小写字符
2 transform(s.begin(), s.end(), s.begin(), ::toupper); // 全部转为大写字符

```

- **substr** ( 分割子串 )

```

1 // substr(起始索引, 字符个数)
2 string t=str.substr(0,2); // 从索引0位置开始分割str的两个字符

```

- **remove+erase** ( 去除所有空格 )

```

1 str.erase(remove(str.begin(),str.end(),' '),str.end());
2
3 auto new_end=remove(str.begin(), str.end(), ' '); // 将所有非空格字符覆
   盖到容器前面,保持相对顺序,返回的迭代器指向新范围的末尾(第一个被"移除"的空格
   的位置)
4 str.erase(new_end, str.end()); // 从 new_end 到容器末尾的所有元素进行真正
   的删除

```

## vector ( 动态数组 )

- 一维动态数组声明

- ```

1 | vector<T>v; // 只声明, 不开辟空间, 后续用v.push_back()插入
2 | vector<T>v(n); // 开辟大小为n的空间并初始化为0, 可直接用v[i]
3 | vector<T>v(str.begin(),str.end()); // 用字符串进行拷贝构造

```
- 增删
 

```

1 | v.push_back(ele); // 在数组末尾增添元素ele
2 | v.pop_back(); // 删除最后一个元素
3 |
4 | v.insert(iterator,ele); // 在迭代器it位置插入元素ele
5 | v.erase(iterator); //删除迭代器it位置的元素
6 | v.erase(iteratorA,iteratorB); // 删除迭代器itA~itB之间的元素

```
  - 查找
 

```

1 | v.empty(); // 判断是否为空
2 | v.size(); // 返回元素个数
3 |
4 | v.begin(); // 返回首元素的值
5 | v.back(); // 返回尾元素的值

```

## stack (栈)

- ```

1 | stack<T> stk; // 声明容器
2 |
3 | stk.push(val); // 向栈顶添加元素val
4 | stk.pop(); // 从栈顶移除第一个元素
5 | stk.top(); // 返回栈顶元素
6 |
7 | stk.empty(); // 判断栈是否为空
8 | stk.size(); // 返回栈的大小

```
- 习题集
  - [验证栈序列](#)

## queue (队列)

- ```

1 | queue<T> que; // 声明容器
2 |
3 | que.push(val); // 往队尾添加元素val
4 | que.pop(); // 从队头移除第一个元素
5 | que.front(); // 返回队首元素
6 | que.back(); // 返回队尾元素
7 |
8 | que.empty(); // 判断队列是否为空
9 | que.size(); // 返回队列的大小

```
- 

## list (双向链表)

- 适用条件: 保留插入顺序,支持拼接
- ```

1 | list<T> lt;
2 |
3 | lt.push_front(val); // 在容器开头插入一个元素
4 | lt.push_back(val); // 在容器尾部加入一个元素

```

```

5
6 lt.pop_front(); // 从容器开头移除第一个元素
7 lt.pop_back(); // 删除容器中最后一个元素
8
9 lt.insert(pos,lt.begin(),ls.end()); //在pos位置插入[beg,end)区间的数据，无返回值。
10 lt.erase(lt.begin(),lt.end()); //删除[beg,end)区间的数据，返回下一个数据的位置。
11 lt.erase(it); //删除迭代器it所指位置的元素，返回下一个数据的位置。
12 lt.remove(val); //删除容器中所有与val值匹配的元素。
13
14 lt.front(); // 返回首元素的值
15 lt.back(); // 返回尾元素

```

- 反转和排序

```

1 lt.sort(); // 默认的排序规则从小到大
2 lt.sort(greater<T>()); // 降序
3 lt.sort(cmp); // 自定义
4
5 lt.reverse();

```

## priority\_queue (优先队列)

- 声明

```

1 priority_queue<T>pq; // 最大值先出队
2 priority_queue<T, vector<T>, greater<T> > pq; // 最小值先出队
3 priority_queue<int, vector<int>, decltype(cmp)> pq(cmp); // 自定义
4
5 auto cmp = [](int a, int b) { return 表达式(例:a > b); };

```

- 两端优先队列考虑使用 multiset

## set (集合)

- set (所有元素在插入时自动被排序)

- 键值对声明

```

1 set<T>st; // 默认构造声明
2 set<T,greater<T1> >st; // 元素降序排序
3 set<T,decltype(cmp)> st(cmp); // 自定义排序规则
4
5 auto cmp= [](T1 a,T2 b){return 表达式(例: a>b );} // 自定义

```

- 基本操作

```

1 st.size(); // 返回元素个数(去重后)
2 st.empty(); // 判断是否为空,返回值bool
3
4 st.insert(ele); // 插入元素ele
5 st.insert(ele).second // 返回bool值,插入成功为true, 已有元素ele则返回false
6 st.erase(ele); // 删除元素ele
7 st.erase(it); // 删除迭代器it所在指向位置的值
8
9 st.count(ele); // 返回值为ele的个数
10 st.find(ele); // 查找元素ele是否存在,存在则返回it,否则返回st.end()

```

- **multiset** ( 允许容器中有重复的元素 )

```

1 mset.erase(val); // 删掉所有为val的值
2
3 mset.erase(mset.rbegin()); // 删掉最大值(默认)
4 ms.erase(prev(ms.end())); // 同理
5 ms.erase(--ms.end()); // 同上
6
7 mset.erase(mset.begin()); // 删掉最小值(默认)
8
9
10 int getPre(int x){ //TODO 实现找前驱
11     auto it = mset.lower_bound(x); // 找到大于等于x的元素的最大值
12     if(it == mset.begin()) return -1;
13     else return *prev(it); // 找到第一个比x小的数,prev()返回it的前一个迭代器,也可以用*(--it)
14 }
15
16 int getBack(int x){ //TODO 实现找后继
17     auto it = mset.upper_bound(x); // 找到第一个比x大的元素
18     if(it == M.end()) return -1;
19     else return *it;
20 }

```

- **unordered\_set** ( 仅用来去重 )

## map (键值对)

- **map** ( 用于key需要排序的情况 )

- 键值对声明

```

1 map<T1,T2>mp; // 一般声明,keys升序排序
2 map<T1,T2,greater<T1> >mp; // keys降序排序
3 map<T1,T2,decltype(cmp)> mp(cmp); // 自定义排序规则
4
5 auto cmp=[](T1 a,T2 b){return 表达式(例: a>b );} // 自定义

```

- 基本操作

```

1 mp.size(); //返回keys元素的数目
2 mp.empty(); //判断是否为空,返回值bool
3
4 mp[key]=value; // 插入键值对{key:val}
5 mp.erase(key); // 删除map中键为key的键值对
6
7 mp.count(key); // 统计key元素的个数

```



```

8 mp.find(key);    // 查找key是否存在,存在则返回it,否则返回mp.end()
9
10 for(auto it = mp.begin();it != mp.end();it++){
11     if(mp.find(key)==mp.end()){        // 未查找到元素的情况
12         cout << "key:"<<it->first;      // 访问用it->first为键key,it->second为值
13         val
14     }
15     cout <<"value:"<< it->second;
16 }

```

- **multimap** ( 用于键key可重复的情况 )
- **unordered\_map** ( 不需要排序, 性能最好, 最常用 )

## 基础算法

### 模拟/枚举

严格遵循题目规则，不重不漏地穷举所有可能状态

1. **建表查询**: 将**状态抽象化**, 对于日期(仅12个固定值), 方位(上,下,左,右)等可枚举的量,直接建表

```

1 // 数组查表处理月份天数
2 int days[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
3 // 数字确定移动方向
4 int dir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

```

2. **边界划分**: 保证所有过程都在问题之内

### 贪心

鼠目寸光: 不从整体最优上加以考虑，而是始终做出**在当前看来最好的选择**

- 注意点:
  1. 可以一直向前走, 没有后顾之忧. ( 贪心思想的选择, 局部最优走到终点即为全局最优 )
  2. 关键问题转化, 边界条件的处理. ( 找到贪心策略 )
  3. 每轮都将问题转化成规模更小的子问题, 直到问题被解决.
  4. 一般配合排序等方法找最值

- 典型例题思维:

- 双向约束(邻域依赖)问题: 将双向约束拆分为两个单向约束, 分别进行遍历。先解决左邻居, 再解决右邻居, 最后取极值。

[分发糖果](#)

[接雨水](#)(贪心,双指针,动态规划,单调栈)

- 容器选择,性能优化: vector(动态数组)的增删操作相对于list(双向链表)的耗费要大得多,依据已有思路可以转变容器优化性能

[根据身高重建队列](#)

- 区间排序方向选择: **不重叠调度**选右端点; **合并区间**选左端点。

- 选右边界排序,可避免区间范围过大的问题

[用最少数量的箭引爆气球](#)(合并区间)

[无重叠区间](#)(左右均可,推荐右)

- 遍历顺序的选择: 不要有后顾之忧,宁愿边界多处理,也不回头看一步

[单调递增的数字](#)

[划分字母区间](#)

## backtrack ( 回溯 )

核心思想是从一个初始状态出发,暴力搜索所有可能的解决方案,当遇到正确的解则将其记录,直到找到解或者尝试了所有可能的选择都无法找到解为止. 本质上是穷举,效率并不高,一般为指数级别,可借助剪枝优化来提高效率.

- 一般解决以下问题:

- 1 | 1. 组合问题: N个数里面按一定规则找出k个数的集合, 不强调元素顺序
- 2 | 2. 切割问题: 一个字符串按一定规则有几种切割方式
- 3 | 3. 子集问题: 一个N个数的集合里有多少符合条件的子集
- 4 | 4. 排列问题: N个数按一定规则全排列, 有几种排列方式, 强调元素顺序
- 5 | 5. 棋盘问题: N皇后, 解数独等等

- 一般模板[ 回溯函数, 终止条件,遍历过程,( 剪枝 )]

```

1  // 参数列表 vector<vector<T>> >&res, vector<T>&path, int startIndex,
   vector<T>nums
2  void backtrack(参数) {
3      // 求子集等情况,可在该行存放结果
4      if (终止条件) {
5          存放结果;
6          return;
7      }
8
9      for (选择: 本层集合中元素 (树中节点孩子的数量就是集合的大小)) {
10         //for(int i=startIndex; i<nums.size()&& (剪枝操作); i++)
11             if(...){continue;} // 也可以剪枝
12             处理节点;
13             // path.push_back(nums[i]);
14             backtrack(路径, 选择列表); // 递归
15             // backtrack(res, path, i+1, nums);
16             回溯, 撤销处理结果;
17             // path.pop_back();
18         }
19     }
20
21     // 调用
22     vector<vector<T>> res;
23     vector<T>path;
24     backtrack(res,path , 0, nums);

```

- 关于startIndex

- i==startIndex是为了避免单纯因为顺序不同而重复出现的子答案
- 在回溯过程中for(i=start; xxxx; i++){},i++是为了确保子答案的选取时不会再从头开始选取

- 如果是排列，不同顺序代表了不同的答案，则int i=0. startIndex不再需要
- 去重操作,if (i > start && num[i] == num[i-1]) continue;
- 关于sum Target
  - 题目要求子项和等于target，因此首先要注意除了正常return，还要考虑sum>target立刻return
  - 剪枝操作，sum+子项<=target，但是要注意子项必须满足升序排列，有必要时提前sort()
- 关于优化
  - 组合类问题, 如果仅用于求符合解的个数, 可以使用dp优化[例: 目标和](#)

## Dynamic Programming (动态规划)

将一个问题**分解**为一系列更小的子问题，并通过**存储子问题的解**来避免重复计算, 旧状态+决策=新状态

- 一般解题步骤:
  1. 确定dp数组（一维或二维）以及下标的含义
  2. 借助数学归纳思想, 确定递推公式（状态转移方程）
  3. dp数组如何初始化
  4. 确定遍历顺序
  5. 举例推导dp数组
- 一般模板

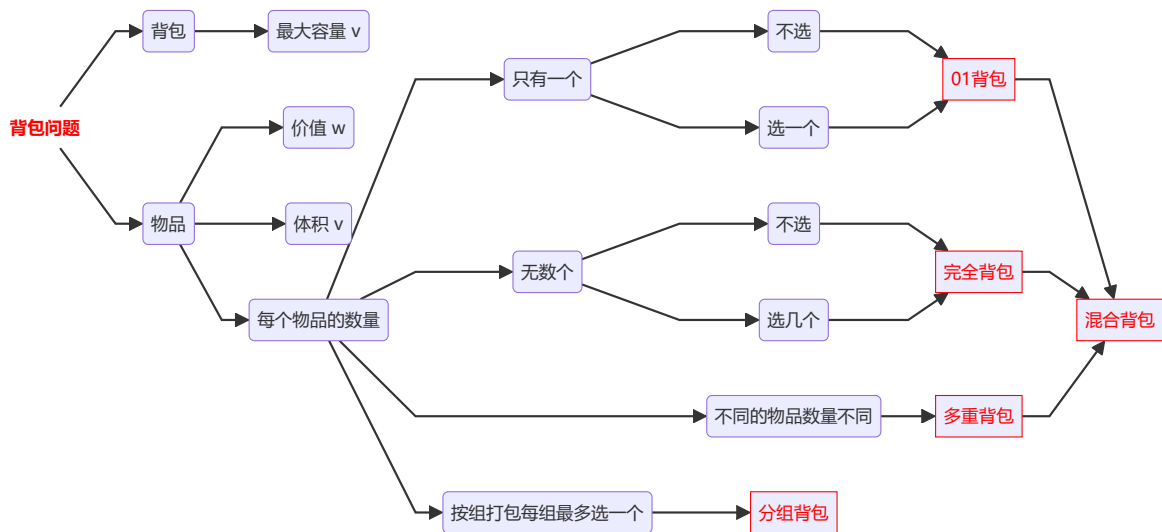
以leetcode 96.[不同的二叉搜索树](#)为例

```

1  int numTrees(int n) {
2      vector<int> dp(n + 1); // 明确下标含义，声明dp数组，大小为n-1
3      dp[0] = 1; // 依据题意初始化基础值
4      for (int i = 1; i <= n; i++) { // 确定遍历规则和顺序
5          for (int j = 1; j <= i; j++) {
6              dp[i] += dp[j - 1] * dp[i - j]; // 递推公式，依赖于子问题的解
7          }
8      }
9      return dp[n];
10 }

```

- 典型类题目->**背包问题**



。 解题步骤( 二维DP数组 ):

1. 声明数组 $dp[i][j]$ :  $i$ 表示第 $i$ 个物品,  $j$ 表示容量为 $j$ 的背包,  $dp[i][j]$ 表示从下标为 $[0 \sim i]$ 的物品里任意取, 放进容量为 $j$ 的背包, 价值总和最大是多少
2. 状态转移分析:
  1. 不放物品  $i$ : 最大价值延续, 即 $dp[i][j] = dp[i-1][j]$
  2. 放物品  $i$ : 背包空出物品 $i$ 的容量, 转为 $j - \text{weight}[i]$ 后, 放物品 $+ \text{value}[i]$ , 即 $dp[i][j] = dp[i-1][j - \text{weight}[i]] + \text{value}[i]$
  3. 得出递推公式  $dp[i][j] = \max(dp[i-1][j], dp[i-1][j - \text{weight}[i]] + \text{value}[i])$ ;
3. 初始化dp数组: 每一个 $(i, j)$ 都需要依赖左上方和正上方的解, 则用第1个物品初始化第一行, 从第2个物品开始分析; 背包容量为0时不放物品, 则第一列初始化为0; 非零下标无需初始化, 任意值均可
4. 遍历顺序( 物品先? 背包先? ): 对于01背包的二维DP均可
5. 打印DP数组( debug ):

| 物品(索引 $i$ ) \ 背包容量 $c$ | 0 | 1      | 2      | 3      | 4      |
|------------------------|---|--------|--------|--------|--------|
| 物品( 0 ), 价值 $val_1$    | 0 | 当前最大价值 | 当前最大价值 | 当前最大价值 | 当前最大价值 |
| 物品( 1 ), 价值 $val_2$    | 0 |        |        |        |        |
| 物品( 2 ), 价值 $val_3$    | 0 |        |        |        | 最终解    |

6. 空间优化( 二维DP -> 一维滚动数组 ):

- $dp[j]$ 表示容量为 $j$ 的背包, 所背的物品价值可以最大为 $dp[j]$ ;
- 递推公式为 $dp[j] = \max(dp[j-1], dp[j - \text{weight}[i]] + \text{value}[i])$ ;
- 初始化 $dp[0] = 0, dp[j] = 0$  (非负数值的最小值, 一般取0就可)
- 遍历顺序: 先顺序遍历物品, 后倒序遍历背包, 保证每个物品只被添加一次

。 01背包模板:

- 以leetcode 416. [分割等和子集](#) 为例(非本题最优解,本题可用vector<bool>优化)

```

1 // 二维DP数组
2 bool canPartition(vector<int>&nums) {
3     int sum = accumulate(nums.begin(), nums.end(), 0);
4     if (sum % 2 != 0) return false; // 排除无效判断
5
6     int target = sum / 2;
7     vector<vector<int>> dp(nums.size(), vector<int>(target+1));
8     for(int j=0; j<=target; j++){ // 初始化第一行,能放则放,否则为 0
9         if(j<=nums[0]) dp[0][j]=nums[0];
10    }
11    // 填充DP表
12    for(int i=1; i<nums.size(); i++){
13        for(int j=1; j<=target; j++){
14            if(j<nums[i]) // 放不下,继承上一行
15                dp[i][j]=dp[i-1][j];
16            else // 能放下,放与不放,取最大值
17                dp[i][j]=max(dp[i-1][j], dp[i-1][j-nums[i]]+nums[i]);
18        }
19    }
20    return dp[nums.size()-1][target]== target;
21 }

1 // 空间优化: 一维滚动数组
2 bool canPartition(vector<int>& nums) {
3     int sum = accumulate(nums.begin(), nums.end(), 0);
4     if (sum % 2 != 0) return false; // 排除无效判断
5
6     int target = sum / 2;
7     vector<int> dp(target + 1, 0); // 声明大小为 n+1的DP数组,表示从 0~j
    可取的最大元素和,初始化为 0
8     for (int i = 0; i < nums.size(); i++) { // 顺序遍历
9         for (int j = target; j >= nums[i]; j--) { // 逆序遍历
10             dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]); // 递推公式
11         }
12     }
13     return dp[target] == target;
14 }

```