# 算法总结：

## 动态规划算法

### 算法特性与步骤

个人以为，在动态规划算法中最重要的，就是找出**动态转移方程**。

但是在此之前，我们需要注意动态规划算法的几个特性：

1. **最优子结构性质**。因为动态规划尝试得到最优解，这个**最优解的子问题包含的解也是最优的**。最优子结构这个性质为动态规划解决问题提供了重要线索。因为我们可以自底向上反过来推导，**以斐波那契数列为例子**，每一个数n的值f(n)=f(n-1)+f(n-2)，其实相当于已经给出了动态转移方程。那么我们只需要用两个数字分别标记f(n-1)和f(n-2)，用一个数字表示和，并且每次循环时，三个标记都往前更新.

**int n1=0, n2=1, sum; for: sum=n1+n2; n1=n2; n2= sum; return n2;**

1. **子问题重叠。**使用递归算法自顶向下进行求解时，产生子问题不会是新的问题，有些子问题会被重复计算多次。动态规划利用了子问题重叠性质，将每次计算好的子问题保存在一个表格中。
2. **无后效性**。子问题的解一旦确定，就不再改变，不会被之后的包含它的更大问题所影响。简单来说就是未来不会影响过去。

动态规划的解题思路：

分析优化解的结构，递归的定义最优解，自底向上的计算优化解的代价保存，获取构造最优解的信息。根据最优解的信息构造最优解。

### 帮助理解的例题

#### 跳台阶

一共有n阶台阶，每次可以跳1阶或者2阶，求到n阶台阶有多少种跳法？

思路：首先找到规律。每次跳一个台阶或者两个台阶，当n=0 return 0, n=1时有1种，n=2时有两种跳法。 当n大于2时，可以得出最后一步，要么跳1阶台阶，要么跳2阶台阶。 也就是最后一跳，要么站在n-2的台阶上，要么站在n-1的台阶上。

根据最优子结构性质，如果假设我们跳到n的时候已经能够取得所有情况，那么n-1和n-2的台阶也是。以这种形式递归下来，假设跳到台阶n的总情况有f(n),那么状态转移方程就是f(n) = f(n-1)+f(n-2)，同斐波那契数列。

那么代码是:

int num1=1,num2=2, sum;

for(int i=3; i<=n; i++){

sum = num1+num2;

num1=num2;

num2 = sum;

}

return num2;

#### 01背包问题（经典）

你有一个承受重量为10的背包，现在有n个物品可以放入，每个物品有他的价值和重量。

每个类型的物品只有一个，选择放入与否，在重量承受范围内使得拿到的物品价值最大化。

设n=4，下列是这些物品的属性（0占位用）

int[ ] value = {0,2,4,3,7};

int[ ] weight = {0,2,3,5,5};

思路：一共有n个物品，依次都需要考虑这个物品拿还是不拿。如果拿，value增加，weight减少，不拿就直接跳过。我们假设现在正在考虑第i个物品拿还是不拿，i以前的物品已经经过挑选得到了最大的价值（最优子结构）。如果考虑以dp[][]二维数组来记录，那么dp[i][j]=value, i表示正在考虑第i个物品，j表示此时包的剩余重量。

则：现有选取到i-1个物品的最大价值dp[i-1][j]，在考虑第i个物品时：

如果拿，此时的价值为 dp[i][j]= dp[i-1][j-weight[i]]+value[i]

如果不拿，此时的价值为dp[i][j] = dp[i-1][j]

则比较那个价值大，所以动态转移方程为f(i,j) = Math.max( f(i-1,j), f(i-1,j-w[i])+v[i]) 其中使用的缩写 w=weight , v = value

int[][] dp =new int[item+1][capacity+1];

for (int i = 1; i < item; i++) {//考虑第i个物品

for (int j = 1; j < capacity; j++) {//当背包容量为j时

if(j<weight[i]) dp[i][j] = dp[i-1][j];//装不下

else{

dp[i][j] = Math.*max*(dp[i-1][j], dp[i-1][j-weight[i]]+value[i]);

//比较拿和不拿，那个value更大

}

}

}

return dp[item][capacity];

当然也可以用dp这个一位数组来记录拿下的数据，dp[i]中，i表示重量，dp[i]表示此时的价值。

#### 不重复的最长子字符串

找出一个字符串中，不包含重复元素的子字符串，返回其长度。

*输入: "abcabcbb"*

*输出: 3*

*解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。*

思路：使用HashMap来存储，因为比较好找重复元素。

public int lengthOfLongestSubstring(String s){

if(s==null) return 0;

if(s.length()==1) return 1;

HashMap<Character,Integer> map = new HashMap<>();

map.put(s.charAt(0),0);

int cur=1,max=1;

//cur=currentLength, max = maxLength

for (int i = 1; i < s.length(); i++) {

int indexOfRepeat = map.getOrDefault(s.charAt(i),-1);

//得到最左边的重复字符对应的索引

cur = cur<i-indexOfRepeat ? cur+1 : i-indexOfRepeat;

//如果在当前长度之中找到了重复元素，舍弃重复元素前半部分，接着往后遍历，如果当前长度之前出现重复元素，那么没关系接着遍历.

max=Math.*max*(cur,max);

map.put(s.charAt(i),i);

}

return max;

}