动态规划

Dynamic programming, 简称dp

动态规划是一种通过把原问题分解为相对简单的子问题， 来求解复杂问题的算法。

动态规划问题的特征

1. 动态规划问题一般都是求最值，比如求最长公共子序列，矩阵最小路径和等等。
2. 求解动态规划的基本操作是穷举 (因为要求最值，大部分情况下要把所有情况穷举处理，然后在其中找最值)
3. 动态规划问题的三要素是重叠子问题，具备最优子结构，无后效性
4. 动态规划算法的根本目的是解决冗余, 是用空间换时间
5. 动态规划最难的地方是写状态转移方程

- 重叠子问题：是指在用递归算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题，有些子问题会被重复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次，然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果，从而获得较高的效率，降低了时间复杂度。
- 最优子结构：如果问题的最优解所包含的子问题的解也是最优的，而且每个子问题间独立, 我们就称该问题具有最优子结构性质
- 无后效性：以前的状态和以前的状态变化过程，不会影响将来的状态变化。将来的状态只能由当前状态直接影响。
- 状态转移方程：

这里面最困难的一步就是写出状态转移方程了。

那么什么是状态转移方程呢?

举个简单的例子，高中数学中我们做过各种递推的题，比如一个斐波那契数列

0、1、1、2、3、5、8、13、21、34

它的递推公式是： F(1)=1，F(2)=1, F(n)=F(n - 1)+F(n - 2)

n的状态是由n-1和n-2的状态相加转移过来的,

这就是状态转移方程!

总结来说：当我知道了n-1阶段的状态和结果时，我们可以通过一种对应关系来得到n阶段的状态，这种用函数表示前后阶段关系的方程，被称为状态转移方程。

解题思想

1. 明确base case 比如斐波那契数列中的 F(1) = 1, F(2) = 1
2. 明确状态
3. 明确决策
4. 定义dp数组的含义

```
# 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 ...
            dp[状态1][状态2][...] = 求最值(选择1，选择2...)
```

重叠子问题举例 - 斐波那契数列

上面描述了一大通, 全是文字的概念, 同学们可能很快就会忘记, 所以咱们针对重要特性，给出几个例子。

斐波那契数列就可以很好的诠释什么叫重叠子问题，咱们来看一下。

普通递归

```
function fib(n) {
    if (n === 0) {
        return 0;
    }
    if (n === 1 || n === 2) {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

上面就是常规的递归写法, 代码非常简洁, 也很好理解。
但是，这种写法是非常低效的！为什么呢？因为它做了很多多余的计算。这里看一下图 递归-斐波那契数列.png

递归算法的时间复杂度是：子问题个数 * 一个子问题所需时间
这里一棵二叉树的节点数是2^n - 1, 算法时间复杂度是O(2^n).

大家应该可以很明显的看出来, 很多节点被重复计算了, 比如图中f(17)和f(18)都被计算了两次，这就是重叠子问题。

备忘录

所以为了优化这些重叠的子问题, 我们可以维护一个数组来记录已经计算出的结果, 一般称之为备忘录

```
var fib = function(n) {
    if (n === 0) {
        return 0;
    };
    // new Array(n+1)是因为索引0用不到, n是从1开始的
    let memoArray = new Array(n+1).fill(0);
    return fibHelper(memoArray, n);
};

var fibHelper = function(memoArray, n) {
    if (n === 1 || n === 2) {
        return 1;
    }
    if (!memoArray[n]) {
        memoArray[n] = fibHelper(memoArray, n-1) +fibHelper(memoArray, n-2);
    }
    return memoArray[n];
}
```

这里看一下图 备忘录-斐波那契数列.png

大家可以很明显的看出来, 时间复杂度和n成正比了, 变成了O(n).

这种解法其实和动态规划差不多了,
只不过这种解法是自顶向下, 就是比如从f(20)逐步向下分解, 直到f(1)和f(2)
而动态规划是自底向上, 从最小的f(1)和f(2)开始往上推, 直到f(20). 这也是为什么动态规划脱离了递归, 而是由迭代来解决问题。

dp

```
var fib = function(n) {
    if (n === 0) {
        return 0;
    };
    let dp = new Array(n+1).fill(0);
    dp[1] = dp[2] = 1;
    for (let i = 3; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
};
```

这里看图， dp-斐波那契数列.jpg
可以看出来, 这种解法其实和上面的备忘录非常相似, 效率也一样, 只不过是自底向上了

状态压缩

细心的同学可能发现了, 这个题目中不需要维护那么长的一个dp表, 我们只需要关心前两个状态, 所以我们进一步优化一下。

```
var fib = function(n) {
    if (n === 0) {
        return 0;
    };
    if (n === 1 || n === 2) {
        return 1;
    }
    let pre = 1;
    let current = 1;
    for (let i = 3; i <= n; i++) {
        let sum = pre + current;
        pre = current;
        current = sum;
    }
    return current;
};
```

这就是所谓的状态压缩, 这个写法就相当于把dp数组的长度从n压缩到了2, 空间复杂度直接从O(n)优化到了O(1)！！！！

解题思路举例 - 零钱兑换问题

- Tips: 判断金额凑不出的小技巧：先初始化DP table各个元素为amount + 1（代表不可能存在的情况），在遍历时如果金额凑不出则不更新，于是若最后结果仍然是amount + 1，则表示金额凑不

出

题目：给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

示例：

> 示例 1:
> 输入: coins = [1, 2, 5], amount = 11
> 输出: 3
> 解释: 11 = 5 + 5 + 1
>
> 示例 2:
> 输入: coins = [2], amount = 3
> 输出: -1

首先咱们来看一下这个题是否可以用动态规划来解?

这是一个动态规划问题，因为它具备最优子结构，且无后效性。

比如你想求 amount = 11 时的最少硬币数，如果你知道凑出 amount = 10 的最少硬币数（子问题），你只需要把子问题的答案加一（再选一枚面值为 1 的硬币）就是原问题的答案。因为硬币的数量是没有限制的，所以子问题之间没有相互制，是互相独立的。

那么咱们接下来用解题思路的四步法尝试解一下?

1. 确定base case

显然目标金额 amount 为 0 时算法返回 0，因为不需要任何硬币就已经凑出目标金额了
也就是dp[0] = 0;

2. 确定状态

状态是什么？状态就是原问题和子问题中会变化的变量。
由于硬币数量无限，硬币的面额也是题目给定的，只有目标金额会不断地向 base case 靠近，所以唯一的「状态」就是目标金额 amount。

3. 确定决策

决策是什么？决策就是导致「状态」产生变化的行为。
目标金额为什么变化呢，因为你在选择硬币，你每选择一枚硬币，就相当于减少了目标金额。所以说所有硬币的面值，就是你的「选择」。

4. 明确dp数组的定义

一般来说dp数组的索引, 就是上面所说的状态;
dp数组每个值就是我们需要计算的数据;

所以我们可以有这样的dp[i]定义：当目标金额为 i 时，至少需要 dp[i] 枚硬币凑出

```
var coinChange = function(coins, amount) {
    // 这里为什么要声明amount+1长度的数组呢?
    // 因为我们还有索引0, 剩下的是1,...., amount

    // 为什么要将每个值都初始化为amount+1呢?
    // 因为凑出amount所需的硬币数最多是amount个, 也就是说dp[i]应该小于amount+1
    // 由于是找最小值, 所以设置这么一个不应存在的大值, 在最后判断的时候, 如果dp[amout]===amount+1,
    let dp = new Array(amount + 1).fill(amount + 1);
    dp[0] = 0;

    for (let i = 0; i < dp.length; i++) {
        for (let coin of coins) {
            if (i - coin < 0) {
                continue;
            }
            // i 为目标金额, coin为当前的硬币的金额, i-coin代表上一个状态的金额
            // dp[i-coin] 代表上一个状态所需的硬币数, 再加上当前的一个coin, 就凑出了dp[i]

            // 比如输入是[1,2,5], amount = 11
            // 选择1: dp[11] = 一个1 + 剩余需要凑出的dp[11 - 1]的最优解
            // 选择2: dp[11] = 一个2 + 剩余需要凑出的dp[11 - 2]的最优解
            // 选择5: dp[11] = 一个5 + 剩余需要凑出的dp[11 - 5]的最优解

            // 然后再在这三种情况里找最小的值

            dp[i] = Math.min(dp[i],  dp[i-coin] + 1);
        }
    }

    return (dp[amount] === amount + 1) ? -1 : dp[amount];
};
```

实战 - 最长公共子序列

实战 - 最长回文子序列

实战 - 编辑距离

总结一波

看到现在应该可以明白, 咱们这些算法其实都是在穷举, 只不过是想方设法用空间换时间, 来更快地穷举。

解题思想

1. 明确base case 比如斐波那契数列中的 F(1) = 1, F(2) = 1
2. 明确状态
3. 明确决策
4. 定义dp数组的含义

动态规划重点概念讲解

最优子结构到底是什么?

"最优子结构"是某些问题的一种特定的性质, 而不只是动态规划特有的。

1. 举例 最优子结构

比如学校有10个班, 现在已知每个班的最高成绩, 要求计算出全校的最高成绩。

是不是很简单, 直接在10个班的最高成绩中比较, 得出最大的即可。

这个问题就具有最优子结构, 每个班的最高成绩就是子问题, 子问题最优可以保证全局问题最优(但是这个问题不具有重叠子问题的特性, 所以没有必要用动态规划, 简单的比较一下就可以了)

2. 举例 最优子结构 二叉树

求一棵二叉树的最大值

```
function maxVal(root) {
    if (root === null) {
        return -1;
    }
    const left = maxVal(root.left);
    const right = maxVal(root.right);
    return Math.max(root.val, left, right);
}
```

这个问题也是符合最优子结构的, 问题的答案可以通过子树的最大值推导出来

3. 总结

- 最优子结构并不是动态规划独有的, 能求最值的问题大部分都具有这个性质
- 但是最优子结构是动态规划问题的必要条件

为什么动态规划遍历 dp 数组的方式五花八门，有的正着遍

历，有的倒着遍历，有的斜着遍历。

同学们或许对dp数组的遍历顺序摸不着头脑

- 正向遍历

```
let dp = new Array(m).fill(new Array(n)); // m 行 n 列
for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
        // 计算具体的dp[i][j]
    }
}
```

- 反向遍历

```
for (let i = m - 1; i >= 0; i--) {
    for (let j = n - 1; j >= 0; j--) {
        // 计算具体的dp[i][j]
    }
}
```

- 斜向遍历

```
for (let l = 2; l <= n>; l++) {
    for (let i = 0; i <= n - l; i++) {
        let j = l + i - 1;
        // 计算具体的dp[i][j]
    }
}
```

原则

如果你仔细观察的话, 会发现两个重点，也可以说是原则

1. 遍历过程中, 所需的状态必须是已经计算出来的
2. 遍历的终点必须是存储结果的位置

举例 - 编辑距离

我们对dp数组定义完后, 确定base case 是dp[x][0]和 dp[0][y], 最终答案是dp[x][y];
我们通过状态转移方程知道dp[x][y]需要从dp[x-1][y] dp[x][y-1] dp[x-1][y-1]转移而来。

那么根据上面两条特点，我们很容易想到应该使用正向遍历

因为只有这样, 在每一步状态转移的过程中, 才能保证左边 上边 左上边的状态都是经过计算过的。

举例 - 回文子序列

我们对dp数组定义完后, 确定base case 处于中间的对角线

dp[i][j]需要从dp[i+1][j], dp[i][j-1], dp[i+1][j-1]转移而来,

我们最终要求的答案是dp[0][n-1]

这里看图1，更清楚的弄明白题。

然后我们发现，现在应该有两种正确的遍历方式，这里再看图2。

要么从左到右斜着遍历, 要么从下向上从左到右遍历，这样才能保证每一步状态转移的过程中，dp[i][j]的左边 下边 左下边都是经过计算过的。

总结

所以遍历顺序取决于 base case 和最终结果的存储位置, 保证遍历过程中使用的数据都是计算完毕的。

贪心算法

为什么先讲动态规划, 后讲贪心算法呢?

因为贪心算法其实可以认为是dp问题的一个特例, 除了动态规划的各种特征外, 贪心算法还需要满足"贪心选择性质", 当然效率也比动态规划要高。

- 贪心选择性质：每一步走做出一个局部最优的选择，最终的结果就是全局最优。

同学们肯定一眼就看出问题来了, 并不是所有问题都是这样的, 很多问题局部最优并不能保证全局最优, 只有小部分问题具有这种特质。

比如你前面堆满了金条, 你只能拿5根, 怎么保证拿到的价值最大？ 答案当然是：每次都拿剩下的金条中最重的那根, 那么最后你拿到的一定是最有价值的。

比如斗地主, 对方出了一个3, 你手上有345678还有一个2, 按照贪心选择, 这时候你应该出4了, 实际上咱们会尝试出2, 然后345678起飞~

区间调度问题

来看一个经典的贪心算法问题 Interval Scheduling 区间调度.

有许多[start, end]的闭区间, 请设计一个算法, 算出这些区间中, 最多有几个互不相交的区间。

```
function intervalSchedule(intvs: number[][]) {}

// 比如intvs = [[1,3], [2,4], [3,6]]
// 这些区间最多有两个区间互不相交, 即 [1,3], [3,6], intervalSchedule函数此时应该返回2
```

这个问题在现实中其实有很多应用场景, 比如你今天有好几个活动可以参加, 每个活动区间用[start, end]表示开始和结束时间, 请问你今天最多能参加几个活动?

贪心求解

大家可以先想一想，有什么思路?

1. 可以每次选择可选区间中开始最早的那个?

不行, 因为可能有的区间开始很早, 结束很晚, 比如[0,10], 使我们错过了很多短区间比如[1,2],[2,3]

2. 可以每次选择可选区间中最短的那个?

不行, 直接看上面这个例子[[1,3], [2,4], [3,6]], 这样的话会选择出 [1, 3], [2, 4], 并不能保证他们不相交

正确思路

1. 从可选区间intvs里, 选择一个end最小的区间x
2. 把所有与x相交的区间从intvs中剔除
3. 重复1,2, 直到intvs为空, 之前选出的各种区间x， 就是我们要求的结果

把整个思路转换成代码的话, 因为我们要选出end最小的区间, 所以我们可以先对区间根据end升序排序.

1. 选出end最小的区间

由于我们已经排过序了, 所以直接选择第一个区间即可

2. 剔除与x相交的区间

这一步就没第一步那么简单了, 这里建议大家画个图看看

- 代码如下

看一下区间调度.js

区间调度算法的应用

1. 无重叠区间

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

可以认为区间的终点总是大于它的起点。
区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。
示例 1:

输入: [[1,2], [2,3], [3,4], [1,3]]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。

示例 2:

输入: [[1,2], [1,2], [1,2]]

输出: 2

解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。

示例 3:

输入: [[1,2], [2,3]]

输出: 0

解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

- 解答

刚才咱们已经找到了最多有几个互不相交的区间数n, 那么总数减去n就可以了~

咱们来从头写一遍, 就当是复习了

```
var eraseOverlapIntervals = function(intervals) {
    if (intervals.length === 0) {
        return 0;
    }
    let sortArray = intervals.sort((a,b) => a[1] - b[1]);

    let count = 1;

    let xEnd = sortArray[0][1];

    for (let item of intervals) {
        // 注意, 这里题目说了区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠, 所以应该是item[0] >= xE
        if (item[0] >= xEnd) {
            xEnd = item[1];
            count++;
        }
    }

    return intervals.length - count;
};
```

2. 用最少的箭头射爆气球

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

Example:

输入:
[[10,16], [2,8], [1,6], [7,12]]

输出:
2

解释:
对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（射爆另外两个气球）。

- 解答

这个问题和区间调度算法又是非常的类似, 大家稍微转换一下思路即可。

如果最多有n个不重叠的空间, 那么就至少需要n个箭头穿透所有空间，所以我们要求的其实就是最多有几个不重叠的空间。

来看一下这张图

但是这个题里的描述, 边界重叠后, 箭头是可以一起射爆的, 所以两个区间的边界重叠也算是区间重叠。