动态规划--最优子结构

2021年11月15日 星期一　　下午 4:03

题目:
爬梯子
买卖股票
最大子序和

动态规划的穷举有点特别，因为这类问题存在「重叠子问题」，如果暴力穷举的话效率会极其低下， 所以需要「备忘录」或者「DP table」来优化穷举过程，避免不必要的计算。

重叠子问题、最优子结构、状态转移方程就是动态规划三要素。
写出状态方程就能解

$$f(n) = \begin{cases} 1, n = 1, 2 \\ f(n-1) + f(n-2), n > 2 \end{cases}$$

「备忘录」：减除冗余的递归树
每次算出某个子问题的答案后别急着返回，先记到「备忘录」里再返回；每次遇到一个子问题先去 「备忘录」里查一查，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。
一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。
带备忘录的递归解法中的「备忘录」，最终完成后就是 DP table，

「状态压缩」，如果我们发现每次状态转移只需要 DP table 中的一部分，那么可以尝试 用状态压缩来缩小 DP table 的大小，只记录必要的数据，

模板

```
//dp[i]代表什么
    int[] dp = new int[n+1];

    //base case:
    dp[1] = 1;
    dp[2] = 2;
    dp[3] = 3;
    dp[4] = 5;
    dp[5] = 8;

     //数学归纳
    //dp[i-2]+dp[i-1]=dp[i];
    for(int i=3 ; i<n+1; i++) {
       dp[i] = dp[i-2]+dp[i-1];
    }
    return dp[n];
```

```
# 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...)
```

确定状态方程:
1.确定basecase，多写一些，画出来更好看出状态方程
2.确定状态， dp[i]是什么
　　注意状态包含当前的所有选择，选择影响状态，不能让状态变成既定
3.确定选择，也就是状态因什么而变，确定本状态因什么于前状态发生改变。

解题:
1. 递归减冗余分支，用备忘录 —— 自顶向下
2. 迭代dp tabel解状态方程 —— 自底向上
一般来说两种都可以，遍历方向相反

求最值，先考虑一下贪心局部最优能不能作为全局，贪心快。

回溯 ——DFS

2021年11月19日 星期五　　下午 10:15

回溯 = 暴力穷举，解决一个回溯问题，实际上就是一个决策树的遍历过程。

3 个问题：

1、路径：也就是已经做出的选择。

2、选择列表：也就是你当前可以做的选择。

3、结束条件：也就是到达决策树底层，无法再做选择的条件。

Template模板

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

二叉树

2021年11月15日 星期一 下午 10:00

二叉树相关题目**最核**心**的思路是明确当前节点需要做的事情是什么。**

双指针

2021年11月19日 星期五 下午 10:15

双指针：

1.快慢指针： 链表，归并排序、中值、链表成环

2.左右指针: 反转数组、

3.滑动窗口: 子串问题 [滑动窗口]

BFS算法

2021年11月19日 星期五　　下午 10:15

核心是利用队列这种数据结构。

且 BFS 算法常见于**求最值**的场景，因为 BFS 的算法逻辑保证了**算法第一次到达目标时的代价是最小的。**

最小深度，最小路径

原理：

层级遍历： queue存放一层的节点，queue.poll() 判断当前节点是否有target，如果没有，queue.offer(相邻的点)，step++。 Set<> visited 避免走回头路

二分搜索

2021年11月19日 星期五　　下午 10:16

分析二分查找的一个技巧是：**不要出现 else**，而**是把所有情况**用 **else if 写清楚，这样可以清楚地展现所有细 节。**

滑动窗口

2021年11月19日 星期五　　下午 10:16

算法框架

滑动窗口算法的思路是这样：

1、我们在字符串 S 中使用双指针中的左右指针技巧，初始化 left = right = 0，把索引左闭右开区间 [left, right) 称为一个「窗口」。

2、我们先不断地增加 right 指针扩大窗口 [left, right)，直到窗口中的字符串符合要求（包含了 T 中 的所有字符）。

3、此时，我们停止增加 right，转而不断增加 left 指针缩小窗口 [left, right)，直到窗口中的字符串 不再符合要求（不包含 T 中的所有字符了）。同时，每次增加 left，我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 right 到达字符串 S 的尽头。

这个思路其实也不难，第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解。

左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动，这就是「滑 动窗口」这个名字的来历。

四个问题：

1、当移动 right 扩大窗口，即加入字符时，应该更新哪些数据？

2、什么条件下，窗口应该暂停扩大，开始移动 left 缩小窗口？

3、当移动 left 缩小窗口，即移出字符时，应该更新哪些数据？

4、我们要的结果应该在扩大窗口时还是缩小窗口时进行更新？

Tips:

要注意right和left的实时值，因为会提前++；

其他（包括java内部函数）

异或（计不同）：只出现一次的数字，汉明距离

2021年11月15日 星期一 下午 10:01

java内部函数：三大： Map Set List

HashMap<Integer, Integer>
HashSet:
List: ArrayList

```
List<Integer> ret = new ArrayList<Integer>();
```

List. add();
List. length;

```
sites.set(2, "Wiki");
```

```
sites.isEmpty();
sites.remove();
```

```
Map<TreeNode, Integer> memo = new HashMap<>();
```

常用方法：

```
memo.size()
memo.containsKey(root)
memo.get(root)
memo.put(root, Math.max(rootRob,rootNoRob))
memo.getOrDefault(c,0)+1
```

```
Queue<TreeNode> q = new LinkedList<TreeNode>();
Size();
```

String 类 常用方法：

```
s.charAt(index)
s.length()
s.contains(sub); // boolean
s.substring(start, end=边界+1)
Str1.concat(String str2) 2接在1后面
```

Char array 可直接new一个string

```
char[] newS = new char[s.length()];
String newString = new String(newS);
```

或者用stringbuffer

```
StringBuffer sb = new StringBuffer();
for (char j : i){
    sb.append(j);
}
sb.toString()
```

使用 Java 的读者要尤其警惕语言特性的陷阱。Java 的 Integer，String 等类型判定相等应该用 equals 方法而不能直接用等号 ==，这是 Java包装类的一个隐晦细节。所以在左移窗口更新数据的时 候，不能直接改写为 window.get(d) == need.get(d)，而要用 window.get(d).equals(need.get(d))，之后的题目代码同理。

Brian Kernighan 算法：f(x)=x & (x−1)），

那么f(x) 恰为x 删去其二进制的最后一个1。删除为0所用的次数即为1的个数。

数组、链表、队列栈

2021年11月15日 星期一　　下午 10:00

链表 字串 数组： 都是双指针！！！！ ***双指针***

寻找回文串的核心思想是从中心向两端扩展，或者双指针从两头往中间。

回文链表：

借助二叉树后序遍历的思路，不需要显式反转原始链表也可以倒序遍历链表，**然后比较反转链表和原始链表。**

反转链表：

```
ListNode reverse(ListNode head) {
    //base case
    if (head.next == null) return head;
    //后序遍历
    ListNode last = reverse(head.next);
    //本后节点交换顺序
    head.next.next = head;
    head.next = null;

    return last;
}
```

细节上是这么个意思，但是本算法的宏观递归的思维是： head 可能是中间的某节点

1. 反转这个节点的后续链表，并将reverse的头节点赋给last

head -> reverse(head.next)

2. 本后倒序：将后续链表的next指向自己，自己指向null

单链表题型，基本都是用的双指针

1、合并两个有序链表

2、合并 k 个有序链表

3、寻找单链表的倒数第 k 个节点 0, k -> n-k, n

4、寻找单链表的中点 slow, fast

5、判断单链表是否包含环并找出环起点 slow k , fast 2k 环C = k倍

6、判断两个单链表是否相交并找出交点 a+b = b+a --*不能修改链表结构*

图——拓扑排序

2021年12月4日 星期六　　下午 4:31

图存储：

List<Integer>[] graph = new LinkedList[numCourses];

Graph[s] 是一个列表，存着s指向的节点

Union-Find算法

2021年12月14日 星期二   下午 6:11

思想：分门别类，无监督聚类，或者指定一个作为类别始祖。

Union-Find算法主要实现下面这个API：

```
class UF {
    /* 将 p 和 q 连接 */
    public void union(int p, int q);
    /* 判断 p 和 q 是否连通 */
    public boolean connected(int p, int q);
    /* 返回图中有多少个连通分量 */
    public int count();
}
```

1、自反性：节点 p 和 p 是连通的。
2、对称性：如果节点 p 和 q 连通，那么 q 和 p 也连通。
3、传递性：如果节点 p 和 q 连通，q 和 r 连通，那么 p 和 r 也连通。

应用思路：

主要思路是适时增加虚拟节点，想办法让元素「分门别类」，建立动态连通关系。

编译器判断同一个变量的不同引用，比如社交网络中的朋友圈计算等等。

UnionFind实现：数组实现森林

主要时间损耗产生就是在Find()产生的，有两层优化，一个优化连接结构，一个优化层数，双层优化达到最平衡的UF森林。

```
class UF {
    // 连通分量个数
    private int count;
    // 存储一棵树
    private int[] parent;
    // 记录树的「重量」
    private int[] size;

    // n 为图中节点的个数
    public UF(int n) {
        this.count = n;
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    // 将节点 p 和节点 q 连通
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;
```

```
        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        } else {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        }
        // 两个连通分量合并成一个连通分量
        count--;
    }

    // 判断节点 p 和节点 q 是否连通
    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    // 返回节点 x 的连通分量根节点
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
           //find 函数每次向树根遍历的同时，顺手将树高缩短了
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    // 返回图中的连通分量个数
    public int count() {
        return count;
    }
}
```