回溯算法牛逼！

Original labuladong labuladong 2021-03-04 16:30

读完本文，可以去力扣解决如下题目：

698. 划分为k个相等的子集（Medium）

之前说过回溯算法是笔试中最好用的算法，只要你没什么思路，就用回溯算法暴力求解，即便不能通过所有测试用例，多少能过一点。

回溯算法的技巧也不难，前文 回溯算法框架套路 说过，回溯算法就是穷举一棵决策树的过程，只要在递归之前「做选择」，在递归之后「撤销选择」就行了。

但是，就算暴力穷举，不同的思路也有优劣之分。

本文就来看一道非常经典的回溯算法问题，子集划分问题，可以帮你更深刻理解回溯算法的思维，得心应手地写出回溯函数。

题目非常简单：

给你输入一个数组 `nums` 和一个正整数 `k`，请你判断 `nums` 是否能够被平分为元素和相同的 `k` 个子集。

函数签名如下：

```
boolean canPartitionKSubsets(int[] nums, int k);
```

我们之前 背包问题之子集划分 写过一次子集划分问题，不过那道题只需要我们把集

合划分成两个相等的集合，可以转化成背包问题用动态规划技巧解决。

但是如果划分成多个相等的集合，解法一般只能通过暴力穷举，时间复杂度爆表，是练习回溯算法和递归思维的好机会。

一、思路分析

把装有 `n` 个数字的数组 `nums` 分成 `k` 个和相同的集合，你可以想象将 `n` 个数字分配到 `k` 个「桶」里，最后这 `k` 个「桶」里的数字之和要相同。

前文 回溯算法框架套路 说过，回溯算法的关键在哪里？

关键是要知道怎么「做选择」，这样才能利用递归函数进行穷举。

那么回想我们这个问题，将 `n` 个数字分配到 `k` 个桶里，我们可以有两种视角：

视角一，如果我们切换到这 `n` 个数字的视角，每个数字都要选择进入到 `k` 个桶中的某一个。

视角二，如果我们切换到这 `k` 个桶的视角，对于每个桶，都要遍历 `nums` 中的 `n` 个数字，然后选择是否将当前遍历到的数字装进自己这个桶里。

你可能问，这两种视角有什么不同？

用不同的视角进行穷举，虽然结果相同，但是解法代码的逻辑完全不同；对比不同的穷举视角，可以帮你更深刻地理解回溯算法，我们慢慢道来。

二、以数字的视角

用 for 循环迭代遍历 `nums` 数组大家肯定都会：

```
for (int index = 0; index < nums.length; index++) {
```

```
    System.out.println(nums[index]);
}
```

递归遍历数组你会不会？其实也很简单：

```java
void traverse(int[] nums, int index) {
    if (index == nums.length) {
        return;
    }
    System.out.println(nums[index]);
    traverse(nums, index + 1);
}
```

只要调用 `traverse(nums, 0)`，和 for 循环的效果是完全一样的。

那么回到这道题，以数字的视角，选择 `k` 个桶，用 for 循环写出来是下面这样：

```java
// k 个桶（集合），记录每个桶装的数字之和
int[] bucket = new int[k];

// 穷举 nums 中的每个数字
for (int index = 0; index < nums.length; index++) {
    // 穷举每个桶
    for (int i = 0; i < k; i++) {
        // nums[index] 选择是否要进入第 i 个桶
        // ...
    }
}
```

如果改成递归的形式，就是下面这段代码逻辑：

```java
// k 个桶（集合），记录每个桶装的数字之和
int[] bucket = new int[k];

// 穷举 nums 中的每个数字
void backtrack(int[] nums, int index) {
    // base case
    if (index == nums.length) {
        return;
    }
    // 穷举每个桶
    for (int i = 0; i < bucket.length; i++) {
        // 选择装进第 i 个桶
        bucket[i] += nums[index];
        // 递归穷举下一个数字的选择
```

```java
        backtrack(nums, index + 1);
        // 撤销选择
        bucket[i] -= nums[index];
    }
}
```

虽然上述代码仅仅是穷举逻辑，还不能解决我们的问题，但是只要略加完善即可：

```java
// 主函数
public boolean canPartitionKSubsets(int[] nums, int k) {
    // 排除一些基本情况
    if (k > nums.length) return false;
    int sum = 0;
    for (int v : nums) sum += v;
    if (sum % k != 0) return false;

    // k 个桶（集合），记录每个桶装的数字之和
    int[] bucket = new int[k];
    // 理论上每个桶（集合）中数字的和
    int target = sum / k;
    // 穷举，看看 nums 是否能划分成 k 个和为 target 的子集
    return backtrack(nums, 0, bucket, target);
}

// 递归穷举 nums 中的每个数字
boolean backtrack(
    int[] nums, int index, int[] bucket, int target) {

    if (index == nums.length) {
        // 检查所有桶的数字之和是否都是 target
        for (int i = 0; i < bucket.length; i++) {
            if (bucket[i] != target) {
                return false;
            }
        }
        // nums 成功平分成 k 个子集
        return true;
    }

    // 穷举 nums[index] 可能装入的桶
    for (int i = 0; i < bucket.length; i++) {
        // 剪枝，桶装装满了
        if (bucket[i] + nums[index] > target) {
            continue;
        }
        // 将 nums[index] 装入 bucket[i]
        bucket[i] += nums[index];
        // 递归穷举下一个数字的选择
```

```
            if (backtrack(nums, index + 1, bucket, target)) {
                return true;
            }
            // 撤销选择
            bucket[i] -= nums[index];
        }

        // nums[index] 装入哪个桶都不行
        return false;
    }
```

有之前的铺垫，相信这段代码是比较容易理解的。这个解法虽然能够通过，但是耗时比较多，其实我们可以再做一个优化。

主要看 `backtrack` 函数的递归部分：

```
for (int i = 0; i < bucket.length; i++) {
    // 剪枝
    if (bucket[i] + nums[index] > target) {
        continue;
    }

    if (backtrack(nums, index + 1, bucket, target)) {
        return true;
    }
}
```

如果我们让尽可能多的情况命中剪枝的那个 if 分支，就可以减少递归调用的次数，一定程度上减少时间复杂度。

如何尽可能多的命中这个 if 分支呢？要知道我们的 `index` 参数是从 0 开始递增的，也就是递归地从 0 开始遍历 `nums` 数组。

如果我们提前对 `nums` 数组排序，把大的数字排在前面，那么大的数字会先被分配到 `bucket` 中，对于之后的数字， `bucket[i] + nums[index]` 会更大，更容易触发剪枝的 if 条件。

所以可以在之前的代码中再添加一些代码：

```
public boolean canPartitionKSubsets(int[] nums, int k) {
    // 其他代码不变
    // ...
```

```java
    /* 降序排序 nums 数组 */
    Arrays.sort(nums);
    int i = 0, j = nums.length - 1;
    for (; i < j; i++, j--) {
        // 交换 nums[i] 和 nums[j]
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
    /*******************/
    return backtrack(nums, 0, bucket, target);
}
```

由于 Java 的语言特性，这段代码通过先升序排序再反转，达到降序排列的目的。

三、以桶的视角

文章开头说了，**以桶的视角进行穷举，每个桶需要遍历 `nums` 中的所有数字，决定是否把当前数字装进桶中；当装满一个桶之后，还要装下一个桶，直到所有桶都装满为止。**

这个思路可以用下面这段代码表示出来：

```java
// 装满所有桶为止
while (k > 0) {
    // 记录当前桶中的数字之和
    int bucket = 0;
    for (int i = 0; i < nums.length; i++) {
        // 决定是否将 nums[i] 放入当前桶中
        bucket += nums[i] or 0;
        if (bucket == target) {
            // 装满了一个桶，装下一个桶
            k--;
            break;
        }
    }
}
```

那么我们也可以把这个 while 循环改写成递归函数，不过比刚才略微复杂一些，首先写一个 `backtrack` 递归函数出来：

```java
boolean backtrack(int k, int bucket,
    int[] nums, int start, boolean[] used, int target);
```

不要被这么多参数吓到，我会一个个解释这些参数。**如果你能够透彻理解本文，也能得心应手地写出这样的回溯函数。**

这个 `backtrack` 函数的参数可以这样解释：

现在 `k` 号桶正在思考是否应该把 `nums[start]` 这个元素装进来；目前 `k` 号桶里面已经装的数字之和为 `bucket` ； `used` 标志某一个元素是否已经被装到桶中； `target` 是每个桶需要达成的目标和。

根据这个函数定义，可以这样调用 `backtrack` 函数：

```java
public boolean canPartitionKSubsets(int[] nums, int k) {
    // 排除一些基本情况
    if (k > nums.length) return false;
    int sum = 0;
    for (int v : nums) sum += v;
    if (sum % k != 0) return false;

    boolean[] used = new boolean[nums.length];
    int target = sum / k;
    // k 号桶初始什么都没装，从 nums[0] 开始做选择
    return backtrack(k, 0, nums, 0, used, target);
}
```

实现 `backtrack` 函数的逻辑之前，再重复一遍，从桶的视角：

1、需要遍历 `nums` 中所有数字，决定哪些数字需要装到当前桶中。

2、如果当前桶装满了（桶内数字和达到 `target` ），则让下一个桶开始执行第 1 步。

下面的代码就实现了这个逻辑：

```java
boolean backtrack(int k, int bucket,
    int[] nums, int start, boolean[] used, int target) {
    // base case
    if (k == 0) {
        // 所有桶都被装满了，而且 nums 一定全部用完了
```

```java
            // 因为 target == sum / k
            return true;
        }
        if (bucket == target) {
            // 装满了当前桶，递归穷举下一个桶的选择
            // 让下一个桶从 nums[0] 开始选数字
            return backtrack(k - 1, 0 ,nums, 0, used, target);
        }

        // 从 start 开始向后探查有效的 nums[i] 装入当前桶
        for (int i = start; i < nums.length; i++) {
            // 剪枝
            if (used[i]) {
                // nums[i] 已经被装入别的桶中
                continue;
            }
            if (nums[i] + bucket > target) {
                // 当前桶装不下 nums[i]
                continue;
            }
            // 做选择，将 nums[i] 装入当前桶中
            used[i] = true;
            bucket += nums[i];
            // 递归穷举下一个数字是否装入当前桶
            if (backtrack(k, bucket, nums, i + 1, used, target)) {
                return true;
            }
            // 撤销选择
            used[i] = false;
            bucket -= nums[i];
        }
        // 穷举了所有数字，都无法装满当前桶
        return false;
    }
```

至此，这道题的第二种思路也完成了。

四、最后总结

本文写的这两种思路都可以通过所有测试用例，不过第一种解法即便经过了排序优化，也明显比第二种解法慢很多，这是为什么呢？

我们来分析一下这两个算法的时间复杂度，假设 `nums` 中的元素个数为 `n` 。

先说第一个解法，也就是从数字的角度进行穷举， `n` 个数字，每个数字有 `k` 个桶可供选择，所以组合出的结果个数为 `k^n` ，时间复杂度也就是 `O(k^n)` 。

第二个解法，每个桶要遍历 `n` 个数字，选择「装入」或「不装入」，组合的结果有 `2^n` 种；而我们有 `k` 个桶，所以总的时间复杂度为 `O(k*2^n)` 。

当然，这是理论上的最坏复杂度，实际的复杂度肯定要好一些，毕竟我们添加了这么多剪枝逻辑。不过，从复杂度的上界已经可以看出第一种思路要慢很多了。

所以，谁说回溯算法没有技巧性的？虽然回溯算法就是暴力穷举，但穷举也分聪明的穷举方式和低效的穷举方式，关键看你以谁的「视角」进行穷举。

通俗来说，我们应该尽量「少量多次」，就是说宁可多做几次选择，也不要给太大的选择空间；宁可「二选一」选 `k` 次，也不要 「 `k` 选一」选一次。

这道题我们从两种视角进行穷举，虽然代码量看起来多，但核心逻辑都是类似的，相信你通过本文能够更深刻地理解回溯算法。

[精华文章目录点这里](#) 🔗