经典回溯算法：集合划分问题

698. 划分为k个相等的子集

473. 火柴拼正方形

2305. 公平分发饼干

回顾框架

前面我们介绍了「**回溯算法的框架**」「**排列/组合/子集 问题**」「**秒杀所有岛屿题目(DFS)**」

首先我们来复习一下回溯的思想（因为今天的内容很硬核！！！）关于回溯的具体内容可点击上述链接查看

在「回溯算法框架」中给出了解决回溯问题需要思考的 3 个问题：

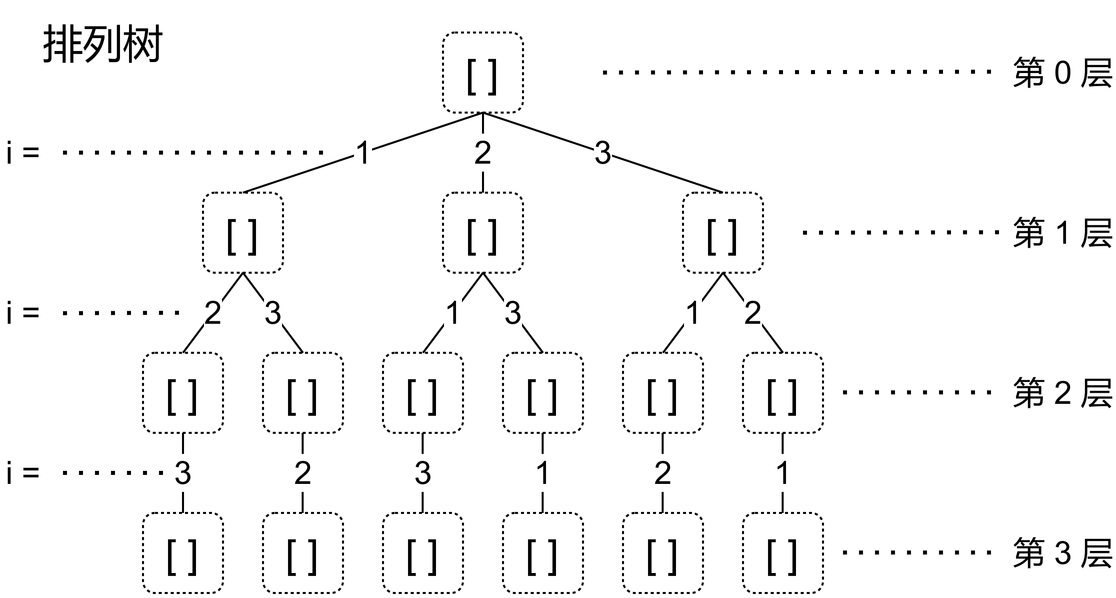
- **路径**：已经做出的选择
- **选择列表**：当前可以做的选择
- **结束条件**：到达决策树底层，无法再做选择的条件

我们先结合下面的决策树，根据「排列」问题来详细分析一下如何理解**「路径」**和**「选择列表」**

- 当我们处于 `第 0 层` 的时候，其实可以做的选择有 3 种，即：选择 1 or 2 or 3
- 假定我们在 `第 0 层` 的时候选择了 1，那么当我们处于 `第 1 层` 的时候，可以做的选择有 2 种，即：2 or 3
- 假定我们在 `第 1 层` 的时候选择了 2，那么当我们处于 `第 2 层` 的时候，可以做的选择有 1 种，即：3
- 当我们到达 `第 3 层` 的时候，我们面前的选择依次为：1，2，3。这正好构成了一个完整的**「路径」**，也正是我们需要的结果

经过上面的分析，我们可以很明显的知道**「结束条件」**，即：所有数都被选择

```
// 结束条件：已经处理完所有数
if (track.size() == nums.length) {
    // 处理逻辑
}
```



引入问题

题目详情可见 划分为k个相等的子集

我们先给出一个样例： `nums = [1, 2, 2, 4, 3, 3], k = 3` ，和题目中的样例不同，下面的所有分析都围绕这个样例展开

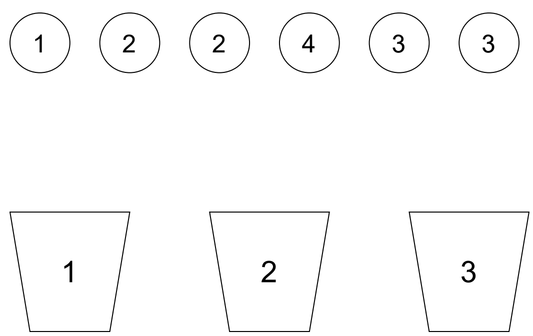
数据预处理

我们先对数据进行预处理，主要就是计算每个子集的和是多少！直接给出代码

```java
// 求总和
int sum = 0;
for (int i = 0; i < nums.length; i++) sum += nums[i];
// 不能刚好分配的情况
if (sum % k != 0) return false;
// target 即每个子集所需要满足的和
int target = sum / k;
```

问题分析

我们先对问题进行一层抽象：有 n 个球，k 个桶，如何分配球放入桶中使得每个桶中球的总和均为 `target` 。如下图所示：



为了可以更好的理解「回溯」的思想，我们这里提供两种不同的视角进行分析对比

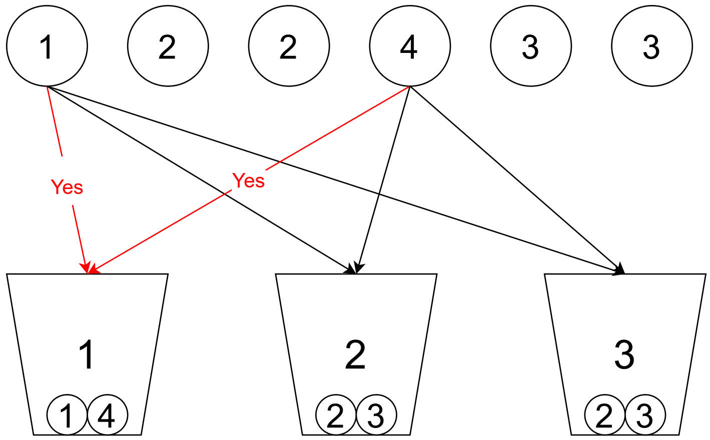
视角一：我们站在球的视角，每个球均需要做出三种选择，即：选择放入 1 号桶、2 号桶、3 号桶。所有球一共需要做 k^n 次选择（分析时间复杂度会用到）

这里提一个点：由于回溯就是一种暴力求解的思想，所以对于每个球的三种选择，只有执行了该选择才知道该选择是否为最优解。说白了就是依次执行这三种选择，如果递归到下面后发现该选择为非最优解，然后开始回溯，执行其他选择，直到把所有选择都遍历完

视角二：我们站在桶的视角，每个桶均需要做出六次选择，即：是否选择 1 号球放入、是否选择 2 号球放入、...、是否选择 6 号球放入。对于一个桶最多需要做 2^n 次选择，所有的桶一共需要做 $(2^n)^k$ 次选择

视角一：球视角

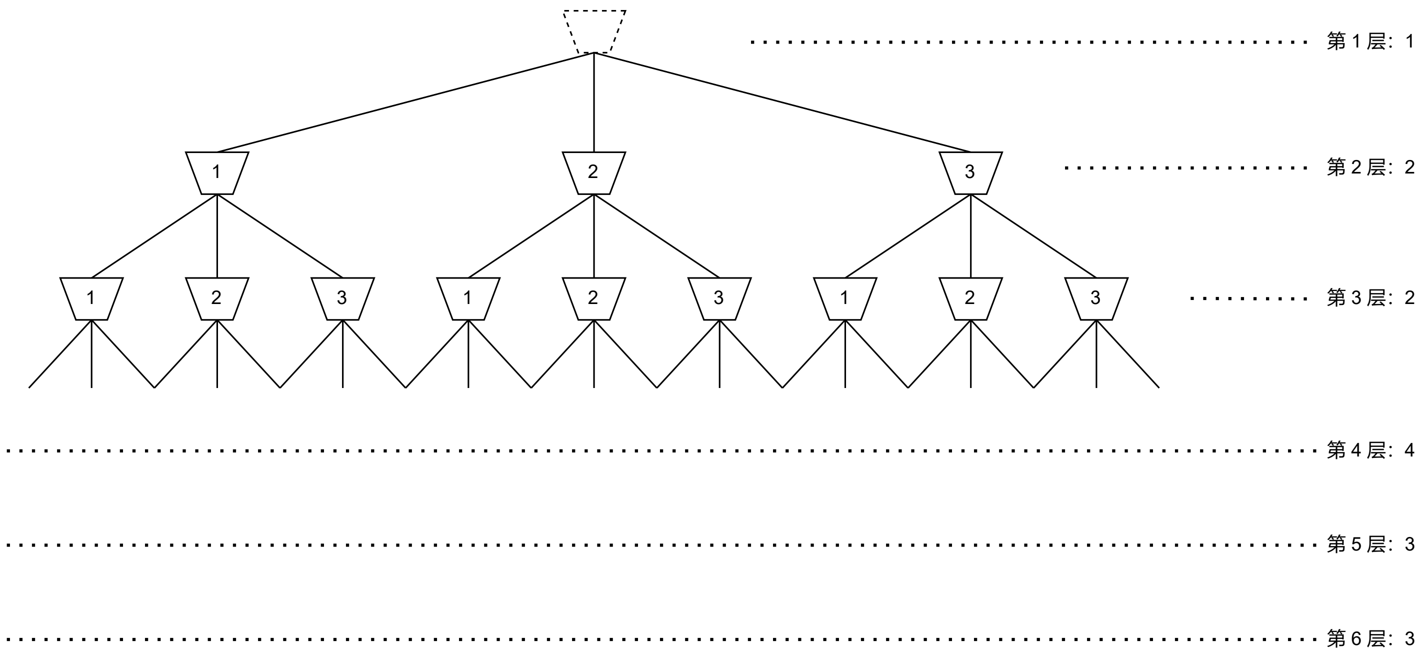
如下图所示，「球」选择「桶」



下面给出「球视角」下的决策树

首先解释一下这棵决策树，`第 i 层` 为 `第 i 个球` 做选择，可做的选择：`选择 1 or 2 or 3 号桶` ，直到 `第 n 个球` 做完选择后结束

由于，每个桶可以放下不止一个球，所以不存在某一个球选择了 1 号桶，另一个球就不能放入 1 号桶。判断是否可以放下的条件为：放入该球后，桶是否溢出？



同样的，根据本文开头给出的框架，详细分析一下如何理解**「路径」**和**「选择列表」**

- 当我们处于 `第 1 层` 的时候，即值为「1」的球开始做选择，可以做的选择有 3 种，即：选择放入 1 or 2 or 3 号桶
- 假定我们在 `第 1 层` 的时候选择了放入 1 号桶，那么当我们处于 `第 2 层` 的时候，即值为「2」的球开始做选择，可以做的选择有 3 种，即：选择放入 1 or 2 or 3 号桶
- 假定我们在 `第 2 层` 的时候选择了放入 1 号桶，那么当我们处于 `第 3 层` 的时候，即值为「2」的球开始做选择，可以做的选择有 3 种，即：选择放入 1 or 2 or 3 号桶
- 假定我们在 `第 3 层` 的时候选择了放入 1 号桶，那么当我们处于 `第 4 层` 的时候，即值为「4」的球开始做选择，可以做的选择有 2 种，即：选择放入 2 or 3 号桶（原因：1 号桶放入了 1 2 2，已经满了）
- 假定我们在 `第 4 层` 的时候选择了放入 2 号桶，那么当我们处于 `第 5 层` 的时候，即值为「3」的球开始做选择，可以做的选择有 1 种，即：选择放入 3 号桶（原因：2 号桶放入了 4，容纳不下 3 了）
- 假定我们在 `第 5 层` 的时候选择了放入 3 号桶，那么当我们处于 `第 6 层` 的时候，即值为「3」的球开始做选择，可以做的选择有 0 种（原因：3 号桶放入了 3，容纳不下 3 了）

- 此时我们已经到达了最后一层！！我们来梳理一下选择的路径，即：「1 号桶：1 2 2」「 2 号桶：4」「3 号桶：3」。显然这条路径是不符合要求的，所以就开始回溯，回溯到 `第 5 层` ，改变 `第 5 层` 的选择，以此类推，直到得出**「最优解」**

经过上面的分析，我们可以很明显的知道**「结束条件」**，即：所有球都做了选择后结束

```
// 结束条件：已经处理完所有球
if (index == nums.length) {
    // 处理逻辑
}
```

这里来一个小插曲。根据上面的分析可以知道，其实我们每一层做选择的球都是按顺序执行的。我们可以很容易的用迭代的方法遍历一个数组，那么如何递归的遍历一个数组呢？？？

```
// 迭代
private void traversal(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        System.out.println(nums[i]);
    }
}

// 递归
private void traversal(int[] nums, int index) {
    if (index == nums.length) return ;
    // 处理当前元素
    System.out.println(nums[i]);
    // 递归处理 index + 1 后的元素
    traversal(nums, index + 1);
}
traversal(nums, 0);
```

第一版代码

好了，下面给出第一版代码：**（温馨提示：结合注释以及上面的分析一起看，便于理解，整个流程高度吻合）**

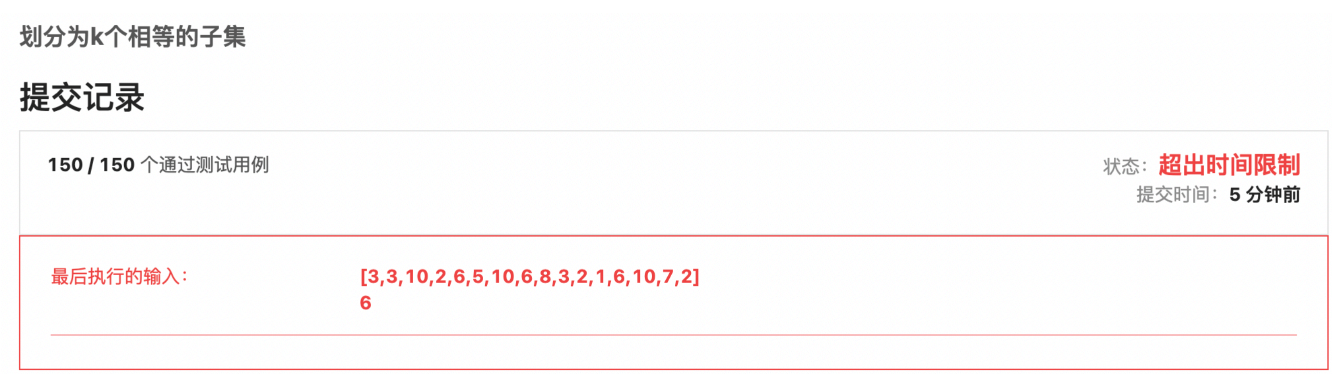
```
public boolean canPartitionKSubsets(int[] nums, int k) {
    int sum = 0;
    for (int i = 0; i < nums.length; i++) sum += nums[i];
    if (sum % k != 0) return false;
    int target = sum / k;
    int[] bucket = new int[k + 1];
    return backtrack(nums, 0, bucket, k, target);
}
// index : 第 index 个球开始做选择
// bucket : 桶
private boolean backtrack(int[] nums, int index, int[] bucket, int k, int target) {

    // 结束条件：已经处理完所有球
    if (index == nums.length) {
```

```
        // 判断现在桶中的球是否符合要求 -> 路径是否满足要求
        for (int i = 0; i < k; i++) {
            if (bucket[i] != target) return false;
        }
        return true;
    }

    // nums[index] 开始做选择
    for (int i = 0; i < k; i++) {
        // 剪枝：放入球后超过 target 的值，选择一下桶
        if (bucket[i] + nums[index] > target) continue;
        // 做选择：放入 i 号桶
        bucket[i] += nums[index];

        // 处理下一个球，即 nums[index + 1]
        if (backtrack(nums, index + 1, bucket, k, target)) return true;

        // 撤销选择：挪出 i 号桶
        bucket[i] -= nums[index];
    }

    // k 个桶都不满足要求
    return false;
}
```

这里有一个好消息和一个坏消息，想先听哪一个呢？？哈哈哈哈哈

好消息：代码没问题

坏消息：超时没通过


划分为k个相等的子集

提交记录

150 / 150 个通过测试用例

状态：超出时间限制

提交时间：5 分钟前

最后执行的输入：

[3,3,10,2,6,5,10,6,8,3,2,1,6,10,7,2]
6

回到最上面的分析 → 跳转，我们必须一直回溯到 `第 2 层`，让第一个值为「2」的球选择 2 号桶，才更接近我们的最优解，其他的以此类推！

现在超时的原因就很明显了，由于我们的时间复杂度为 $O(k^n)$，呈指数增加，直接爆掉了

第一次尝试剪枝

我们有一个优化的思路，先看剪枝部分的代码：

```java
// 剪枝：放入球后超过 target 的值，选择一下桶
if (bucket[i] + nums[index] > target) continue;
```

如果我们让 `nums[]` 内的元素递减排序，先让值大的元素选择桶，这样可以增加剪枝的命中率，从而降低回溯的概率

```java
public boolean canPartitionKSubsets(int[] nums, int k) {
    // 其余代码不变

    // 降序排列
    Arrays.sort(nums);
    int left = 0, right= nums.length - 1;
    while (left < right) {
        int temp = nums[left];
        nums[left] = nums[right];
        nums[right] = temp;
        left++;
        right--;
    }

    return backtrack(nums, 0, bucket, k, target);
}
```

很遗憾，还是超时，**但肯定比第一版的快点**

其实主要原因还是在于这种思路的时间复杂度太高，无论怎么优化，还是很高！！！直接 $O(k^n)$，这谁顶得住呀！！！

第二次尝试剪枝

🔥🔥🔥 **发现新大陆！！！**

突然看到了一种新的「剪枝」思路，这个剪枝绝了，不得不更新一下文章

首先需要优化的第一个点：

```java
// 结束条件：已经处理完所有球
if (index == nums.length) {
    // 有人提出，其实这个地方不需要判断，因为当 index == num.length 时，所有球已经按要求装入所有桶，所以肯定是一个满足要求的解
    // 即：每个桶内球的和一定为 target
    /** // 判断现在桶中的球是否符合要求 -> 路径是否满足要求
    for (int i = 0; i < k; i++) {
        if (bucket[i] != target) return false;
    }**/
    return true;
}
```

其次可以优化的第二个点和 **排列/组合/子集 问题** 中「元素可重不可复选」情况下「子集」的处理情况很相似！！！

```java
for (int i = 0; i < k; i++) {
    // 如果当前桶和上一个桶内的元素和相等，则跳过
    // 原因：如果元素和相等，那么 nums[index] 选择上一个桶和选择当前桶可以得到的结果是一致的
    if (i > 0 && bucket[i] == bucket[i - 1]) continue;
    // 其他逻辑不变
}
```

最后可以优化的第三个点，对于第一个球，任意放到某个桶中的效果都是一样的，所以我们规定放到第一个桶中

```java
for (int i = 0; i < k; i++) {
    if (i > 0 && index == 0) break ;
    // 其他逻辑不变
}
```

刚刚有个小伙伴指出，其实「优化点二」已经包含了「优化点三」

第一个球选择每一个桶时，每个桶内元素和都为 0。所以，当第一个球选择第二个桶以及后续桶的时候，就会因为「优化点二」而跳过，间接的实现了「优化点三」中所说的第一个球放到第一个桶内的规定！！

最终优化代码（包含所有优化）

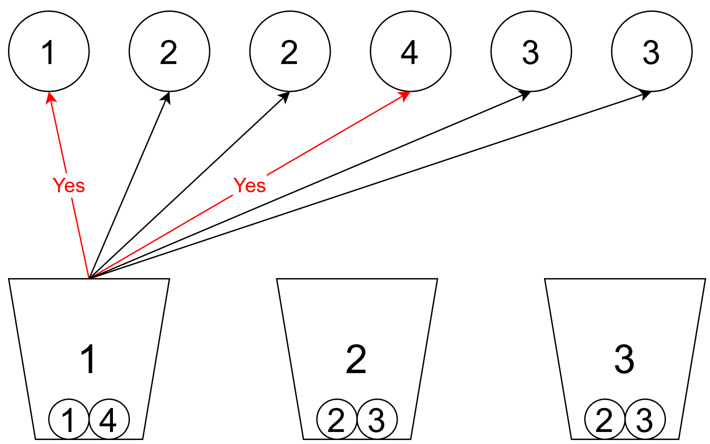
```java
public boolean canPartitionKSubsets(int[] nums, int k) {
    int sum = 0;
    for (int i = 0; i < nums.length; i++) sum += nums[i];
    if (sum % k != 0) return false;
    int target = sum / k;
    // 排序优化
    Arrays.sort(nums);
    int l = 0, r = nums.length - 1;
    while (l <= r) {
        int temp = nums[l];
        nums[l] = nums[r];
        nums[r] = temp;
        l++;
        r--;
    }
    return backtrack(nums, 0, new int[k], k, target);
}
private boolean backtrack(int[] nums, int index, int[] bucket, int k, int target) {
    // 结束条件优化
    if (index == nums.length) return true;
    for (int i = 0; i < k; i++) {
        // 优化点二
        if (i > 0 && bucket[i] == bucket[i - 1]) continue;
        // 剪枝
        if (bucket[i] + nums[index] > target) continue;
        bucket[i] += nums[index];
        if (backtrack(nums, index + 1, bucket, k, target)) return true;
```

```
            bucket[i] -= nums[index];
        }
        return false;
    }
```

下面给出优化后的执行时间：（惊不惊喜，意不意外！！！！）

视角二：桶视角

现在来介绍另外一种视角，如下图所示，「桶」选择「球」



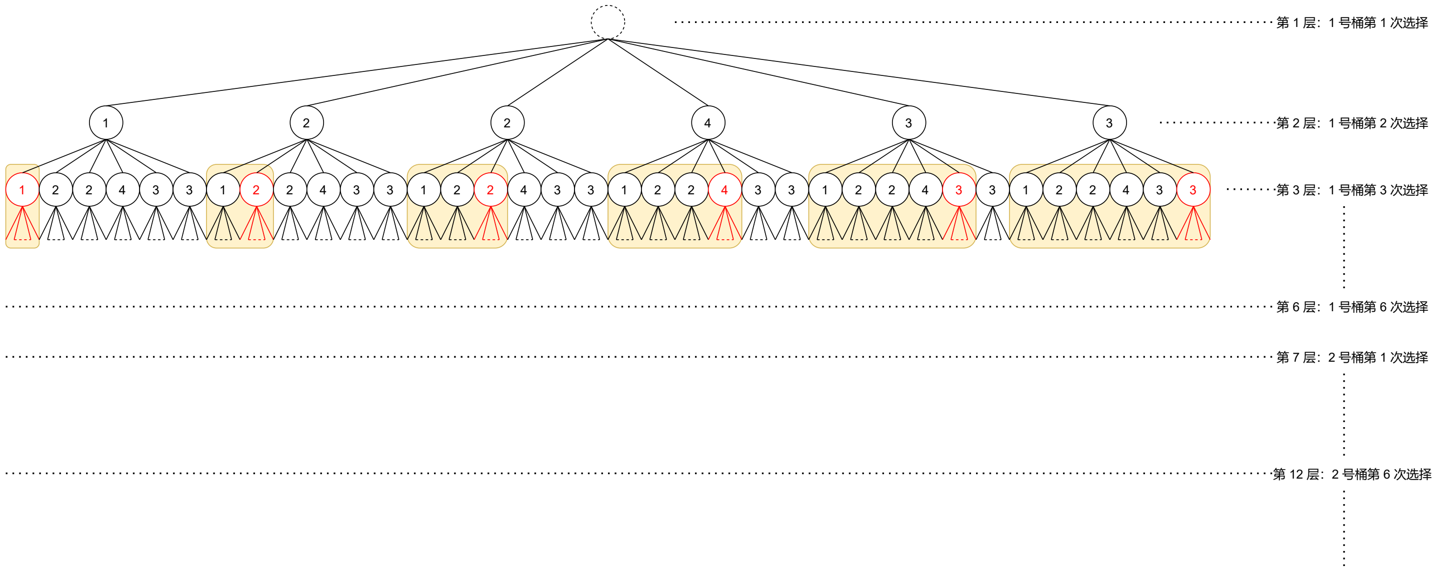
下面给出「桶视角」下的决策树

首先解释一下这棵决策树，`第 i 层` 为 `j 号桶` 做出 `第 x 次选择` ，可做的选择：`是否选择 1 ~ 6 号球` ，直到 `k 个桶` 均装满后结束

由于，每个球只能被一个桶选择，所以当某一个球被某一个桶选择后，另一个桶就不能选择该球，如下图红色标注出的分支

判断是否可以选择某个球的条件为：（1）该球是否已经被选择？ （2）放入该球后，桶是否溢出？

这里还需要强调的一点是，我们是根据每个桶可以做的最多次选择来绘制的决策树，即 6 次选择，但在实际中可能经过两三次选择后桶就装满了，然后下一个桶开始选择。之所以会有 6 次选择，是因为可能在后面回溯的过程过中进行其他选择

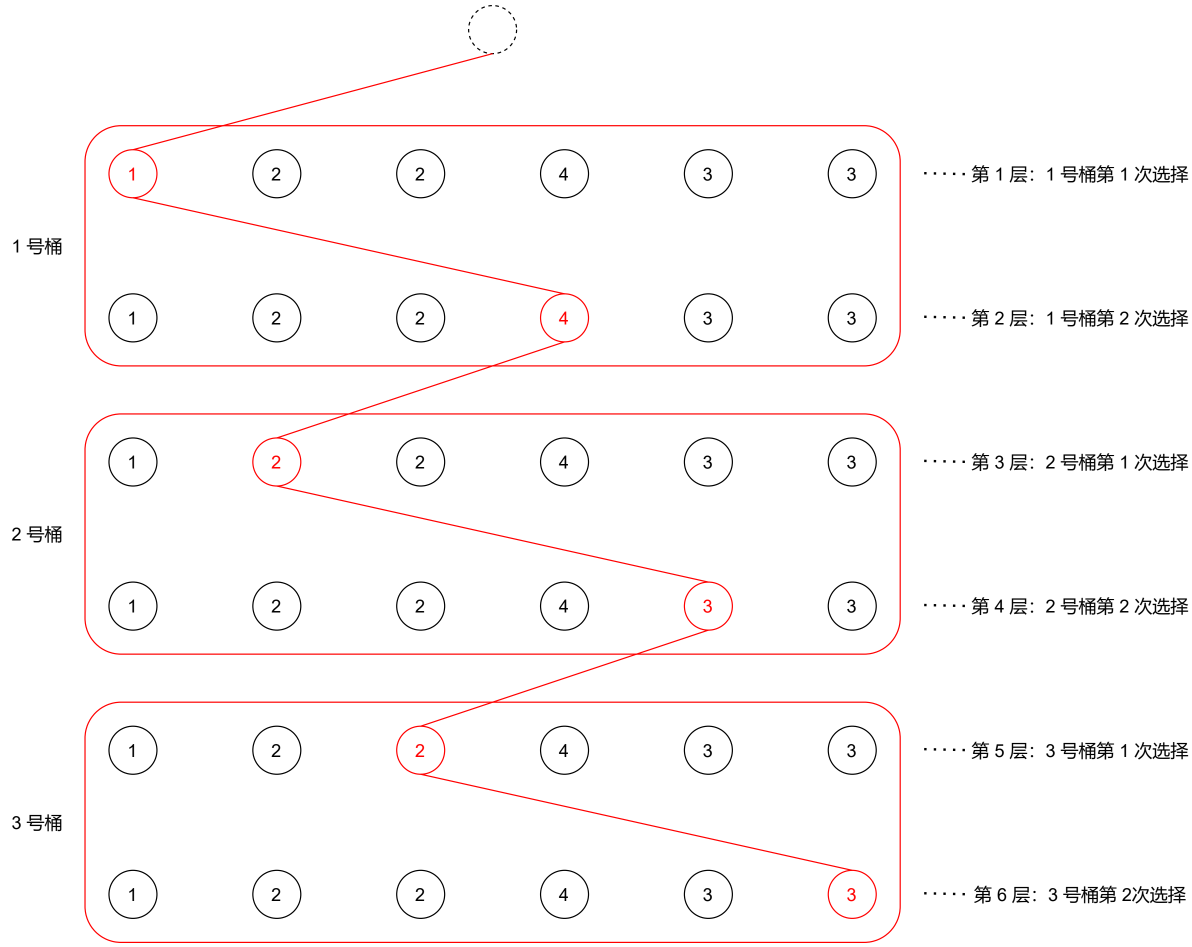


其中黄色框出来的分支是当前桶选择的冗余分支！！！举个简单例子，「1」号桶开始先选值为「1」的球，再选值为「2」的球 和「1」号桶开始先选值为「2」的球，再选值为「1」的球是等价的，因为没有顺序的约束！！

同样的，根据本文开头给出的框架，详细分析一下如何理解**「路径」**和**「选择列表」**

- 当我们处于 `第 1 层` 的时候，即「1」号桶开始第「1」次选择，可以做的选择有 6 种，即：选择值为「1 or 2 or 2 or 4 or 3 or 3」的球
- 假定我们在 `第 1 层` 的时候「1」号桶选择了值为「1」的球，那么当我们处于 `第 2 层` 的时候，即「1」号桶开始第「2」次选择，可以做的选择有 5 种，即：选择值为「2 or 2 or 4 or 3 or 3」的球
- 假定我们在 `第 2 层` 的时候「1」号桶选择了值为「2」的球，那么当我们处于 `第 3 层` 的时候，即「1」号桶开始第「3」次选择，可以做的选择有 4 种，即：选择值为「2 or 4 or 3 or 3」的球
- 假定我们在 `第 3 层` 的时候「1」号桶选择了值为「2」的球，那么当我们处于 `第 4 层` 的时候，开始下一个桶开始选择（原因：1 号桶选择了 1 2 2，已经满了）即「2」号桶开始第「1」次选择，可以做的选择有 3 种，即：选择值为「4 or 3 or 3」的球
- ......
- 开始回溯......以此类推，直到得出**「最优解」**

假定得到了**「最优解」**，我们来梳理一下此时选择的路径，即：「1 号桶：1 4」「 2 号桶：2 3」「3 号桶：2 3」，具体如下图所示：



经过上面的分析，我们可以很明显的知道**「结束条件」**，即：所有均装满后结束

```
if (k == 0) {
    // 处理逻辑
}
```

现在我们再次回到本文最开始复习的「回溯」框架需要思考的三个问题，即：**「路径」「选择列表」「结束条件」**

有没有发现一个很有意思的现象，这难道不是和求「树的所有从根到叶子节点的路径」如出一辙嘛！！不信的话可以先去写一下 **257. 二叉树的所有路径**

我们先复习一下求「树的所有路径」的思路

- **选择列表：**每次我们都可以选择当前节点的「左孩子」或「右孩子」
- **结束条件：**遇到叶子节点
- **路径：**在遍历过程中记录我们所有的选择的一条路

回到「回溯」问题上，相比于「树的所有路径」

- 只不过这棵树需要我们自己抽象出来而已，即「决策树」
- 只不过结束条件需要我们根据题目意思自己确定而已
- 只不过我们需要的是一条「最优解」路径，即：从所有路径中得到最优解路径
- 同时，我们需要通过「剪枝」来减少对「决策树」的递归遍历而已

第一版代码

好了，下面给出第一版代码：**（温馨提示：结合注释以及上面的分析一起看，便于理解，整个流程高度吻合）**

```java
private boolean backtrack(int[] nums, int start, int[] bucket, int k, int target, boolean[] used) {
    // k 个桶均装满
    if (k == 0) return true;
    // 当前桶装满了，开始装下一个桶
    if (bucket[k] == target) {
        // 注意：桶从下一个开始；球从第一个开始
        return backtrack(nums, 0, bucket, k - 1, target, used);
    }
    // 第 k 个桶开始对每一个球选择进行选择是否装入
    for (int i = start; i < nums.length; i++) {
        // 如果当前球已经被装入，则跳过
        if (used[i]) continue;
        // 如果装入当前球，桶溢出，则跳过
        if (bucket[k] + nums[i] > target) continue;

        // 装入 && 标记已使用
        bucket[k] += nums[i];
        used[i] = true;

        // 开始判断是否选择下一个球
        // 注意：桶依旧是当前桶；球是下一个球
        // 注意：是 i + 1
        if (backtrack(nums, i + 1, bucket, k, target, used)) return true;

        // 拿出 && 标记未使用
        bucket[k] -= nums[i];
        used[i] = false;
    }
    // 如果所有球均不能使所有桶刚好装满
    return false;
}
```

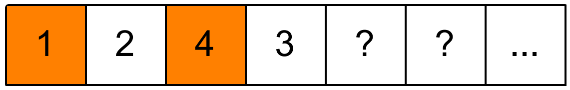
可是可是可是，虽然可以过，但是执行时间吓死个人！！！

第一次尝试优化

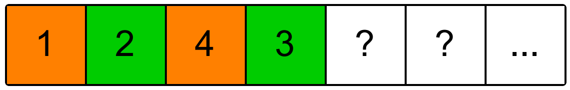
前文分析过该视角下的时间复杂度为 $(2^n)^k$。其实我们上面的代码在递归的过程中存在很多冗余的计算，导致超时

现在我们假设一种情况，`num = {1, 2, 4, 3, ......}, target = 5`

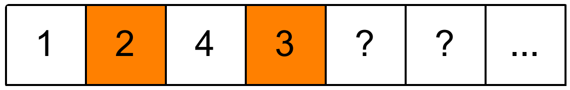
第一个桶会首先选择 `1 4`。如下图橙色所示



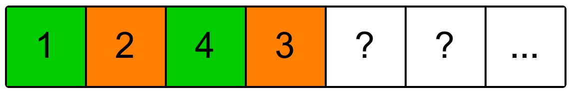
第二个桶会选择 2 3。如下图绿色所示



现在假设后面的元素无法完美组合成目标和，程序会进行回溯！！假设当前回溯到了「1」号桶开始第「1」次选择，故「1」号桶的第「1」次选择会发生改变 1 → 2。如下图橙色所示



接着第二个桶的选择也会改变。如下图绿色所示



显然，虽然这一次的回溯结果中「1」号桶和「2」号桶选择的元素发生了改变，但是它们组合起来的选择没有变化，依旧是 1 2 4 3，剩下的元素未发生改变，所以依旧无法完美组合成目标和

如果我们把这样的组合记录下来，下次遇到同样的组合则直接跳过。那如何记录这种状态呢？？？ ⟶ **借助 `used[]` 数组**

可以看到上述四张图片中的 `used[]` 状态分别为：

- 图片一：「true, false, true, false, false, ...」
- 图片二：「true, true, true, true, false, ...」
- 图片三：「false, true, false, true, false, ...」
- 图片四：「true, true, true, true, false, ...」

第一次优化代码如下：

```java
// 备忘录，存储 used 数组的状态
private HashMap<String, Boolean> memo = new HashMap<>();
private boolean backtrack(int[] nums, int start, int[] bucket, int k, int target, boolean[] used) {
    // k 个桶均装满
    if (k == 0) return true;

    // 将 used 的状态转化成形如 [true, false, ...] 的字符串
    // 便于存入 HashMap
    String state = Arrays.toString(used);

    // 当前桶装满了，开始装下一个桶
    if (bucket[k] == target) {
        // 注意：桶从下一个开始；球从第一个开始
        boolean res = backtrack(nums, 0, bucket, k - 1, target, used);
        memo.put(state, res);
        return res;
    }

    if (memo.containsKey(state)) {
        // 如果当前状态曾今计算过，就直接返回，不要再递归穷举了
```

```java
            return memo.get(state);
        }

        // 其他逻辑不变！！
    }
```

虽然耗时少了很多，但效率依然是比较低

通过	607 ms	44.3 MB	Java	2022/05/05 20:17	添加备注

第二次尝试优化

这次不是因为算法逻辑上的冗余计算，而是代码实现上的问题

因为每次递归都要把 `used` 数组转化成字符串，这对于编程语言来说也是一个不小的消耗，所以我们还可以进一步优化

结合题目意思，可以知道 `1 <= len(nums) <= 16`，所以我们可以用 16 位二进制来记录元素的使用情况，即：如果第 i 个元素使用了，则第 i 位二进制设为 1，否则为 0

关于位运算技巧，**详情可见 位运算技巧**

下面给出第二次优化代码（最终完整代码），如下：

```java
// 备忘录，存储 used 的状态
private HashMap<Integer, Boolean> memo = new HashMap<>();

public boolean canPartitionKSubsets(int[] nums, int k) {
    int sum = 0;
    for (int i = 0; i < nums.length; i++) sum += nums[i];
    if (sum % k != 0) return false;
    int target = sum / k;
    // 使用位图技巧
    int used = 0;
    int[] bucket = new int[k + 1];
    return backtrack(nums, 0, bucket, k, target, used);
}
private boolean backtrack(int[] nums, int start, int[] bucket, int k, int target, int used) {
    // k 个桶均装满
    if (k == 0) return true;

    // 当前桶装满了，开始装下一个桶
    if (bucket[k] == target) {
        // 注意：桶从下一个开始；球从第一个开始
        boolean res = backtrack(nums, 0, bucket, k - 1, target, used);
        memo.put(used, res);
        return res;
    }

    if (memo.containsKey(used)) {
        // 如果当前状态曾今计算过，就直接返回，不要再递归穷举了
```

```
            return memo.get(used);
        }

        // 第 k 个桶开始对每一个球选择进行选择是否装入
        for (int i = start; i < nums.length; i++) {
            // 如果当前球已经被装入，则跳过
            if (((used >> i) & 1) == 1) continue;
            // 如果装入当前球，桶溢出，则跳过
            if (bucket[k] + nums[i] > target) continue;

            // 装入 && 标记已使用
            bucket[k] += nums[i];
            // 将第 i 位标记为 1
            used |= 1 << i;

            // 开始判断是否选择下一个球
            // 注意：桶依旧是当前桶；球是下一个球
            // 注意：是 i + 1
            if (backtrack(nums, i + 1, bucket, k, target, used)) return true;

            // 拿出 && 标记未使用
            bucket[k] -= nums[i];
            // 将第 i 位标记为 10
            used ^= 1 << i;
        }
        // 如果所有球均不能使所有桶刚好装满
        return false;
    }
```

至此，终于还算完美的通过了，太不容易了！！！😭😭😭😭

牛的不行的剪枝

🔥🔥🔥 **再次发现新大陆！！！**

刚刚有个小伙伴给出了一种新的剪枝策略，在这里感谢一下这位小伙伴 `@yttttt-e`

先看代码：（温馨提醒：前提是数组需要有序）

```java
for (int i = start; i < nums.length; i++) {
    // 其他逻辑不变 ....
    used ^= 1 << i;

    // 下一个如果和当前的相同肯定也不行
    while (i + 1 < nums.length && nums[i + 1] == nums[i]) i++;
}
```

解释一下是什么意思！当我们在处理第 `i` 个球的时候发现无法满足要求，如果这个时候下一个球和当前球的值是一样的，那么我们就可以直接跳过下一个球

按照惯例看一下耗时：（有很大的提高！！）

时间复杂度分析

对于两种不同视角下的时间复杂度，前文也给出了简约的分析！！ $\rightarrow$ **Link**

对于视角一（球视角）和视角二（桶视角），前者为 $O(k^n)$，后者为 $O((2^n)^k)$。其实差距还是挺大的，尤其是当 k 越大时，这种差距越明显！！

现在结合回溯的每一次选择分析时间复杂度，**「尽可能让每一次的可选择项少」**才能使时间复杂度降低维度！！

- 对于视角一（球视角），每个球每一次的可选择项都为「所有桶」，所以每一次可选择项均为桶的数量 k。故时间复杂度指数的底数为 k
- 对于视角二（桶视角），每个桶每一次的可选择项都为「是否装入某个球」，所以每一次可选择项均为 `n` 个球「装入」or「不装入」。故时间复杂度指数的底数为 2^n

所以，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 `n` 次「 `k` 选一」$\rightarrow O(k^n)$，也不要 `k` 次「 2^n 选一」$\rightarrow O((2^n)^k)$