子集问题

> 输入一个不包含重复数字的数组，要求算法输出这些数字的所有子集。

分析：

1. 对子集类回溯问题，在回溯时，需要针对数组所有元素从头到尾遍历，每个结果子集直接加入到结果集中即可。

```
class Solution {
public:
  std::vector<std::vector<int>> subsets(std::vector<int>& nums) {
    backtrace(nums, 0);

    return res;
  }

private:
  void backtrace(std::vector<int>& nums, int index) {
    res.push_back(subset);

    for (int i = index; i < nums.size(); i++) {
      // 做选择
      subset.push_back(nums[i]);
      backtrace(nums, i + 1);
      // 撤销选择
      subset.pop_back();
    }
  }

  std::vector<int>              subset;
  std::vector<std::vector<int>> res;
};
```

> 输入一个包含重复数字的数组，要求算法输出这些数字的所有子集。

分析：

1. 首先需要对数组进行排序，方便后续计算；
2. 与无重复元素计算类似，直接将子集加入结果中；
3. 在遍历时需要跳过重复元素；重复元素跳过模式固定代码为如下方式：

```
// index 从0开始
for(int i = index;i<len;i++) {
    if(i > index && nums[i-1] == nums[i]){
```

```
            continue;
        }
    }
```

算法实现

```
class Solution {
public:
    std::vector<std::vector<int>> subsetsWithDup(std::vector<int>& nums) {
        std::sort(nums.begin(), nums.end());
        backtrack(nums, 0);
        return res;
    }

private:
    void backtrack(std::vector<int>& nums, int index) {
        res.push_back(subset);

        for (int i = index; i < nums.size(); i++) {
            // 跳过重复元素
            if (i > index && nums[i] == nums[i - 1]) {
                continue;
            }
            subset.push_back(nums[i]);  // 做选择
            backtrack(nums, i + 1);
            subset.pop_back();  // 撤销选择
        }
    }

    std::vector<int>              subset;  // 记录走过的路
    std::vector<std::vector<int>> res;     // 记录结果
};
```

组合问题

输入两个数字 n, k，算法输出 [1..n] 中 k 个数字的所有组合。

分析： 回溯递归终止条件，以及子集加入结果集的方式不同。

```
class Solution {
public:
    std::vector<std::vector<int>> combine(int n, int k) {
        backtrace(n, k, 1);
        return res;
    }

private:
```

```
  void backtrace(int n, int k, int index) {
    if (subset.size() == k) {
      // 满足条件加入结果集
      res.push_back(subset);
      return;
    }

    for (int i = index; i <= n; i++) {
      // 做选择
      subset.push_back(i);
      backtrace(n, k, i + 1);
      // 撤销选择
      subset.pop_back();
    }
  }

  std::vector<int>              subset;  // 遍历路径
  std::vector<std::vector<int>> res;     // 结果集
};
```

组合数和

1. 数组元素可以重复使用，每个数字可以使用多次。

```
class Solution {
public:
  std::vector<std::vector<int>> combinationSum(std::vector<int>
&candidates,
                                               int               target) {
    std::sort(candidates.begin(), candidates.end());

    sum = 0;
    backtrace(candidates, 0, target, sum);
    return res;
  }

private:
  void backtrace(std::vector<int> &candidates,
                 int               index,
                 int               target,
                 int &             sum) {
    if (sum > target) {
      return;
    }

    if (sum == target) {
      res.push_back(subset);
      return;
    }

    for (int i = index; i < candidates.size(); i++) {
```

```
      if (sum + candidates[i] > target) {
        continue;
      } else {
        // 做选择
        sum += candidates[i];
        subset.push_back(candidates[i]);
        // 递归，因为元素可以重复，所以递归开始字段仍为当前索引
        backtrace(candidates, i, target, sum);

        // 撤销选择
        sum -= candidates[i];
        subset.pop_back();
      }
    }
  }

  std::vector<int>              subset;  // 遍历路径
  int                           sum;     // 路径上元素和
  std::vector<std::vector<int>> res;
};
```

2. 数组元素不可重复使用，每个数字使用一次。

```
class Solution {
public:
  std::vector<std::vector<int>> combinationSum2(std::vector<int>
&candidates,
                                                int               target)
{
    std::sort(candidates.begin(), candidates.end());
    sum = 0;
    backtrace(candidates, 0, target, sum);
    return res;
  }

private:
  void backtrace(std::vector<int> &nums, int index, int target, int &sum)
{
    if (target < sum) {
        return;
    }

    if (target == sum) {
      res.push_back(subset);
      return;
    }

    for (int i = index; i < nums.size(); i++) {
      // 去重
      if (i > index && nums[i] == nums[i - 1]) {
        continue;
      }
```

```
      if (sum + nums[i] > target) {
        // 当前和大于target, 不做处理
        continue;
      } else {
        // 加入结果集
        sum += nums[i];
        // 做选择
        subset.push_back(nums[i]);
        backtrace(nums, i + 1, target, sum);

        // 撤销选择
        sum -= nums[i];
        subset.pop_back();
      }
    }
  }

  std::vector<int>              subset;  // 走过的路径
  int                           sum;     // 路径中所有元素之和
  std::vector<std::vector<int>> res;     // 结果集
};
```

通过上述两个例子，可以总结出一个结论：对于路径类问题: 对于数据可以重复使用的例子，在进行递归时起点仍为当前元素。 不可重复使用则递归起点为下一个元素。 对路径类，子集类和组合类问题： 数据查找的循环从指定的index开始，不是从头开始。 因为其每次查找的结果集均是给定数据集的一部分，不是数据集的全部。

排列类问题

1. 无重复数组全排列

```
class Solution {
public:
  std::vector<std::vector<int>> permute(std::vector<int>& nums) {
    int len = nums.size();
    visited = std::vector<bool>(len + 1, false);
    backtrace(nums, len);
    return res;
  }

private:
  void backtrace(std::vector<int>& nums, int len) {
    if (track.size() == len) {
      res.push_back(track);
      return;
    }

    for (int i = 0; i < len; i++) {
      // 是否已遍历过，如果已遍历过，继续循环
      if (visited[i]) {
```

```
        continue;
      }

      // 做选择
      visited[i] = true;
      track.push_back(nums[i]);
      backtrace(nums, len);
      // 撤销选择
      visited[i] = false;
      track.pop_back();
    }
  }

  std::vector<int>              track;    // 遍历路径
  std::vector<std::vector<int>> res;      // 遍历结果集
  std::vector<bool>             visited;  // 是否遍历过标记
};
```

2. 有重复数组全排列

```
class Solution {
public:
  std::vector<std::vector<int>> permuteUnique(std::vector<int>& nums) {
    std::sort(nums.begin(), nums.end());
    int len = nums.size();
    visited = std::vector<bool>(len + 1, false);
    backtrace(nums, len);
    return res;
  }

private:
  void backtrace(std::vector<int>& nums, int len) {
    if (track.size() == len) {
      res.push_back(track);
      return;
    }

    for (int i = 0; i < len; i++) {
      if (visited[i]) {
        continue;
      }

      if (i > 0 && !visited[i - 1] && nums[i] == nums[i - 1]) {
        continue;
      }

      // 做选择
      visited[i] = true;
      track.push_back(nums[i]);
      backtrace(nums, len);

      // 撤销选择
```

```
        visited[i] = false;
        track.pop_back();
      }
    }

    std::vector<bool>              visited;
    std::vector<int>               track;  // 遍历路径
    std::vector<std::vector<int>> res;    // 结果集
};
```

总结

1. 对子集，路径和，组合等回溯问题，每次求解都是给定数据集的部分或全部数据，所以其搜索范围为**[index...len-1]**;
2. 如果在回溯的过程中，元素可以反复使用，则下一次递归的起点仍为当前元素，否则为下一个元素；
3. 对排列等问题，每次求解的结果均为给定数据的全部元素，所以每次循环都是 **[0..len]**,同时需要标记上一个元素是否已经被使用过`visited`,使用过就跳过，否则从当前值开始回溯；
4. 对这两类问题，对重复元素的处理不同；当相同点是回溯之前均需要排序；
5. 对重复元素处理：
    - 子集，路径和等问题，只需判断当前元素是否与前一个元素相等即可；

```
for(int i = start; i < len; i++) {
    if(i>0 && nums[i] == nums[i-1]){
        continue;
    }
}
```

- 对全排列类问题，存在状态数组`visited`, 仅当前一个相同元素未使用过时跳过。

```
for (int i =0;i<len;i++) {
    if (i > 0 && !visited[i-1] && nums[i] == nums[i-1]) {
        continue;
    }
}
```

补充： 求数组的下一个全排列： 分析：

1. 从右向左查找非递减序列(从左到右为递减序列)，记这个非递减序列的第一个数的索引为`index`, 该数前一个元素索引记为`indexpre`;
2. 如果`index < 0`不存在下一个序列直接返回；
3. 从`index`之后的非递减序列中，查找最小值,记其索引为`indexmin`, 该值满足在大于`nums[indexpre]`的数值中最小；
4. 交换`nums[indexmin], nums[indexpre]`，就得到了下一个排列。

```cpp
class Solution {
public:
  void nextPermutation(std::vector<int>& nums) {
    int n = nums.size(), i, j;
    for (i = n - 2; i >= 0; i--) {
      if (nums[i] < nums[i + 1]) {
        break;
      }
    }
    if (i < 0) {
      std::reverse(nums.begin(), nums.end());
      return;
    } else {
      for (j = n - 1; j > i; j--) {
        if (nums[j] > nums[i]) {
          break;
        }
      }
      std::swap(nums[i], nums[j]);
      std::reverse(nums.begin() + i + 1, nums.end());
      return;
    }
  }
};
```