# 4 Backtracking

## 4.1 problem list

- 来源列表：LeetCode – 回溯算法。
- 这些题的难度标签很多不太准确，大多实为不太难的题目。

| id | title | label |
| --- | --- | --- |
| 1 | p401 二进制手表 easy | |
| 2 | p78 子集 normal | |
| 3 | p22 括号生成 normal | |
| 4 | p46 全排列 normal | |
| 5 | p89 格雷编码 normal | 推荐 |
| 6 | p17 电话号码的字母组合 normal | |
| 7 | p93 复原IP地址 normal | |
| 8 | p357 计算各个位数不同的数字个数 normal | |
| 9 | p90 子集 II normal | 推荐 |

## 4.2 unofficial solution

### 4.2.1 p401 二进制手表

**枚举**

```
class Solution {
private:
    string gaoh(int h) {
        string result;
        if (h >= 10) {
            result += h / 10 + '0';
        }
        result += h % 10 + '0';
        return result;
    }

    string gaom(int m) {
```

```
        string result;
        result += m / 10 + '0';
        result += m % 10 + '0';
        return result;
    }
    string gao(int h, int m) {
        return gaoh(h) + ":" + gaom(m);
    }
public:
    vector<string> readBinaryWatch(int num) {
        vector<string> result;
        for (int h = 0; h < 12; h++) {
            for (int m = 0; m < 60; m++) {
                if (__builtin_popcount(h) + __builtin_popcount(m) == num) {
                    result.push_back(gao(h, m));
                }
            }
        }
        return result;
    }
};
```

### 4.2.2 p78 子集

**回溯**

$O(2^{N-1}N)$。

每次通过忽略最后一个元素，递归处理问题，处理完子问题再处理最后一个元素的影响，如此迭代。

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int> nums) {
        vector<vector<int> > result;
        if ( ! nums.empty()) {
            int elem = nums.back();
            nums.pop_back();
            result = subsets(nums);
            for (int i = result.size() - 1; i >= 0; i--) {
                result.push_back(result[i]);
                result.back().push_back(elem);
            }
        }
        else {
            result.push_back({});
        }
        return result;
    }
};
```

### 4.2.3 p22 括号生成

#### 做法一

- 首先第一个字符一定是 '('。
- 所以我们枚举和它匹配的 ')' 在哪个位置。
- 然后递归处理。

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
        if (n == 0) {
            result.push_back("");
            return result;
        }
        for (int right = 1; right < 2 * n; right += 2) {
            auto p1 = generateParenthesis(right - 1 >> 1);
            auto p2 = generateParenthesis(2 * n - right - 1 >> 1);
            for (auto e1 : p1) {
                for (auto e2 : p2) {
                    result.push_back("(" + e1 + ")" + e2);
                }
            }
        }
        return result;
    }
};
```

#### 做法二

- 做法一存在的问题是子问题重复处理。
- 所以我们可以尝试记忆化。

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
        if (n == 0) {
            result.push_back("");
            return result;
        }
        for (int right = 1; right < 2 * n; right += 2) {
            auto p1 = generateParenthesis(right - 1 >> 1);
            auto p2 = generateParenthesis(2 * n - right - 1 >> 1);
            for (auto e1 : p1) {
                for (auto e2 : p2) {
```

```
                    result.push_back("(" + e1 + ")" + e2);
                }
            }
        }
        return result;
    }
};
```

**做法三**

采用搜索的姿势进行枚举。

保证任意时刻使用的右括号数目不超过左括号数目，时间复杂度同 **做法二** 一样。

```
class Solution {
private:
    void dfs(int l, int r, vector<string> &result, string tmp = "") {
        if (!l && !r) {
            result.push_back(tmp);
            return;
        }
        if (l) {
            dfs(l - 1, r, result, tmp + '(');
        }
        if (r > l) {
            dfs(l, r - 1, result, tmp + ')');
        }
    }
public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
        dfs(n, n, result);
        return result;
    }
};
```

## 4.2.4 p46 全排列

**做法一**

- 使用内置的 $next\_permutaion$ 函数。

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int> > result;
        do {
            result.push_back(nums);
        }   while(next_permutation(nums.begin(), nums.end()));
        return result;
    }
};
```

**做法二**

- 考虑规约成子问题。
- 考虑第一个元素所在的位置，而后去除第一个元素。
- 备注：该做法效率极低。

```
class Solution {
public:
    vector<vector<int>> permute(vector<int> nums) {
        vector<vector<int> > result;
        if (nums.empty()) {
            result.push_back({});
            return result;
        }
        int elem = nums.back();
        nums.pop_back();
        auto temp = permute(nums);
        for (auto per : permute(nums)) {
            for (int i = 0; i <= nums.size(); i++) {
                result.push_back(per);
                result.back().insert(result.back().begin() + i, elem);
            }
        }
        return result;
    }
};
```

### 4.2.5 p89 格雷编码

- 递归生成，其实是一个巧妙的构造方法。
- 先生成长度为 $n-1$ 的格雷码。
- 我们把它镜像复制一份，并且在所有复制上最高位补一个1，我们就会得到长度 $n$ 的格雷码。

```
class Solution {
public:
    vector<int> grayCode(int n) {
        if (n == 0) {
            return {0};
        }
        auto result = grayCode(n - 1);
        for (int i = result.size() - 1; i >= 0; i--) {
            result.push_back(1 << (n - 1) | result[i]);
        }
        return result;
    }
};
```

### 4.2.6 p17 电话号码的字母组合

- 回溯枚举每个数字可以替换的字母。

```
class Solution {
private:
    void dfs(int current, string &digits, string tmp, vector<string> &result) {
        if (current == digits.size()) {
            result.push_back(tmp);
            return;
        }
        for (int i = 0; i < 3 + (digits[current] == '7' || digits[current] == '9'); i++) {
            char ch = digits[current];
            tmp.push_back('a' + 3 * (ch - '2') + i + (ch >= '8'));
            dfs(current + 1, digits, tmp, result);
            tmp.pop_back();
        }
    }
public:
    vector<string> letterCombinations(string digits) {
        vector<string> result;
        if (digits.empty()) {
            return result;
        }
        dfs(0, digits, "", result);
        return result;
    }
};
```

### 4.2.7 p93 复原IP地址

- 根据4部分进行回溯判断。

```
class Solution {
private:
```

```
    bool ok(string s) {
        if (s.empty() || s.size() > 3) {
            return false;
        }
        if (s.size() > 1 && s[0] == '0') {
            return false;
        }
        int value = 0;
        for (auto c : s) {
            value = value * 10 + c - '0';
        }
        return value <= 255;
    }
    void dfs(int step, string s, string temp, vector<string> &result) {
        if (step == 4) {
            if (ok(s)) {
                result.push_back(temp + s);
            }
            return;
        }
        for (int len = 1; len <= min(3, (int)s.size()); len++) {
            string part = s.substr(0, len);
            if (!ok(part)) continue;
            dfs(step + 1, s.substr(len), temp + part + ".", result);
        }
    }
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> result;
        dfs(1, s, "", result);
        return result;
    }
};
```

### 4.2.8 p357 计算各个位数不同的数字个数

- 最高位不能为0，所以枚举长度。

```
class Solution {
private:
    int dfs(int len, int remain = 9) {
        return !len ? 1 : dfs(len - 1, remain - 1) * remain;
    }
public:
    int countNumbersWithUniqueDigits(int n) {
        int result = 0;
        for (int len = 1; len <= n; len++) {
            result += 9 * dfs(len - 1);
        }
        return result + 1;
    }
```

```
};
```

### 4.2.9 p90 子集 II

- 每次抠掉一个元素，回溯的时候再把元素的可能数量考虑进去。

```
class Solution {
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums, bool sorted = false) {
        if (!sorted) {
            sort(nums.begin(), nums.end());
        }
        if (nums.empty()) {
            return {{}};
        }
        int elem = nums.back(), cnt = 0;
        while (!nums.empty() && nums.back() == elem) {
            cnt++;
            nums.pop_back();
        }
        auto result = subsetsWithDup(nums, true);
        for (int i = result.size() - 1; i >= 0; i--) {
            auto temp = result[i];
            for (int j = 1; j <= cnt; j++) {
                temp.push_back(elem);
                result.push_back(temp);
            }
        }
        return result;
    }
};
```