2022.11.5

每日一题 P1106 解析布尔表达式

这道题就是一个经典的栈的题目，跟实现一个简单的四则运算的程序类似，只需要不断对栈进行操作即可。

```
class Solution {
public:
    bool isop(char ch) { return ch == '!' || ch == '&' || ch == '|';}

    char cal(char op, vector<char> nums) {
        char ans = 't';
        switch(op) {
            case '!' :  ans = nums[0] == 't' ? 'f' : 't'; break;
            case '&' :
                for(auto ch : nums) if(ch == 'f') { ans = 'f'; break;}
                break;
            case '|':
                ans = 'f';
                for(auto ch : nums) if(ch == 't') { ans = 't'; break;}
                break;
        }
        return ans;
    }


    bool parseBoolExpr(string expression) {
        stack<char> op;
        stack<char> num;
        for(auto ch : expression) {
            if(isop(ch)) op.push(ch);
            else {
                if(ch == ',') continue;  //逗号不加入栈
                if(ch != ')') num.push(ch); //除了右括号以外还剩的 f t ( 入栈
                else {
                    vector<char> nums;  // 取出括号内的全部操作数
                    while(num.top() != '(') {
                        nums.push_back(num.top());
                        num.pop();
                    }
                    num.pop();
                    num.push(cal(op.top(),nums)); //进行运算后再把结果塞回去
                    op.pop();
                }
            }
        }
        return num.top() == 't';
        // 这道题比较特殊，全部括号匹配判断完以后就正好结束了
        // 如果是四则运算的话还需要加一步把剩下的 op 栈内没有完成的操作再执行完
    }
};
```

P139 单词拆分

首先尝试暴力写字典树，未果

```
class Solution {
private:
    struct node{
        int next[27];
        bool isend;
    }tire[20000];
    int head[27];
    int tot = 0;

public:
    bool wordBreak(string s, vector<string>& wordDict) {
        for(auto str : wordDict) build(str);
        int ptr = head[s[0] - 'a'];
        if(ptr == 0) return false;
        for(int i = 1; i < s.size(); i++) {
            char ch = s[i];
            if(tire[ptr].isend) ptr = head[ch - 'a'];
            else {
                ptr = tire[ptr].next[ch - 'a'];
                if(ptr == 0) return false;
            }
        }
        return tire[ptr].isend;  //返回此时是否是一个单词末尾
    }

    void build(string str) {
        int ch = str[0], len = str.size();
        if(head[ch - 'a'] == 0) head[ch - 'a'] = ++tot;
        int temp = head[ch - 'a'];
        for(int i = 1; i < len; i++) {
            if(!tire[temp].next[str[i] - 'a'])
                tire[temp].next[str[i] - 'a'] = ++tot;
            temp = tire[temp].next[str[i] - 'a'];
        }
        tire[temp].isend = true;  //这句话位置值得注意
    }
};
```

给出 $hack$ 数据如下;

> s = "aaaaaaa"
>
> wordDict = ["aaaa","aaa"]

原因是： 对 s 来说，先匹配到前三个 a 后回归，又匹配了三个 a 回归， 此时只剩下一个 a 了当然无法匹配。

所以这个样例启示我们，在进行 tire 树匹配时要尽量多的匹配下去，而不能找到第一个就结束匹配。因此此题如何判断是哪一个字符串，怎么判断结束跳转才是关键。由此引出区间DP

正解：区间DP

设DP数组中 dp[i] 表示对 s 的前 [1...,i-1,i] 个字符组成的字符串是否可以被用字典里的词来表示，则对于 dp[i+1] 来说每次只需要检查[j,...,i,i+1] 组成的字符串是否可被表示，然后遍历 j 即可

$$dp[i] = \bigcup_{j=1}^{i-1}(dp[j] \ \ \&\& \ \ check[j+1,\dots i])$$

对于判断函数可以遍历写判断，也可以借助字典树实现判断。

```
class Solution {
private:
    struct node{
        int next[27];
        bool isend;
    }tire[20000];
    int head[27];
    bool dp[400];
    int tot = 0;

public:
    bool wordBreak(string s, vector<string>& wordDict) {
        for(auto str : wordDict) build(str);
        dp[0] = true;
        int len = s.size();
        for(int i = 1; i <= len; i++) {
            dp[i] = check(s.substr(0,i));
            for(int j = 1; j < i; j++)
                dp[i] |= check(s.substr(i-j,j)) && dp[i-j];
        }
        return dp[len];
    }

    void build(string str) {
        int ch = str[0], len = str.size();
        if(head[ch - 'a'] == 0) head[ch - 'a'] = ++tot;
        int temp = head[ch - 'a'];
        for(int i = 1; i < len; i++) {
            if(!tire[temp].next[str[i] - 'a'])
                tire[temp].next[str[i] - 'a'] = ++tot;
            temp = tire[temp].next[str[i] - 'a'];
        }
        tire[temp].isend = true;
    }

    bool check(string s) {
        int len = s.size();
```

```
        int ptr = head[s[0] - 'a'];
        if(ptr == 0) return false;
        for(int i = 1; i < len; i++) {
            ptr = tire[ptr].next[s[i] - 'a'];
            if(!ptr) return false;
        }
        return tire[ptr].isend;
    }
};
```

或者也可以建立一个哈希表实现近似的 $O(1)$ 查询

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        auto wordDictSet = unordered_set <string> ();
        for (auto word: wordDict) {
            wordDictSet.insert(word);
        }

        auto dp = vector <bool> (s.size() + 1);
        dp[0] = true;
        for (int i = 1; i <= s.size(); ++i) {
            for (int j = 0; j < i; ++j) {
                if (dp[j] && wordDictSet.find(s.substr(j, i - j)) !=
wordDictSet.end()) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.size()];
    }
};
```

其实一般来说对于区间DP由于其复杂度是由「状态数 + 找前驱」的复杂度所共同决定，因此导致了一旦使用区间DP的题目其给的数据一定不会太大。

P146 单词拆分二

在上面一道题的铺垫下，这个题很明显也是一个区间DP的题目，不同的是增加了回溯，要求把全部的可能成果均输出。

那怎么遍历呢，我的做法是：把每一个节点的后继可能的节点都保存下来，然后进行 DFS 深搜即可

```
class Solution {
private:
    vector<int> pre[25];  //pre[i] 表示长度为 i 时可能到达的后继节点
    vector<string> ans;
public:
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        auto wordDictSet = unordered_set <string> ();
        for (auto word: wordDict) wordDictSet.insert(word);
```

```
        auto dp = vector <bool> (s.size() + 1);
        dp[0] = true;
        for (int i = 1; i <= s.size(); ++i) {
            for (int j = 0; j < i; ++j) {
                if (dp[j] && wordDictSet.find(s.substr(j, i - j)) !=
wordDictSet.end()) {
                    dp[i] = true;
                    pre[j].push_back(i);
                }
            }
        }
        DFS(s,0,"");
        return ans;
    }

    void DFS(string s,int idx,string str) {
        vector<int>& now = pre[idx];
        if(idx == s.size()) {if(str != "") ans.push_back(str); return;}
        for(auto next : now) {
            string sub = str;
            if(idx != 0) sub += " ";
            sub += s.substr(idx,next-idx);
            DFS(s,next,sub);
        }
    }
};
```