Algorithm

39. Combination Sum

```cpp
class Solution {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int
target) {
        std::vector<int> solution;
        std::sort(candidates.begin(), candidates.end());
        combinationSumPri(candidates, 0, target, solution);
        return results;
    }
private:
    void combinationSumPri(std::vector<int>& candidates, int index,
                           int target, std::vector<int>& solution){
        if(target == 0){
            results.push_back(solution);
            return;
        }
        if(index >= candidates.size()){
            return;
        }
        if(candidates[index] > target){
            return;
        }
        combinationSumPri(candidates, index + 1, target, solution); //not
chose current element

        int cnt = 0;
        while(true){
            if(target >= candidates[index]){ //chose for multi time
                target -= candidates[index];
                solution.push_back(candidates[index]);
                combinationSumPri(candidates, index+1, target, solution);
                cnt++;
            }else{
                break;
            }
        }
        while(cnt){
            solution.pop_back();
            cnt--;
        }
    }

private:
    std::vector<std::vector<int>> results;
};
```

这里跟普通的在数组中寻找组合不同，这里可能会出现重复的元素，因此在考虑选择某个元素的时候，需要考虑每个元素选择多次的场景。

Review

Tech

词法分析器的原理入门

程序编译中，第一步就是要做词法分析，所谓词法分析，就是将一个表达式解析成一个一个的词语，用Token表示一个词语。一个Token包含两个属性：

1. type，即这个token所属的类型，是一个标志符，整数，类型，或者操作符。
2. text，即这个token所包含的值。

比如：`age >= 45`中包含三个token，age，>=, 45;

词法分析的过程是一个有限自动机的过程，即当分析到某个词时，进入一个状态，接着读入词，当满足一定条件时，状态就会随之转换。随着状态的转换，最终完成词法的分析过程，这里给出`age >= 45`的分析程序：

```
import com.sun.deploy.util.SyncAccess;

import java.util.ArrayList;

import static java.lang.Character.isAlphabetic;
import static java.lang.Character.isDigit;

public class Lexical {
    public void parse(String str){
        for (int i = 0; i < str.length(); i++) {
            if(str.charAt(i) == ' ') continue;
            //if(str.charAt(i) == ';') saveToken();
            step(str.charAt(i));
        }
    }

    private DfaState initToken(char ch){
        token = new Token();
        DfaState newState = DfaState.Initial;
```

```
        if(isAlphabetic(ch)){
            newState = DfaState.Id;
            token.tokenType = TokenType.Indentifier;
            tokenText.append(ch);
        }else if(isDigit(ch)){
            newState = DfaState.IntLiteral;
            token.tokenType = TokenType.Intliteral;
            tokenText.append(ch);
        }else if(ch == '>'){
            newState = DfaState.GT;
            token.tokenType = TokenType.GT;
            tokenText.append(ch);
        }
        return newState;
    }
    private void step(char ch){
        switch (nextState){
            case Initial:
                nextState = initToken(ch);
                break;
            case Id:
                if(isAlphabetic(ch) || isDigit(ch)){
                    tokenText.append(ch);
                }else{
                    saveToken();
                    nextState = initToken(ch);
                }
                break;
            case GT:
                if(ch == '='){
                    token.tokenType = TokenType.GE;
                    nextState = DfaState.GE;
                    tokenText.append(ch);
                }else {
                    saveToken();
                    nextState = initToken(ch);
                }
                break;
            case GE :
                saveToken();
                nextState = initToken(ch);
                break;
            case IntLiteral:
                if(isDigit(ch)){
                    tokenText.append(ch);
                }else{
                    saveToken();
                    nextState = initToken(ch);
```

```
                }
                break;
            default:
                break;
        }
    }

    public void printTokens(){
        for(int i = 0; i < tokens.size(); ++i){
            System.out.println(tokens.get(i).tokenType);
        }
    }

    private void saveToken(){
        token.text = tokenText.toString();
        tokens.add(token);
        tokenText.setLength(0);
    }

    private ArrayList<Token> tokens = new ArrayList<>();
    private DfaState nextState = DfaState.Initial;
    private StringBuilder tokenText = new StringBuilder();
    Token token;
}
```

Share

事务的隔离及实现

隔离

在RR级别下，事务T启动时候会创建一个视图read_view，之后事务T执行期间，即使有其他事务修改了数据，事务T看到的仍然跟在启动时看到的一样。

begin/start transaction开始的事务时候其实并不是事务的起点，而是第一个操作InnoDB表的语句，事务才真正启动。如果马上想启动一个事务，可以使用start transaction with consistent snapshot这个命令。

视图

在mysql中，有两个视图的概念。

1. view,用查询语句定义的虚拟表。
2. InnoDB在实现mvcc时用到的一致性读视图，即consistent read view，用于支持RC和RR隔离级别。

快照在mvcc是如何工作的

在RR隔离级别下，事务在启动时就拍了个快照，这个快照是基于整个库的。这时候会有一个transaction id，是一个严格递增的数据。每个数据有多个版本，当一个事务开始时，便分配给这个事务数据一个新的版本，为row trx_id，旧的版本要保留，并且在新的数据版本中，能够有信息能够直接拿到它。

即数据表的行记录，可能存在多个版本，每个版本具有不同的row trx_id。 view并不是真实存在的，而是根据当前版本和undo log计算出来的。

事务在启动时候，只会承认事务版本小于等于自身事务id的数据。数据版本的可见性规则，基于数据的row trx_id和这个一致性视图的对比结果得到。 row trx_id分成了几种情况，

1. 已提交事务
2. 未提交事务集合
3. 未开始事务

对于一个事务视图来说，除了自己的更新总是可见以外，有三种情况：

1. 版本未提交，不可见
2. 版本已提交，但是是在视图创建后提交的，不可见
3. 版本已提交，而且是在视图创建前提交的，可见

更新逻辑

更新数据都是先读后写的，只能读当前值，称为“当前读”。（current read） 可重复读的核心就是“一致性读”，而事务更新数据的时候，只能用当前读。如果当前的记录的行锁被其他事务占用的话，就需要进入锁等待。