求满减子集合方法getSubCoupons优化效果分析

```java
public static List<List<Coupon>> getSubCoupons(List<Coupon> enableUseCoupons, List<ProductLine>
        List<List<Coupon>> result = new ArrayList<>();
        int length = enableUseCoupons.size();
        int max = 1 << length;
        int num = length == 0 ? 0 : max;
        // 判断某种组合是否被判断过,默认为false
        boolean[] ifVisited = new boolean[max];

        //商品行信息，用于券组合过滤
        int lineSize = productLines.size();
        BigDecimal totalValue = productLines.stream().map(c->new BigDecimal(c.getPrice())).reduc

        for (int i = 1; i < num; i++) {
            // 判断当前方案是否判断过
            if (ifVisited[i]) {
                continue;
            }
            // 该方案之前没有判断过
            List<Coupon> subCoupon = new ArrayList<>();
            int index = i;
            for (Coupon enableUseCoupon : enableUseCoupons) {
                if ((index & 1) == 1) {
                    subCoupon.add(enableUseCoupon);
                }
                index >>= 1;
                if (index == 0) {
                    break;
                }
                count++;
            }
            // 当前方案检查完毕
            ifVisited[i] = true;
            if (!couponArrayCanUse(subCoupon, totalValue, lineSize)) {
                // i对应的方案不可行,枚举i对应方案的其他子集
                // --> i的二进制位子上 和 (1 << length) - 1: 相同为0,不同为1 --> 相当于取反码 01011
                int reversedI = i ^ ((1 << length) - 1);
                // k 从 reversedI 逐渐减少到1 eg: 01011 --> 01010 --> 01001 --> 01000 --> ...
                for (int k = reversedI; k > 0; ) {
                    int subI = i | k; // 这些也是不可行的方案
                    ifVisited[subI] = true;
                    k = k - 1;// k每次自减1
                    k = k & reversedI; // 保证在k递减的过程中,reversedI为0的位置,k始终为0
                    count++;
                }
                continue;
            }
            // 小于等于,继续执行
            result.add(subCoupon);
        }
        return result;
    }
```

Points

- `int max = 1 << length`；**需要对length进行限制,否则会造成max向上溢出**
  - Integer.MAX_VALUE == $2^{31}-1$,所以需要确保券的数量length $\leq$ 30
- 时间复杂度(纯暴力)
  - $length \leftarrow$ 券的数量,每一张券可以选择或者不选 $O(2^{length})$
  - 如果券最多为30,则时间复杂度为$O(2^{30})$近似$O(10^9)$,大概1s

优化效果估算方案设计

券数量

- 10张券,20张券

券的满减门槛

	低	中	高
门槛值	10元以内	10元-99元	100元-200元
比例	30%	30%	40%

发放方案(可自行配置,表格中数值为自行设置)

	低门槛	中门槛	高门槛
10张券	$5, 8, 10$	$20, 40, 80$	$120, 140, 160, 180$
20张券	$4, 5, 7, 8, 9, 10$	$20, 30, 40, 50, 70, 80$	$120, 130, 140, 150, 160, 170, 180, 190$

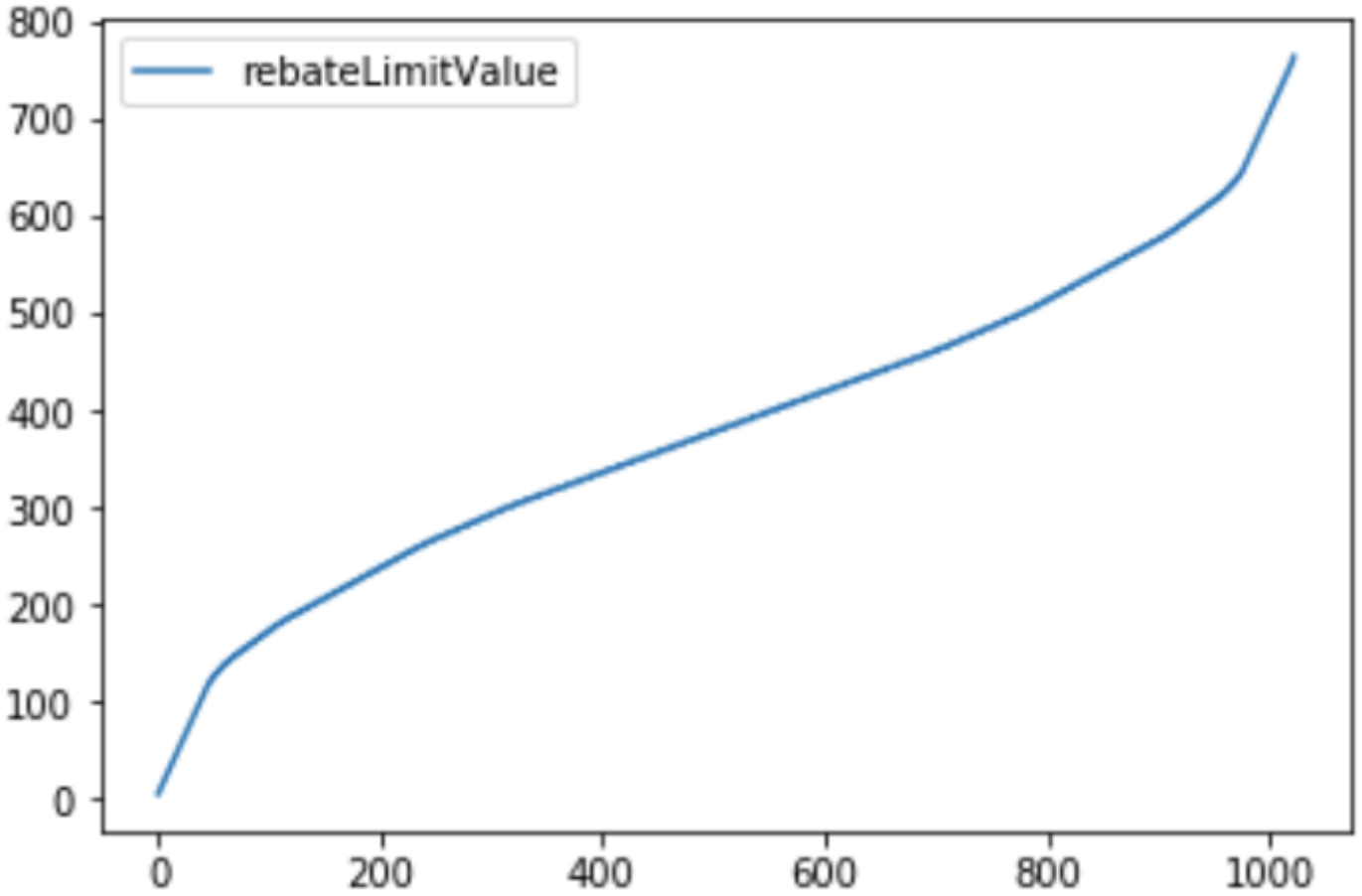
```
LT_10 = "<=10" # 满减门槛 <= 10
LT_100 = ">10And<=100" # 10 < 满减门槛 <= 100
LT_200 = ">100And<=200" # 100 < 满减门槛 <= 200
# 10张可用满减券的配置
REBATE10_CONFIG = dict()
REBATE10_CONFIG[LT_10]  =  [5,   8,   10]
REBATE10_CONFIG[LT_100] =  [20,  40,  80]
REBATE10_CONFIG[LT_200] =  [120, 140, 160, 180]

# 20张可用满减券的配置
REBATE20_CONFIG = dict()
REBATE20_CONFIG[LT_10]  =  [4,   5,   7,   8,   9,  10]
REBATE20_CONFIG[LT_100] =  [20,  30,  40,  50,  70,  80]
REBATE20_CONFIG[LT_200] =  [120, 130, 140, 150, 160, 170, 180, 190]
```

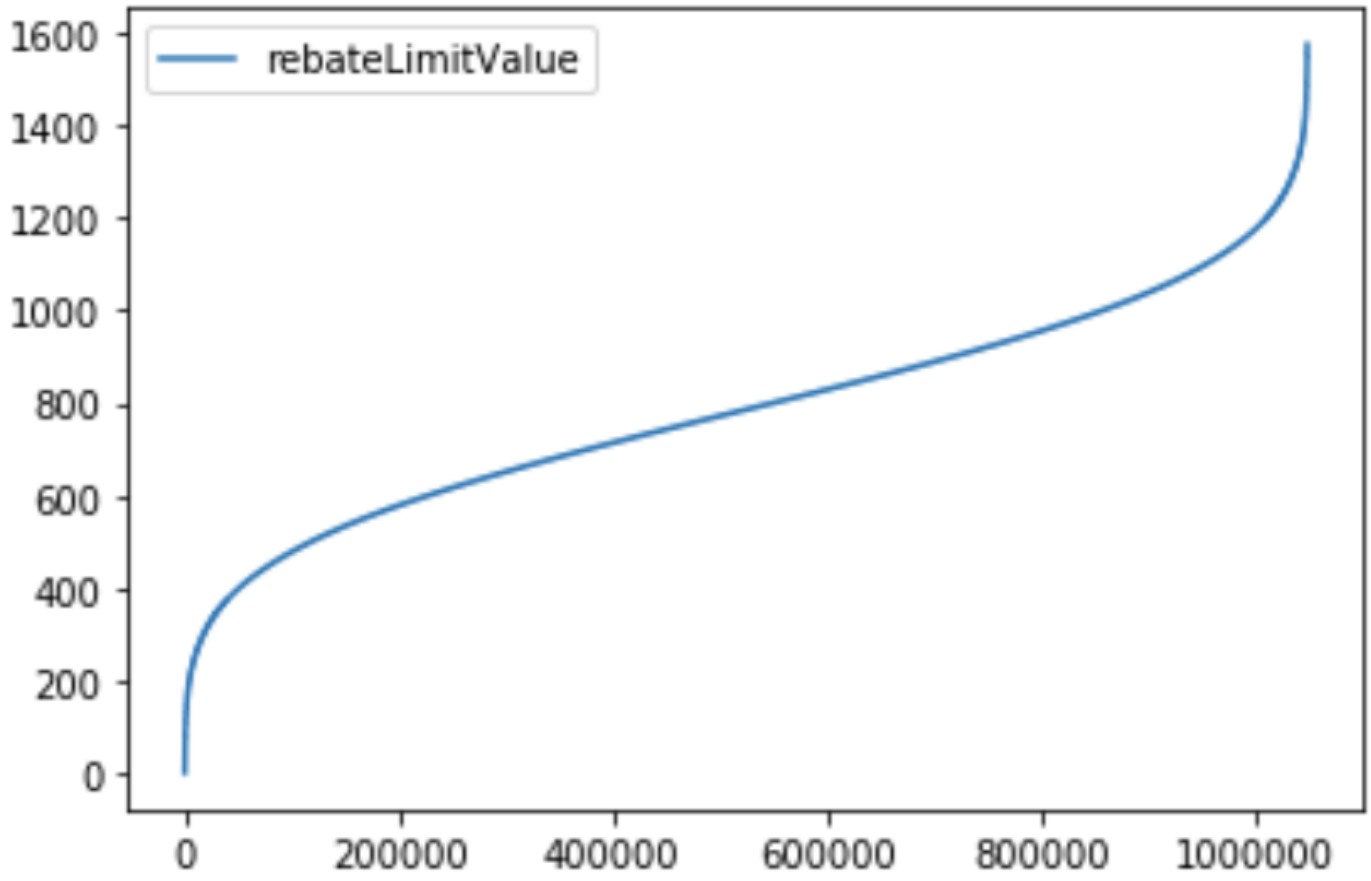
分别计算10张券2^{10}种组合方案,20张券2^{20}种组合方案对应的满减门槛累计值,并绘制曲线

```
def rebateLimitValues(rebatesN):
    results = []
    numberOfCoupons = len(rebatesN)
    numberOfRebateN = 1 << numberOfCoupons
    for i in tqdm_notebook(range(1, numberOfRebateN)):
        index = i
        curLimitValue = 0 # 当前方案i对应的优惠券选取方案的满减门槛
        for j in range(numberOfCoupons):
            if(index & 1)==1:
                # 满减门槛
                curLimitValue += rebatesN[j]
            index >>= 1
        results.append(curLimitValue)
    return results
```

10张券,一共2^{10},即1024种方案对应的满减门槛值曲线图



20张券,一共2^{20},即$1024 * 1024$种方案对应的满减门槛值曲线图



统计数据汇总

	10张券(低:中:高=3:3:4)	**20张券(低:中:高=3:3:4)**
25%(所有方案中25%的方案的满减门槛累计值≤该值)	270	627
50%(所有方案中50%的方案的满减门槛累计值≤该值)	383	787
75%(所有方案中75%的方案的满减门槛累计值≤该值)	493	946

优化效果计算

- 原方案

```
public static Set<Set<Integer>> getSubsetMethod(int[] set, int configLimitValue){
        Set<Set<Integer>> result = new HashSet<>();
        int length = set.length;
        int num = length == 0 ? 0 : (1 << length);
        for (int i = 1; i < num; i++) {
            Set<Integer> subSet = new HashSet<>();
            int index = i;
            int curLimitValue = 0; // 当前方案i对应的满减门槛累计值
            for (int j = 0; j < length; j++) {
                if((index & 1) == 1){
                    subSet.add(set[j]);
                    curLimitValue += set[j];
                }
                index >>= 1;
            }
            // 当前方案对应的满减门槛 > 商品总价
            if(!func(curLimitValue, configLimitValue)){
                // 该满减券方案无法使用
                continue;
            }
            // 小于等于,继续执行
            result.add(subSet);
        }
        return result;
    }
```

- 优化方案

```
public static Set<Set<Integer>> getSubsetMethodModified(int[] set, int configLimitValue){
        Set<Set<Integer>> result = new HashSet<>();
        int length = set.length;
        int num = length == 0 ? 0 : (1 << length);
        boolean[] ifVisited = new boolean[1 << length]; // 判断某种组合是否被判断过,默认为false
        for (int i = 1; i < num; i++) {
            if(ifVisited[i]){
                continue;
            }
            Set<Integer> subSet = new HashSet<>();
            int index = i;
            int curLimitValue = 0; // 当前方案i对应的满减门槛累计值
            for (int j = 0; j < length; j++) {
                if((index & 1) == 1){
                    subSet.add(set[j]);
                    curLimitValue += set[j];
                }
                index >>= 1;
            }
            // 当前方案检查完毕
            ifVisited[i] = true;
            // 当前方案对应的满减门槛 > 商品总价
            if(!func(curLimitValue, configLimitValue)){
                // 该满减券方案无法使用
                int reversedI = i ^ ((1 << length) - 1);
                // k 从 reversedI 逐渐减少到1 eg: 01011 --> 01010 --> 01001 --> 01000 --> ...
                for(int k = reversedI; k > 0;){
                    int subI = i | k; // 这些也是不可行的方案
                    ifVisited[subI] = true;
                    k = k - 1;// k每次自减1
                    k = k & reversedI; // 保证在k递减的过程中,reversedI为0的位置,k始终为0
                }
                continue;
            }
            // 小于等于,继续执行
            result.add(subSet);
        }
        return result;
    }
```

原方法

券数量 ×运行次数	10张 × 1000次	20张 × 10次
低商品总价(所有方案中25%的方案的满减门槛累计值 <= 商品总价)	275ms	71191ms

券数量 ×运行次数	10张 × 1000次	20张 × 10次
中商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价)	264ms	256075ms
高商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价)	325ms	2415811ms

```
原方法
10张券,低商品总价(所有方案中25%的方案的满减门槛累计值 ≤ 商品总价),运行1000次运行时间: 275
10张券,中商品总价(所有方案中50%的方案的满减门槛累计值 ≤ 商品总价),运行1000次运行时间: 264
10张券,高商品总价(所有方案中75%的方案的满减门槛累计值 ≤ 商品总价),运行1000次运行时间: 325
20张券,低商品总价(所有方案中25%的方案的满减门槛累计值 ≤ 商品总价),运行10次运行时间: 71191
20张券,中商品总价(所有方案中50%的方案的满减门槛累计值 ≤ 商品总价),运行10次运行时间: 256075
20张券,中商品总价(所有方案中75%的方案的满减门槛累计值 ≤ 商品总价),运行10次运行时间: 2415811
```

优化后方法

券数量 × 运行次数	10张 × 1000次	提升比例 $\frac{\text{原耗时}-\text{优化耗时}}{\text{原耗时}} * 100\%$	20张 × 10次	提升比例 $\frac{\text{原耗时}-\text{优化耗时}}{\text{原耗时}} * 100\%$
低商品总价(所有方案中25%的方案的满减门槛累计值 <= 商品总价)	54ms	80.36%	65758ms	7.63%
中商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价)	130ms	50.95%	215808ms	15.72%

券数量 × 运行次数	**10张 × 1000次**	**提升比例** $\frac{\text{原耗时}-\text{优化耗时}}{\text{原耗时}} * 100\%$	**20张 × 10次**	**提升比例** $\frac{\text{原耗时}-\text{优化耗时}}{\text{原耗时}} * 100\%$
高商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价)	187ms	42.46%	**393649**ms	83.71%

Points

- 为什么 券数量 × 运行次数组合的数值这样设置?
  - $2^{10} * 1000$ 约为 10^9
  - $2^{20} * 10$ 约为 10^9
- **建议该方法中券的数量最多为10张**

代码

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class RebateCouponBasicConfig {
    // 满减券门槛类型
    final static String LT_10 = "<=10"; // 低门槛
    final static String LT_100 = ">10And<=100"; // 中门槛
    final static String LT_200 = ">100And<=200"; // 高门槛

    static String REBATE10 = "10张满减券";
    static String REBATE20 = "20张满减券";
    static Map<String, List<Integer>> REBATE10_CONFIG = null;
    static Map<String, List<Integer>> REBATE20_CONFIG = null;
    static {
        // 低门槛券:中门槛券:高门槛券 == 3:3:4
        REBATE10_CONFIG = new HashMap<>();
        REBATE10_CONFIG.put(LT_10, Arrays.asList(5,8,10));
        REBATE10_CONFIG.put(LT_100, Arrays.asList(20,40,80));
        REBATE10_CONFIG.put(LT_200, Arrays.asList(120,140,160,180));

        REBATE20_CONFIG = new HashMap<>();
        REBATE20_CONFIG.put(LT_10, Arrays.asList(4,5,7,8,9,10));
        REBATE20_CONFIG.put(LT_100, Arrays.asList(20,30,40,50,70,80));
        REBATE20_CONFIG.put(LT_200, Arrays.asList(120,130,140,150,160,170,180,190));
    }

}
```

```java
import java.util.*;

import static couponChoose.RebateCouponBasicConfig.*;

public class RebateCouponConfig {
    // 商品总价:当商品总价 >= 满减组合对应的满减门槛时才能使用满减券
    // 券的总张数10 + 低门槛券:中门槛券:高门槛券 == 3:3:4
    final static int REBATE_10_LOWACC = 270; // 所有方案中25%的方案的满减门槛累计值 <= 该值
    final static int REBATE_10_MIDACC = 383; // 所有方案中50%的方案的满减门槛累计值 <= 该值
    final static int REBATE_10_HIGHACC = 493; // 所有方案中75%的方案的满减门槛累计值 <= 该值

    // 券的总张数20 + 低门槛券:中门槛券:高门槛券 == 3:3:4
    final static int REBATE_20_LOWACC = 627; // 所有方案中25%的方案的满减门槛累计值 <= 该值
    final static int REBATE_20_MIDACC = 787; // 所有方案中50%的方案的满减门槛累计值 <= 该值
    final static int REBATE_20_HIGHACC = 946; // 所有方案中75%的方案的满减门槛累计值 <= 该值

    // 券的总张数10对应的所有满减券集合
    static Map<String, List<Integer>> REBATE_CONFIG = null;
    static {
        REBATE_CONFIG = new HashMap<>();
        Iterator<Map.Entry<String, List<Integer>>> iter0 = REBATE10_CONFIG.entrySet().iterator()
        while(iter0.hasNext()){
            Map.Entry<String, List<Integer>> rebateTypeLimitValues = iter0.next();
            List<Integer> rebateCoupons = REBATE_CONFIG.getOrDefault(REBATE10, new ArrayList<>()
            rebateCoupons.addAll(rebateTypeLimitValues.getValue());
            REBATE_CONFIG.put(REBATE10, rebateCoupons);
        }
        Iterator<Map.Entry<String, List<Integer>>> iter1 = REBATE20_CONFIG.entrySet().iterator()
        while(iter1.hasNext()){
            Map.Entry<String, List<Integer>> rebateTypeLimitValues = iter1.next();
            List<Integer> rebateCoupons = REBATE_CONFIG.getOrDefault(REBATE20, new ArrayList<>()
            rebateCoupons.addAll(rebateTypeLimitValues.getValue());
            REBATE_CONFIG.put(REBATE20, rebateCoupons);
        }
       // System.out.println(REBATE_CONFIG.toString());
    }

    public static void main(String[] args) {

    }
}
```

```java
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import static couponChoose.RebateCouponBasicConfig.*;
import static couponChoose.RebateCouponConfig.*;
public class Method0 {
    static int[] limitValuesWith10RebateCoupons; // 10张满减券的情况下,每一张券对应的满减限制额度
    static int[] limitValuesWith20RebateCoupons; // 20张满减券的情况下,每一张券对应的满减限制额度


    static {
        limitValuesWith10RebateCoupons = REBATE_CONFIG.get(REBATE10).stream().mapToInt(k -> k).t
        limitValuesWith20RebateCoupons = REBATE_CONFIG.get(REBATE20).stream().mapToInt(k -> k).t

    }


    public static void main(String[] args) {
        System.out.println("原方法");
        // 10张满减券 运行10000次 券选取 (10^3) * 迭代次数 (10^4)
        System.out.print("10张券,低商品总价(所有方案中25%的方案的满减门槛累计值 <= 商品总价),运行1000
        getSubsetMethod0(limitValuesWith10RebateCoupons, REBATE_10_LOWACC, 1000);
        System.out.print("10张券,中商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价),运行1000
        getSubsetMethod0(limitValuesWith10RebateCoupons, REBATE_10_MIDACC, 1000);
        System.out.print("10张券,高商品总价(所有方案中75%的方案的满减门槛累计值 <= 商品总价),运行1000
        getSubsetMethod0(limitValuesWith10RebateCoupons, REBATE_10_HIGHACC, 1000);

        // 20张满减券 运行10次 券选取 (10^3 * 10^3) * 迭代次数 (10)
        System.out.print("20张券,低商品总价(所有方案中25%的方案的满减门槛累计值 <= 商品总价),运行10次
        getSubsetMethod0(limitValuesWith20RebateCoupons, REBATE_20_LOWACC, 10);
        System.out.print("20张券,中商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价),运行10次
        getSubsetMethod0(limitValuesWith20RebateCoupons, REBATE_20_MIDACC, 10);
        System.out.print("20张券,中商品总价(所有方案中75%的方案的满减门槛累计值 <= 商品总价),运行10次
        getSubsetMethod0(limitValuesWith20RebateCoupons, REBATE_20_HIGHACC, 10);


        System.out.println("优化后方法");
        // 10张满减券 运行10000次 券选取 (10^3) * 迭代次数 (10^4)
        System.out.print("10张券,低商品总价(所有方案中25%的方案的满减门槛累计值 <= 商品总价),运行1000
        getSubsetMethod1(limitValuesWith10RebateCoupons, REBATE_10_LOWACC, 1000);
        System.out.print("10张券,中商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价),运行1000
        getSubsetMethod1(limitValuesWith10RebateCoupons, REBATE_10_MIDACC, 1000);
        System.out.print("10张券,高商品总价(所有方案中75%的方案的满减门槛累计值 <= 商品总价),运行1000
        getSubsetMethod1(limitValuesWith10RebateCoupons, REBATE_10_HIGHACC, 1000);

        // 20张满减券 运行10次 券选取 (10^3 * 10^3) * 迭代次数 (10)
        System.out.print("20张券,低商品总价(所有方案中25%的方案的满减门槛累计值 <= 商品总价),运行10次
        getSubsetMethod1(limitValuesWith20RebateCoupons, REBATE_20_LOWACC, 10);
        System.out.print("20张券,中商品总价(所有方案中50%的方案的满减门槛累计值 <= 商品总价),运行10次
        getSubsetMethod1(limitValuesWith20RebateCoupons, REBATE_20_MIDACC, 10);
```

```
    System.out.print("20张券,中商品总价(所有方案中75%的方案的满减门槛累计值 <= 商品总价),运行10次
    getSubsetMethod1(limitValuesWith20RebateCoupons, REBATE_20_HIGHACC, 10);
}
// 原始方法
public static void getSubsetMethod0(int[] set, int configLimitValue, int iterTmes){
    long startTime = System.currentTimeMillis();    //获取开始时间

    for (int i = 0; i < iterTmes; i++) {
        getSubsetMethod(set, configLimitValue);
    }

    long endTme = System.currentTimeMillis(); // 获取结束时间
    System.out.println("运行时间: " + (endTme - startTime));



}
// 改进后的
public static void getSubsetMethod1(int[] set, int configLimitValue, int iterTmes){
    long startTime = System.currentTimeMillis();    //获取开始时间

    for (int i = 0; i < iterTmes; i++) {
        getSubsetMethodModified(set, configLimitValue);
    }

    long endTme = System.currentTimeMillis(); // 获取结束时间
    System.out.println("运行时间: " + (endTme - startTime));
}


/*
limitValuesWithNRebateCoupons:使用10/20张券对应的每一张券的限制额度
 */
public static Set<Set<Integer>> getSubsetMethod(int[] set, int configLimitValue){
    Set<Set<Integer>> result = new HashSet<>();
    int length = set.length;
    int num = length == 0 ? 0 : (1 << length);
    for (int i = 1; i < num; i++) {
        Set<Integer> subSet = new HashSet<>();
        int index = i;
        int curLimitValue = 0; // 当前方案i对应的满减门槛累计值
        for (int j = 0; j < length; j++) {
            if((index & 1) == 1){
                subSet.add(set[j]);
                curLimitValue += set[j];
            }
            index >>= 1;
        }
        // 当前方案对应的满减门槛 > 商品总价
        if(!func(curLimitValue, configLimitValue)){
            // 该满减券方案无法使用
```

```
            continue;
        }
        // 小于等于,继续执行
        result.add(subSet);
    }
    return result;
}

public static Set<Set<Integer>> getSubsetMethodModified(int[] set, int configLimitValue){
    Set<Set<Integer>> result = new HashSet<>();
    int length = set.length;
    int num = length == 0 ? 0 : (1 << length);
    boolean[] ifVisited = new boolean[1 << length]; // 判断某种组合是否被判断过,默认为false
    for (int i = 1; i < num; i++) {
        if(ifVisited[i]){
            continue;
        }
        Set<Integer> subSet = new HashSet<>();
        int index = i;
        int curLimitValue = 0; // 当前方案i对应的满减门槛累计值
        for (int j = 0; j < length; j++) {
            if((index & 1) == 1){
                subSet.add(set[j]);
                curLimitValue += set[j];
            }
            index >>= 1;
        }
        // 当前方案检查完毕
        ifVisited[i] = true;
        // 当前方案对应的满减门槛 > 商品总价
        if(!func(curLimitValue, configLimitValue)){
            // 该满减券方案无法使用
            int reversedI = i ^ ((1 << length) - 1);
            // k 从 reversedI 逐渐减少到1 eg: 01011 --> 01010 --> 01001 --> 01000 --> ...
            for(int k = reversedI; k > 0;){
                int subI = i | k; // 这些也是不可行的方案
                ifVisited[subI] = true;
                k = k - 1;// k每次自减1
                k = k & reversedI; // 保证在k递减的过程中,reversedI为0的位置,k始终为0
            }
            continue;
        }
        // 小于等于,继续执行
        result.add(subSet);
    }
    return result;
}

public static boolean func(int curLimitValue, int configLimitValue){
    return curLimitValue <= configLimitValue;
}
```

```
}
```