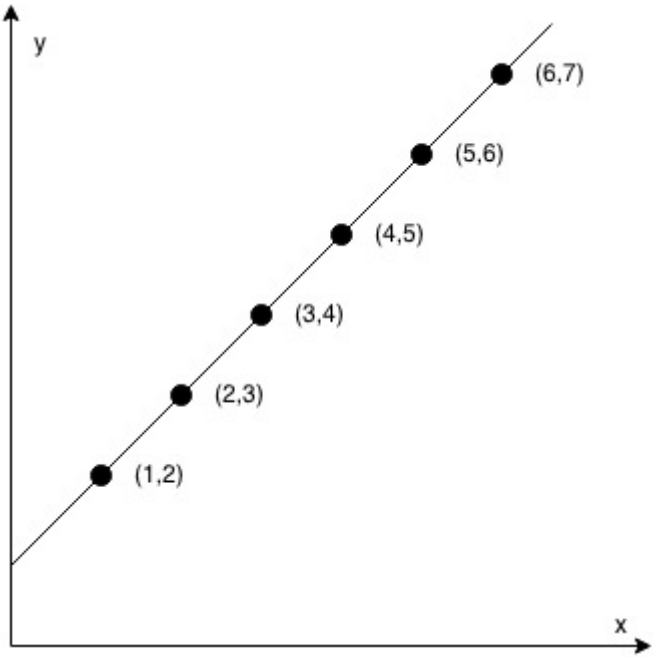
1232. 缀点成线（简单）

1. 题目描述

在一个 XY 坐标系中有一些点，我们用数组 `coordinates` 来分别记录它们的坐标，其中 `coordinates[i] = [x, y]` 表示横坐标为 `x`、纵坐标为 `y` 的点。

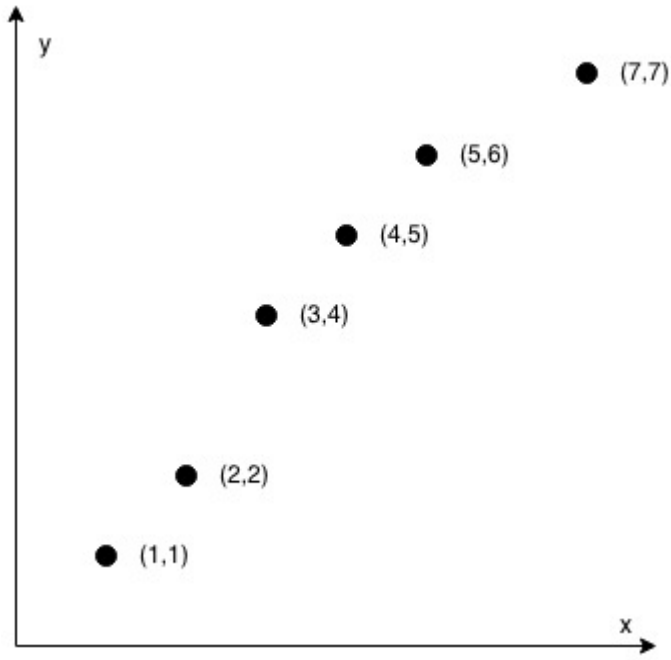
请你来判断，这些点是否在该坐标系中属于同一条直线上，是则返回 `true`，否则请返回 `false`。

示例 1：




```
输入：coordinates = [[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]]
输出：true
```

示例 2：




```
输入：coordinates = [[1,1],[2,2],[3,4],[4,5],[5,6],[7,7]]
输出：false
```

提示：

- `2 <= coordinates.length <= 1000`
- `coordinates[i].length == 2`
- `-10^4 <= coordinates[i][0], coordinates[i][1] <= 10^4`
- `coordinates` 中不含重复的点

2. 简单实现

```
class Solution {
public:
    bool checkStraightLine(vector<vector<int>>& coordinates) {
        if(coordinates.size() == 2)
            return true;
        if(coordinates[0][0] == coordinates[1][0]){//竖直
            for(int i = 2; i < coordinates.size(); i++)
                if(coordinates[i][0] != coordinates[0][0])
                    return false;
        }
        else if(coordinates[0][1] == coordinates[1][1]){//水平
            for(int i = 2; i < coordinates.size(); i++)
                if(coordinates[i][1] != coordinates[0][1])
                    return false;
        }
        else{//斜线
            float a = (coordinates[0][1] - coordinates[1][1]) / (coordinates[0][0]
- coordinates[1][0]);//斜率
            float b = coordinates[0][1] - a * coordinates[0][0];//截距
            for(int i = 2; i < coordinates.size(); i++)
                if(coordinates[i][1] != a * coordinates[i][0] + b)
                    return false;
```

```
            }
            return true;
        }
    };
```

1233. 删除子文件夹（中等）

1. 题目描述

   你是一位系统管理员，手里有一份文件夹列表 `folder`，你的任务是要删除该列表中的所有 **子文件夹**，并以 **任意顺序** 返回剩下的文件夹。

   我们这样定义「子文件夹」：

   - 如果文件夹 `folder[i]` 位于另一个文件夹 `folder[j]` 下，那么 `folder[i]` 就是 `folder[j]` 的子文件夹。

   文件夹的「路径」是由一个或多个按以下格式串联形成的字符串：

   - `/` 后跟一个或者多个小写英文字母。

   例如，`/leetcode` 和 `/leetcode/problems` 都是有效的路径，而空字符串和 `/` 不是。

   示例 1：

   ```
   输入：folder = ["/a","/a/b","/c/d","/c/d/e","/c/f"]
   输出：["/a","/c/d","/c/f"]
   解释："/a/b/" 是 "/a" 的子文件夹，而 "/c/d/e" 是 "/c/d" 的子文件夹。
   ```

   示例 2：

   ```
   输入：folder = ["/a","/a/b/c","/a/b/d"]
   输出：["/a"]
   解释：文件夹 "/a/b/c" 和 "/a/b/d/" 都会被删除，因为它们都是 "/a" 的子文件夹。
   ```

   示例 3：

   ```
   输入：folder = ["/a/b/c","/a/b/d","/a/b/ca"]
   输出：["/a/b/c","/a/b/ca","/a/b/d"]
   ```

   提示：

   - `1 <= folder.length <= 4 * 10^4`
   - `2 <= folder[i].length <= 100`
   - `folder[i]` 只包含小写字母和 `/`
   - `folder[i]` 总是以字符 `/` 起始
   - 每个文件夹名都是唯一的

2. 简单实现——前缀树

```
class Trie {
public:
    unordered_map<string, Trie*> children;
```

```
    string word;
    /** Initialize your data structure here. */
    Trie() {
        word = "";//word不为空时相当于isWord=true
    }
    //以'/'分割字符串s,结果存在ans中
    void split(string& s, vector<string>& ans){
        string temp = "";
        int i = 1;
        while(i < s.size()){
            if(s[i] == '/'){
                ans.push_back(temp);
                temp = "";
            }
            else temp += s[i];
            i++;
        }
        if(temp != "") ans.push_back(temp);
    }
    /** Inserts a word into the trie. */
    void insert(string& word) {
        Trie* cur = this;
        vector<string> words;
        split(word, words);
        for(int i = 0; i < words.size(); i++){
            if(cur->children.count(words[i]) <= 0)
                cur->children[words[i]] = new Trie();
            cur = cur->children[words[i]];
        }
        cur->word = word;
    }
};

class Solution {
public:
    vector<string> ans;
    void remove(Trie* t){//前序遍历删除子文件夹
        if(t){
            if(t->word != "") //访问到非子文件夹，加入ans，其孩子无需再遍历
                ans.push_back(t->word);
            else
                for(auto it = t->children.begin(); it != t->children.end(); it++)
                    remove(it->second);
        }
    }
    vector<string> removeSubfolders(vector<string>& folder) {
        Trie* t = new Trie();
        for(int i = 0; i < folder.size(); i++)
            t->insert(folder[i]);
        remove(t);
        return ans;
    }
};
```

3. 最简解法

先排序，则父文件夹和其子文件夹必定是连续排列的，所以若一个文件夹不是上一个父文件夹的子文件夹，那它自己就是一个父文件夹

```
class Solution {
public:
    vector<string> removeSubfolders(vector<string>& folder) {
        sort(folder.begin(),folder.end());
        vector<string> ans;
        ans.push_back(folder[0]);
        for(int i=1;i<folder.size();++i){
            string father=ans.back()+"/";//+"/"是为了避免 将 /a/bc 视为 /a/b的子文件夹
            string cur=folder[i];
            if(cur.find(father)==cur.npos)
                ans.push_back(cur);
        }
        return ans;
    }
};
```

1234. 替换子串得到平衡字符串（中等）

1. 题目描述

有一个只含有 `'Q', 'W', 'E', 'R'` 四种字符，且长度为 `n` 的字符串。

假如在该字符串中，这四个字符都恰好出现 `n/4` 次，那么它就是一个「平衡字符串」。

给你一个这样的字符串 `s`，请通过「替换一个子串」的方式，使原字符串 `s` 变成一个「平衡字符串」。

你可以用和「待替换子串」长度相同的 **任何** 其他字符串来完成替换。

请返回待替换子串的最小可能长度。

如果原字符串自身就是一个平衡字符串，则返回 `0`。

示例 1：

```
输入：s = "QWER"
输出：0
解释：s 已经是平衡的了。
```

示例 2：

```
输入：s = "QQWE"
输出：1
解释：我们需要把一个 'Q' 替换成 'R'，这样得到的 "RQWE" (或 "QRWE") 是平衡的。
```

示例 3：

```
输入: s = "QQQW"
输出: 2
解释: 我们可以把前面的 "QQ" 替换成 "ER"。
```

示例 4:

```
输入: s = "QQQQ"
输出: 3
解释: 我们可以替换后 3 个 'Q', 使 s = "QWER"。
```

提示:

- `1 <= s.length <= 10^5`
- `s.length` 是 `4` 的倍数
- `s` 中只含有 `'Q'`, `'W'`, `'E'`, `'R'` 四种字符

2. 简单实现

- 统计四个字符出现的次数，比较四个字符出现次数与期望次数
- 对于出现次数超过期望次数的字符，假设字符Q超出了n个，那么我们要找的子字符串就得至少含有n个Q 说至少，是因为含有n+1个Q也没关系，这多出来的1个Q可以继续让它为Q。
- 用滑动窗口找到上述字符串

```
class Solution {
public:
    int balancedString(string s) {
        vector<char> chars{'Q', 'W', 'E', 'R'};
        unordered_map<char,int> cnt;
        for(char ch:s) ++cnt[ch];
        int expection=s.size()/4;   //每个字母期望出现的次数
        bool balance=true;
        for(char ch:chars){
            if(cnt[ch]!=expection)
                balance=false;
            cnt[ch]-=expection;
        }
        if(balance==true) //不用替换就已经平衡
            return 0;

        int left=0,right=0,n=s.size(),ans=n; //滑动窗口，用左右两个指针
        while(left<=right&&right<n){
            --cnt[s[right]];
            bool find=true;//find表示是否找到可以替换的子字符串
            while(find){
                //当找到可以替换的子字符串时，有可能该子字符串的前缀包含了一些无关替换的字符
                //所以循环检测，每次left指针右移一位
                for(char ch:chars){ //判断目前是否已经找到子字符串
                   if(cnt[ch]>0){
                       find=false;
                       break;
                   }
                }
```

```
                if(find==true){          //找到了就计算子字符串长度，找不到就继续找：right++
                    ans=min(ans, right-left+1);
                    ++cnt[s[left++]];
                }
            }
            ++right;
        }
        return ans;
    }
};
```

1235. 规划兼职工作（困难）

1. 题目描述

你打算利用空闲时间来做兼职工作赚些零花钱。

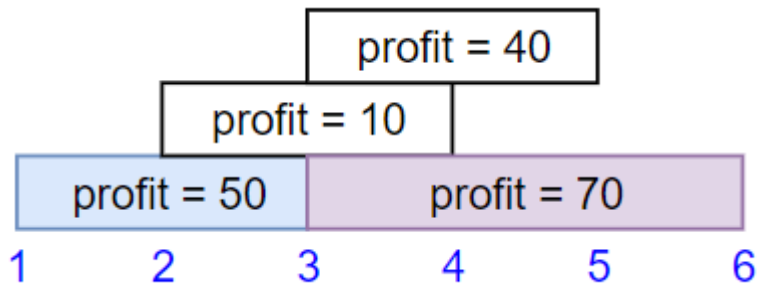
这里有 `n` 份兼职工作，每份工作预计从 `startTime[i]` 开始到 `endTime[i]` 结束，报酬为 `profit[i]`。

给你一份兼职工作表，包含开始时间 `startTime`，结束时间 `endTime` 和预计报酬 `profit` 三个数组，请你计算并返回可以获得的最大报酬。

注意，时间上出现重叠的 2 份工作不能同时进行。

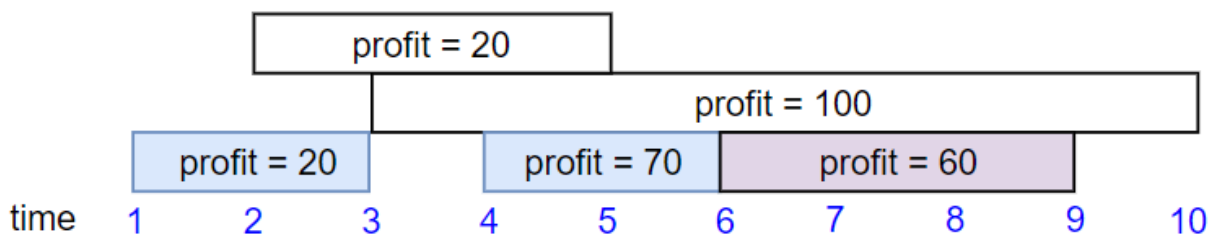
如果你选择的工作在时间 `X` 结束，那么你可以立刻进行在时间 `X` 开始的下一份工作。

示例 1：



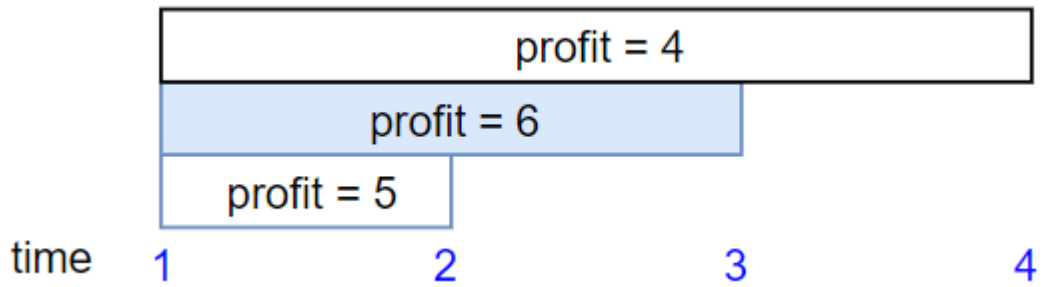

```
输入：startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]
输出：120
解释：
我们选出第 1 份和第 4 份工作，
时间范围是 [1-3]+[3-6]，共获得报酬 120 = 50 + 70。
```

示例 2：



```
输入：startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60]
输出：150
解释：
我们选择第 1，4，5 份工作。
共获得报酬 150 = 20 + 70 + 60。
```

示例 3：




```
输入：startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]
输出：6
```

提示：

- `1 <= startTime.length == endTime.length == profit.length <= 5 * 10^4`
- `1 <= startTime[i] < endTime[i] <= 10^9`
- `1 <= profit[i] <= 10^4`

2. 使用回溯剪枝会超时
3. 正确解法——动态规划+Map

题意等价于：求不相交区间的最大可能收益
$dp[b]$：在时间b内的最大收益

思考1：最后的结果(假设已经得到)中,对于第i个区间$[a,b]$有哪些状态?

> 0,1两个状态，取这区间(在满足条件下拿到这个收益)，不取这个区间（不要这个收益）。

思考2：为什么要按照endTime**排序**?

> 一个区间$[a,b]$的状态取决前面区间的最大结束时间（如果小于等于a,可以拿这个区间$[a,b]$，否则无法拿这个区间）。

从上可以看出这是一个**01背包**问题。

思考3：**状态**如何**转移**?(区间$[a,b]$,收益c)

> 按照01背包问题分析
> $dp[b]=max(dp[a]+c,dp[b])$

从以上分析可以得到以下的DP代码：

```
vector<int> dp(cnt+1,0);    // dp[b]
for(int i=0;i<n;i++){
    vector<int> tdp=dp;
    for(int j=0;j<cnt+1;j++){
        if(j<tim[i][1]) continue;
        tdp[j]=max(tdp[j],dp[tim[i][0]]+tim[i][2]); // 状态转移
    }
    //注意以下代码
    // 意义在于：若 e < f,保证必有dp[f]>=dp[e]
    for(int j=1;j<(cnt+1);j++){
        tdp[j]=max(tdp[j],tdp[j-1]);
    }
    dp=tdp;
}
```

这是基础代码，会发现超时。

思考4：上面代码慢在哪里?

> 对于一个区间$[a,b]$,需要修改的状态过多（整个dp），但从思考3中可以看到$dp[b]$只依赖与$dp[a]$与$dp[b]$,那么需要***快速找到***最大的$dp[j](j<=a)$与当前的$dp[k](k<=b)$，TreeMap可满足条件。

综上：

1. dp是$TreeMap$，不用线性数据结构表示。
2. $dp[b]$表示在b时间内的最大收益

```
class Solution {
public:
    int jobScheduling(vector<int>& s, vector<int>& e, vector<int>& p) {
        int n = s.size();
        vector<vector<int>> tim(n,vector<int>(3,0));
        for(int i=0;i<n;i++){
            tim[i][0]=s[i];
            tim[i][1]=e[i];
            tim[i][2]=p[i];
        }
        sort(tim.begin(),tim.end(),[](vector<int>& a,vector<int> &b){
```

```
            return a[1]<b[1];
        }); // 按照endTime排序

        // dp[b]表示在b时间内能达到的最大收益，题意即求dp[最大endTime]
        map<int,int> t;
        t[0]=0; // dp的初始条件
        int re=0;
        for(int i=0;i<n;i++){
            int a = tim[i][0];
            int b = tim[i][1];
            int c = tim[i][2];
            // 选取这个分组 dp[a] (01背包中1，快速找到dp[a]
            auto ite=t.upper_bound(a);
            --ite;
            // 不选取这个分组dp[b] (01背包中0)，快速找到dp[b]
            auto ite2 =t.upper_bound(b);
            --ite2;
            t[b]=max(ite2->second,ite->second+c);   // 01背包中取较大值
            re=max(re,t[b]);
        }
        return re;
    }
};
```

PS:

```
map::lower_bound(key):返回map中第一个大于或等于key的迭代器指针
map::upper_bound(key):返回map中第一个大于key的迭代器指针
```