1、

算法思路：

将孩子的胃口和饼干的大小进行排序，然后设置依次遍历取得孩子的胃口，然后遍历比对饼干的大小，符合要求就数量+1。直到将所有的孩子的胃口遍历完毕退出循环，或者当最大的饼干都无法满足当前的 temp 值时也退出循环。

复杂度分析：

k=max(n,m)，其中 n 为 g 的大小，m 为 s 的大小。

排序算法为:O(klogk)，while 循环为:O(n+m)

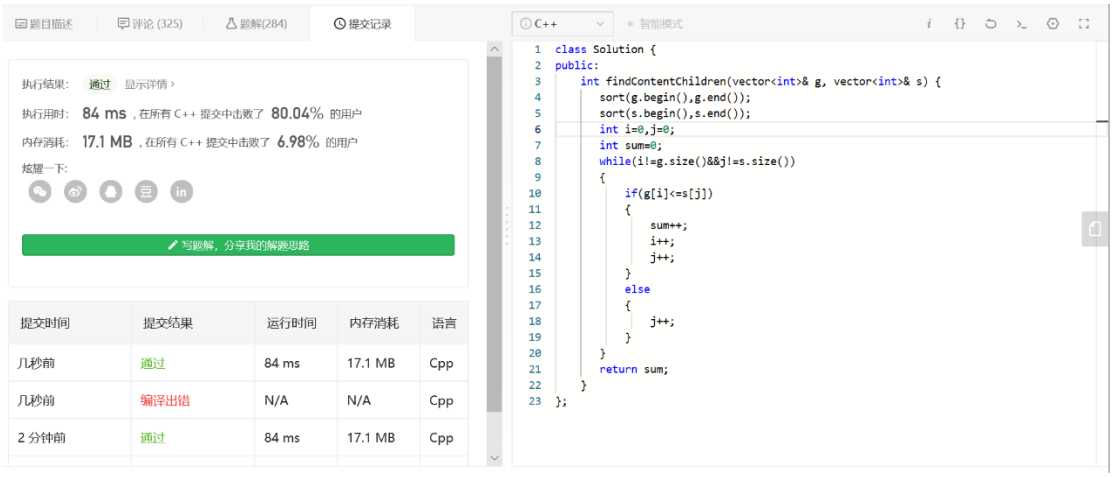
所以时间复杂度：O(klogk)。

空间复杂度为:O(1)。

代码：

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
       sort(g.begin(),g.end());
       sort(s.begin(),s.end());
       int i=0,j=0;
       int sum=0;
       while(i!=g.size()&&j!=s.size())
       {
           if(g[i]<=s[j])
           {
               sum++;
               i++;
               j++;
           }
           else
           {
               j++;
           }
       }
       return sum;
    }
};
```

截图：



2、

算法思路:

每次比较一下 A 和 B 的大小，将大的那个给添加到字符串的末尾，添加的条件是字符串的最后两个字母和对应的大写字母不是其小写字符，否则就将另一个字符添加到末尾，每次添加完毕后将对应的值减一，直到 A 和 B 都为 0 为止。

复杂度分析:

时间复杂度：O(A+B)；空间复杂度：O(1)。

代码:

```
class Solution {
public:
    string strWithout3a3b(int A, int B) {
        string s;
        while(A+B>0)
        {
            if(A>B)
            {
                if(s.size()<2)
                {
                    s.push_back('a');
                    A--;
                    continue;
                }
                if(s[s.size()-1]=='a'&&s[s.size()-2]=='a')
                {
                    s.push_back('b');
                    B--;
                }
                else
                {
                    s.push_back('a');
                    A--;
                }
```

```
            }
            else
            {
                if(s.size()<2)
                {
                    s.push_back('b');
                    B--;
                    continue;
                }
                if(s[s.size()-1]=='b'&&s[s.size()-2]=='b')
                {
                    s.push_back('a');
                    A--;
                }
                else
                {
                    s.push_back('b');
                    B--;
                }
            }
        }
        return s;
    }
};
```

截图：

3、

算法思路：

用一个相同大小的二维数组去存储到达该点的最短路径，即 dp[i][j]代表到达 i 行 j 列的最短路径。那么有下面的状态转移方程：

```
if(j==0)
                {
                    dp[i][j]=dp[i-1][j]+triangle[i][j];
                }
```

```
                else if(j==i)
                {
                    dp[i][j]=dp[i-1][i-1]+triangle[i][j];
                }
                else
                {
                    dp[i][j]=min(dp[i-1][j],dp[i-1][j-
1])+triangle[i][j];
                }
```

最后只需要将 dp 的最后一行的最小值返回即可。

复杂度分析:

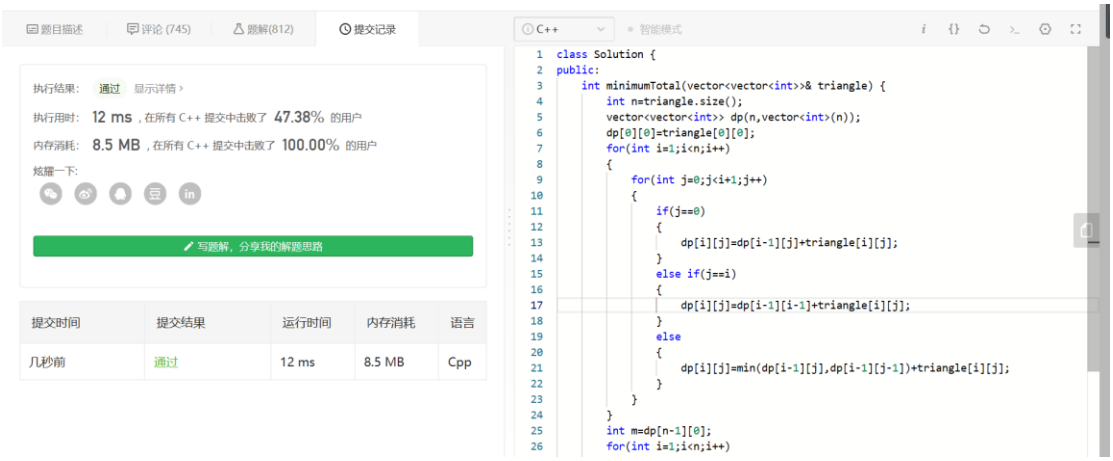
时间复杂度：$O(n^2)$，空间复杂度：$O(n^2)$，n 为数组大小

代码:

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n=triangle.size();
        vector<vector<int>> dp(n,vector<int>(n));
        dp[0][0]=triangle[0][0];
        for(int i=1;i<n;i++)
        {
            for(int j=0;j<i+1;j++)
            {
                if(j==0)
                {
                    dp[i][j]=dp[i-1][j]+triangle[i][j];
                }
                else if(j==i)
                {
                    dp[i][j]=dp[i-1][i-1]+triangle[i][j];
                }
                else
                {
                    dp[i][j]=min(dp[i-1][j],dp[i-1][j-
1])+triangle[i][j];
                }
            }
        }
        int m=dp[n-1][0];
        for(int i=1;i<n;i++)
        {
            //cout<<dp[n-1][i]<<endl;
            if(m>dp[n-1][i])
            m=dp[n-1][i];
```

```
        }
        return m;
    }
};
```

截图：



4、

算法思路：

用两个变量来记录自己的收益：buy 和 sell，其中，buy 代表的是手上不持有股票的收益，sell 就是持有股票的收益。对于 buy，我们可以选择继续持有或者卖出，对于 sell 我们可以选择买入，或者不买入。最后返回不持有股票的收益即可。

复杂度分析：

时间复杂度：O(n)，空间复杂度：O(1)。n 为数组大小。

代码：

```
class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
        int buy=0;int sell=-prices[0];
        for(int i=1;i<prices.size();i++)
        {
            buy=max(buy,sell+prices[i]-fee);
            sell=max(sell,buy-prices[i]);
        }
        return buy;
    }
};
```

截图：

5、

算法思路：

我们遍历字符串，当遍历到是 s[i]时，如果 s[i]为'0'，那么 s[i-1]应该为 1 或者 2 才有解码的可能。且是和前面的数字绑在一起进行解码，也就是说它的解码方法和遍历到 s[i-2]时是一样的（因为 s[i-1]和 s[i]是固定的解码手段）。

如果 s[i-1]为 1，或者 2 且 s[i]属于 1~6 这个区间，那么就会有绑在一起解码和分开解码两种方法，也就是绑在一起和分开解码的方法之和，也就是 now+front。

其他的情况的解码方法是没有变化的，所以不用更新。

复杂度分析：

时间复杂度：O(n)，空间复杂度:O(1)。n 为数组大小。

代码：

```
class Solution {
public:
    int numDecodings(string s) {
        int front=1,now=1;
        if(s[0]=='0')
        return 0;
        for(int i=1;i<s.size();i++)
        {
            int temp=now;
            if(s[i]=='0')
            {
                if(s[i-1]=='1'||s[i-1]=='2')
                {
                    now=front;
                }
                else
                return 0;
            }
            else if(s[i-1]=='1'||(s[i-1]=='2'&&s[i]>='1'&&s[i]<='6'))
            {
                now=now+front;
```

```
            }
            front=temp;
        }
        return now;
    }
};
```

截图：