算法作业4

第1题 最大m子段和问题

定义dp[i][j]为已经分了 j 段，且最后一个数字下标为 i 的最大结果

考虑两种情况的转移

- 从该数字开始，开启一个新子段
- 该数字加入前面的值最大的最后一个子段

时间复杂度分析：程序主体复杂度上限为状态转移的for循环，第一层执行次数为m，第二层执行次数为n，所以复杂度为O(m*n)

```
const int N=5e3+10,inf=-1e9;
int n,m;
int dp[N][N],a[N];
void solve(){
    for (int i=1; i<=m; i++) {
        for (int j=1; j<=n; j++) {
            dp[j][i]=inf;
        }
    }
    int ans=inf;
    for (int i=1; i<=m; i++) {
        int mx=inf;
        for (int j=1; j<=n; j++) {
            dp[j][i]=dp[j-1][i]+a[j];//情况1
            if (j>=i) {
                mx=max(dp[j-1][i-1],mx);
                dp[j][i]=max(dp[j][i],mx+a[j]);//情况2
            }
        }
    }
    for (int i=1; i<=n; i++) {
        ans=max(ans,dp[i][m]);//所有a[i]结尾的子序列且分了m段中的最大值
    }
}
```

第 2 题 交替硬币游戏

采用记忆化搜索的形式进行动态规划，定义dp[i][j]为对于区间i~j的最大价值，那么最后dp[1][n]即为先手所能取得的最大值，若该最大值比后手值大，则决定先手，否则后手

考虑状态转移，对于当前i~j的区间，我方有两种选择，即拿走最左边的或拿走最右边的，对方也可以选择拿走最左边的或拿走最右边的，总共有四种转移的情况，对四种值取最大值即位当前状态的最大值。

时间复杂度分析：因每个状态计算时，从它的四个子状态取最大值转移过来，对于相同的状态，采用了记忆化避免重复计算，所以时间复杂度就是状态总数，即O(n^2)

```
int n;
int dp[N][N],v[N];
int dfs(int l,int r){
    int &val=dp[l][r];
    if (val!=-1) {
        return val;
    }
    if (l==r) {
        return val=v[l];
    }
    if (l>r) {
        return 0;
    }
    int mx=0;
    mx=max({v[l]+dfs(l+1, r-1),v[l]+dfs(l+2, r),v[r]+dfs(l+1, r-
1),v[r]+dfs(l, r-2)});
    return val=mx;
}
void solve(){
    memset(dp, -1, sizeof dp);
    int sum=0;
    for (int i=1; i<=n; i++) {
        sum+=v[i];
    }
    int first=dfs(1,n);
    int second=sum-first;
    if (first>second) {
        cout<<"先手";
    }
```

```
    else cout<<"后手";
}
```

第3题 编辑距离

定义dp[i][j]为将a中1-i的子串变成b中1-j的子串的最小操作次数

初始化:

dp[0][i]如果a初始长度就是0，那么只能用插入操作让它变成b
dp[i][0]同样地，如果b的长度是0，那么a只能用删除操作让它变成b

状态转移:

- a[i]删掉之后a[1-i]和b[1-j]匹配,所以之前要先做到a[1-(i-1)]和b[1-j]匹配，

  所以dp[i][j]=dp[i-1][j]+1
- 插入之后a[i]与b[j]完全匹配，所以插入的就是b[j],那填之前a[1-i]和b[1-(j-1)]匹配

  dp[i][j]=dp[i][j-1]+1
- 把a[i]改成b[j]之后想要a[1-i]与b[1-j]匹配,那么修改这一位之前，a[1-(i-1)]应该与b[1-(j-1)]匹配dp[i][j]=dp[i-1][j-1] + 1

  但是如果本来a[i]与b[j]这一位上就相等，那么不用改，即dp[i][j]=dp[i-1][j-1]

时间复杂度分析：分析程序主体代码可知，复杂度瓶颈在于计算所有的状态，而对于每一个状态的计算，它可以用三种之前的状态转移过来，该转移是近似O(1)的，所以总时间复杂度是O(nm)

```
string a,b;
void solve(){
    n=a.size();
    m=b.size();
    for(int i=0,j=0;j<=m;j++) dp[i][j]=j;//a为空，只用插入操作
    for(int i=0,j=0;i<=n;i++) dp[i][j]=i;//b为空，只用删除操作
    for (int i=1; i<=n; i++) {
        for (int j=1; j<=m; j++) {
            dp[i][j]=min(dp[i-1][j]+1,dp[i][j-1]+1);
            if (a[i-1]==b[j-1]) {
                dp[i][j]=min(dp[i-1][j-1],dp[i][j]);
            }
            else dp[i][j]=min(dp[i-1][j-1]+1,dp[i][j]);
        }
```

```
    }
    cout<<dp[n][m];
}
```

第 4 题 附加题

不满足最优子结构，可能存在某种情况，一个状态的最优解并不是由它的子状态的最优情况转移过来，比如当一个位置上的数字是负数的时候，那么对于该位置结尾的序列，它的最优解不包含该数字，而若在后面加上几个正数，如果连续片段的个数不够用了（即已经分成了m段），则会和该位置连在一起，它们的和仍是正数，所以对于以后面数字结尾的序列是更优的。