2016 级算法第 4 次上机解题报告

15081070 张雨任

一、总结

本次上机主要主要考察知识点为动态规划、贪心等简单的算法知识。其中包括矩阵链乘、最长公共子序列等简单 DP 以及贪心问题。

二、解题报告

A. Bamboo 和人工 zz

思路分析

本题考察矩阵链乘法问题，可以通过动态规划来计算划分的位置和乘法运算次数，通过递归输出矩阵链的解决方案。

算法分析

首先通过 dp 解决矩阵链 A[1]A[2]…A[n]的划分位置并计算乘法运算次数。用 A[i…j]（i<=j）表示 A[i]A[i+1]…A[j]乘积的结果矩阵。可以看出，要对 A[i]A[i+1]…A[j]进行括号化，我们必须在某个 A[k]和 A[k+1]之间将矩阵链划分开（i<=k<=j 且 k 为整数）。此方案的计算代价等于矩阵的 A[i…k]的计算代价，加上矩阵 A[k+1…j]的计算代价，再加上两者相乘的计算代价。

令 ans[i][j]表示矩阵 A[i…j]所需标量乘法次数的最小值，那么，原问题的最优解——计算 A[1…n]所需的最低代价就是 ans[1][n]。因此我们可以递归定义 ans[i][j]。对于 i=j 的问题，矩阵链只包含一个矩阵，因此不需要做标量乘法，即 ans[i][i]=0。对于 i<j 的问题，那么 ans[i][j]就等于“计算 A[i…k]和 A[k+1…j]的代价和加上两者相乘的代价”的最小值。前者的代价即为 ans[i][k]以及 ans[k+1][j]，而 A[i]矩阵维度是 p[i-1]×p[i]，A[k]矩阵维度是 p[k-1]×p[k]，因此 A[i…k]矩阵维度是 p[i-1]×p[k]，同理可知 A[k+1…j]矩阵维度是 p[k]×p[j]，所以矩阵 A[i…k]和 A[k+1…j]矩阵相乘所需要的代价为 p[i-1]p[k]p[j]，因此得到 ans[i][j]=ans[i][k]+ans[k+1][j]+p[i-1]p[k]p[j]，要想求得最小值，则需要遍历所有的 k（i<=k<=j-1）。因此可以写出状态转移方程

$$m[i,j]=\begin{cases}0 & \text{如果 } i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{如果 } i<j\end{cases}$$

算出的 ans[1][n]即为最优代价。最外层循环遍历所有可能的矩阵链长度，中间一层遍历矩阵链的起点，最内层遍历所有合法的分割点。时间复杂度 $O(n^3)$.

想要求出矩阵链划分的具体的括号化方案，还需记录每个最优值 ans[i][j]的分割点 k，从而确定括号的位置，记录下 ans[i][j]最小时的 k 即可，即 s[i][j]=k。从而得到记录了每个子矩阵链的分割点的表 s[1…n-1][2…n]，从而递归地输出括号化方案。我们知道 $A_{1\ldots n}$ 的最优计算方案中最后一次矩阵乘法运算的括号化表示可以写成 ((A[1…s[1][n]])(A[s[1][n]+1…n]))，类似地，我们可以递归求解 A[1…s[1][n]]和 A[s[1][n]+1…n]的括号化方案。此步骤时间复杂度 O(nlgn)。

参考代码

```
#include <cstdio>
#include <cstring>
#define ll long long
using namespace std;

int n;
int p[307];
ll ans[307][307];
int s[307][307];

void printRes(int s[][307],int i,int j) {
    if(i==j) {
        printf("A%d",i);
    }
    else {
        printf("(");
        printRes(s,i,s[i][j]);
        printRes(s,s[i][j]+1,j);
        printf(")");
    }
}

int main() {
    while(~scanf("%d",&n)) {
```

```
        for(int i=0;i<=n;i++) {
            scanf("%d",&p[i]);
        }
        memset(ans,0,sizeof(ans));
        memset(s,0,sizeof(s));
        for(int l=2;l<=n;l++) {
            for(int i=1;i<=n-l+1;i++) {
                int j=i+l-1;
                ans[i][j]=0x3f3f3f3f;
                for(int k=i;k<=j-1;k++) {
                    int tmp=ans[i][k]+ans[k+1][j]+p[i-1]*p[k]*p[j];
                    if(tmp<=ans[i][j]) {
                        ans[i][j]=tmp;
                        s[i][j]=k;
                    }
                }
            }
        }
        printf("%lld\n",ans[1][n]);
        printRes(s,1,n);
        printf("\n");
    }
}
```

B. ModricWang 的序列问题

思路分析

经典 dp 问题，考察最长上升子序列长度的 O(nlgn)算法。可以考虑使用 dp，也可以使用类似于单调栈的思路。

算法分析

思路一：

上次上机的题解中解释了 $O(n^2)$的算法，转移方程为

$$a[i]=\min(a[j]+1),\ 1<=j<=i-1 \text{ 且 } data[j]>data[i]$$

本次要求 O(nlgn)完成。设输入数据为数组 data[1…n]，该数组的最长上升子序列长度为 res，考虑引入数组 last[1…res]，定义 last[i]为所有长度为 i 的最长上升子序列的最后一个元素的最小值，考虑引入数组 f[1…n]，定义 f[i]为以 data[1…n]中第 i 个元素为结尾的最长上升子序列长度，f[1…n]的最大值即为所求。因此 last[1…res]是一个不下降序列。当到第 i 个字符为止的 last[1…i]已知时，那么 last[1…i]中第一个大于或等于 data[i]的元素的位置，记为 pos，就等于以该元素为结尾的最长不上升子序列长度，也就是 f[i]。因为 data[i]需要放置在该位置以达到和前面序列组成上升子序列的目的。因此可以使用二分下界查找。然后更新 last[pos]，长度为 pos 的最长上升子序列中最后一个元素更新为 data[i]。

最后求得 f[1…n]的最大值即可。也可以不设立 f[1…n]数组，在迭代中记录并更新 f[1…n]的最大值。

时间复杂度 O(nlgn)。

思路二：

由于最长上升子序列是单调递增的，因此可以联想到单调栈的应用。对于最长上升子序列，为了让这个子序列尽量的长，应保持单调栈具有较大的“潜力”，就是在维护栈单调的同时，保持栈内元素尽量小。维护栈单调能够保证栈顶的指针为当前状态下最长上升子序列的长度，可以想象，这个栈只会伸长，不会缩短。维持栈内元素尽量小，使得栈 s[0…n]的元素 s[i]代表原串中长度为 i+1 的递增子序列的最后一个数的最小值，当一个上升序列末尾的值越小，其后面能够放入的更大的数就更多，具有较大的潜力。

因此，每当读入数据 d 大于栈顶元素时，d 进栈，当前序列的最长上升子序列变长。当

读入数据 d 小于栈顶元素，则要到栈里寻找第一个大于等于 d 的数，并替换它。保证栈内的元素在单调的情况下，尽量小。寻找第一个大于等于 d 的数，由于栈是单调的，所以可以考虑使用二分下界查找，保证 O(lgn)的时间复杂度。

总共输入 n 个数据，时间复杂度 O(n)，在栈内二分查找时间复杂度为 O(lgn)，因此算法的时间复杂度为 O(nlgn)。

关于此方法，搭配上文和 http://blog.csdn.net/shuangde800/article/details/7474903 中的例子食用更佳。

拓展

如何求得给定序列的所有最长上升子序列的具体方案？

可以通过 LCS（最长公共子序列）求得。设给定序列为 data[1…n]，然后对 data 按照升序排序，得到 sorted[1…n]，求这两个序列的最长公共子序列。由于 sorted[1…n]是单调递增的，因此 data[1…n]的长度为 k 的最长上升子序列一定是 sorted[1…n]的一个子序列，且 sorted[1…n]的任意一个长度大于 k 的子序列，虽然满足单调递增，但一定不是 data[1…n]的子序列。因此 data[1…n]和 sorted[1…n]的 LCS 一定是 data[1…n]的最长上升子序列，根据书上的代码可以求得 data[1…n]和 sorted[1…n]的 LCS（也就是 data[1…n]的最长上升子序列）的长度以及所有方案。

参考代码一

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;

const int inf=0x3f3f3f3f;
int n;
int data[500007];
int last[500007];
int res;

int main() {
    while(~scanf("%d",&n)) {
        for(int i=0;i<n;i++) {
```

```
            scanf("%d",&data[i]);
        }
        memset(last,0x3f,sizeof(last));
        res=0;
        for(int i=0;i<n;i++) {
            int j=lower_bound(last+1,last+1+res,data[i])-last;
            if(res<j) res=j;
            last[j]=data[i];
        }
        printf("%d\n",res);
    }
}
```

参考代码二

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;

const int inf=0x3f3f3f3f;
int n;
int data[500007];
int last[500007];
int f[500007];
int res;

int main() {
    while(~scanf("%d",&n)) {
        for(int i=0;i<n;i++) {
            scanf("%d",&data[i]);
        }
        memset(last,0x3f,sizeof(last));
        res=0;
        for(int i=0;i<n;i++) {
            int j=lower_bound(last+1,last+1+n,data[i])-last;
            f[i+1]=j;
            last[j]=data[i];
        }
```

```
        for(int i=1;i<=n;i++) {
            if(res<f[i]) res=f[i];
        }
        printf("%d\n",res);
    }
}
```

参考代码三

```
#include <cstdio>
#include <algorithm>
using namespace std;

int data;
int s[500007];
int n,k,top,pos;

int main() {
    while(~scanf("%d",&n)) {
        top=-1;
        for(int i=0;i<n;i++) {
            scanf("%d",&data);
            if(top==-1 || (top!=-1 && data>s[top])) {
                s[++top]=data;
            }
            else if(data<s[top]) {
                pos=lower_bound(s,s+top+1,data)-s;
                s[pos]=data;
            }
        }
        printf("%d\n",top+1);
    }
}
```

C. AlvinZH 的 1021 实验

思路分析

本题可以通过贪心策略解决，也可以从三进制的角度入手，还可以使用暴力做法。

算法分析

思路一：

考虑使用贪心策略。引入数组 data[0…5]记录砝码质量。

情况一：

对于当前输入的 n 满足 data[i]<n<data[i+1]，那么可以选择 data[i]+，也可以选择 data[i+1]-，那么两者如何选择：如果 n 可以分解为 data[0…i]的和，那么选择 data[i]+，否则，由于即使选择所有 data[0…i]相加也无法达到 n，只能选择 data[i+1]-。

情况二：

若 n 等于任意的一个 data[i]，则直接输出 n，不用输出符号，退出即可。

例如：n=14，在 data[0…5]={1,3,9,27,81,243}中，有 9<(n=14)<27，因此 14 可以选择 9+，也可以选择 27-，由于即使选择所有小于等于 9 的砝码 1+3+9=13 也无法达到 14，因此只能选择 27-，然后继续进行之后的选择。

所以，还要引入一个数组 sum[0…5]来存储 data[0…5]的前缀和，用来判定当前 n 选择 data[i]+还是 data[i+1]-。然后更新 n 的值：选择 data[i]+的情况下，n 失去 data[i]，变为 n-data[i]；选择 data[i+1]-的情况下，剩余的 n 变为 data[i+1]-n。

但是会发现结果的正负号有问题。原因在于负号的变号问题。举例说明，初始状态下 n=14，选择了 27-，剩余的 n 为 27-14=13。但接下来再分解 13 的时候，是忽略前面的负号独立考虑的，因此此处要处理一个变号问题。设立变量 chg 来标记变号情况。chg 为 false，则不变号，输出本来的情况；chg 为 true，则变号，输出相反的符号。明确只要进入 data[i+1]的情况，就需要改变变号标记 chg 的状态。

思路二：

可以考虑暴力做法。给定砝码中有 6 个，每一个选取都有三种选择——加、减、不选。那么根据乘法原理，总共有 3^6 种选法。考虑引入数组 w[0…5]，定义 w[i]为 data[i]砝码的选取情况，-1 代表“减”，0 代表“不选”，1 代表“加”。因此可以给出一个六重循环，遍历每种砝

码的可能的选择，在最内层循环累加得到当前数组 w[0…5]对应的累加和，判断是否与 n 相等即可。

最终输出时，w[i]=-1 输出减，w[i]=1 输出加，w[i]=0 代表当前砝码不选用。注意从高到低输出。

N 个砝码情况下，时间复杂度 $O(3^N)$。

思路三：

观察到给定砝码的质量分别为 3^0、3^1、3^2、3^3、3^4、3^5，对应的三进制均是最高位为 1，其余为 0。对于砝码选用，有“加”、“减”、“不选”三种处理办法。“加”对应该砝码权重为 1，“减”对应-1，“不选”对应 0。引入数组 w[0…5]，定义 w[i]代表第 i 个砝码的权重。

把 n 转化为三进制，存到数组 ter[0…5]中，从低到高遍历 n 的每一个三进制位，每一位有 0、1、2 三种选择，由于借位还会有 3 的情况。

2 必定是借位减法造成的。因为被加数的和加数的 1 都是错开的，所以不会有两个 1 对应相加。因此被减数对应的位是 0，减数对应的位是 1，即减数的最高位就是这一位，所以这一位为 1 的砝码的权重可以确定为-1。w[i]置-1。由于借位，前一位要增加 1。

3 属于原本这一位是 2，由于低一位是 2，借位，造成这一位加 1，相当于被减数和减数这一位都是 0，因此不存在此位为 1 的操作数，对应的权重为 0，w[i]置 0。

1 必定是加法造成的，由于不会出现类似于$(100)_3$-$(20)_3$=$(10)_3$ 的操作，因此只能由 0+1 得到，即其中一个加数的最高位就是这一位，所以这一位为 1 的砝码的权重可以确定为 1。w[i]置 1。

0 仅由 0+0 得到，代表最高位是这一位的砝码的权重为 0，直接把 w[i]置 0 即可。

最后按照思路二的办法输出结果即可。

参考代码一

```
#include <cstdio>
#include <string>
using namespace std;

int data[]={1,3,9,27,81,243};
int sum[]={1,4,13,40,121,364};
int n,i;
bool chg;
```

```
int main() {
    while(~scanf("%d",&n)) {
        chg=false;
        while(n) {
            if(n<243) {
                for(i=0;i<7;i++) {
                    if(n<data[i]) break;
                }
                i--;
            }
            else i=5;
            if(n==data[i]) {
                printf("%d",n);
                break;
            }
            else {
                if(n>sum[i]) {
                    printf("%d",data[i+1]);
                    n=data[i+1]-n;
                    if(chg) {
                        printf("+");
                        chg=false;
                    }
                    else {
                        printf("-");
                        chg=true;
                    }
                }
                else {
                    printf("%d",data[i]);
                    n-=data[i];
                    if(chg) printf("-");
                    else printf("+");
                }
            }
        }
        printf("\n");
    }
}
```

参考代码二

```
#include <cstdio>
using namespace std;

int data[]={1,3,9,27,81,243};
int w[10];
int n,sum;
bool jmp,first;

int main() {
    while(~scanf("%d",&n)) {
        jmp=false;
        for(w[0]=-1;w[0]<=1;w[0]++) {
            for(w[1]=-1;w[1]<=1;w[1]++) {
                for(w[2]=-1;w[2]<=1;w[2]++) {
                    for(w[3]=-1;w[3]<=1;w[3]++) {
                        for(w[4]=-1;w[4]<=1;w[4]++) {
                            for(w[5]=-1;w[5]<=1;w[5]++) {
                                sum=0;
                                for(int i=0;i<=5;i++) {
                                    sum+=w[i]*data[i];
                                }
                                if(sum==n) {
                                    jmp=true;
                                }
                                if(jmp) break;
                            }
                            if(jmp) break;
                        }
                        if(jmp) break;
                    }
                    if(jmp) break;
                }
                if(jmp) break;
            }
            if(jmp) break;
        }
        first=true;
        for(int i=5;i>=0;i--) {
```

```
            if(w[i]==1 && first) {
                printf("%d",data[i]);
                first=false;
            }
            else if(w[i]==1) printf("+%d",data[i]);
            else if(w[i]==-1) printf("-%d",data[i]);
        }
        printf("\n");
    }
}
```

参考代码三

```
#include <cstdio>
#include <cstring>
using namespace std;

int data[]={1,3,9,27,81,243};
int n,ans;
int ter[10];
int w[10];
bool first;

int main() {
    while(~scanf("%d",&n)) {
        memset(w,0,sizeof(w));
        memset(ter,0,sizeof(ter));
        int i;
        for(i=0;n;i++) {
            ter[i]=n%3;
            n/=3;
        }
        for(i=0;i<=6;i++) {
            if(ter[i]==2) {
                ter[i+1]++;
                w[i]=-1;
            }
            else if(ter[i]==3) {
                ter[i+1]++;
                w[i]=0;
```

```
            }
            else if(ter[i]==1) {
                w[i]=1;
            }
            else if(ter[i]==0) {
                w[i]=0;
            }
        }
        first=true;
        for(int i=5;i>=0;i--) {
            if(w[i]==1 && first) {
                printf("%d",data[i]);
                first=false;
            }
            else if(w[i]==1) printf("+%d",data[i]);
            else if(w[i]==-1) printf("-%d",data[i]);
        }
        printf("\n");
    }
}
```

D. AlvinZH 的 1021 实验 plus

思路分析

本题考查贪心策略，是一道难题。给定 n 个砝码，增加 cnt 个砝码，使得这 n＋cnt 个砝码能够通过相加计算 1～m 的任意一个重量，求 cnt 的最小值。

算法分析

可以考虑使用较为暴力的模拟整个过程的方法。

考虑引入数组 data[0…n-1] ，代表初始状态下的 n 个砝码，在贪心过程中，会加入新的砝码，因此 data 可伸长。最终 data 加入了 cnt 个新的砝码，得到数组 data[0…n+cnt-1] ，表示最终的砝码序列。因此 cnt 即为所求。注意，要随时保证 data 升序排列，保证选取的是最优方案。

首先，考虑初始状态。由于重量 m 的最小取 1 ，因此至少保证 data[0] = 1，因此对于给定没有砝码（ n＝0 ）的情况和 data[0] != 1 （最小砝码重量不为 1 ）的情况，要在 data 末尾加入 1 ，保证至少能计算 m＝1 的初始情况，每次更新 data 后都要排序，保证选取的是最优方案。

接下来使用指针 i 从 1 到 m 遍历每一个目标重量，来查看当前重量能否用已有的砝码加出来。在遍历重量的同时使用指针 j 遍历已有的砝码。这是本题的关键步骤。

考察重量 i+1 能否用已有砝码计算出：

若当前砝码（ data[j] ）小于等于当前重量（ i + 1 ）时（也要保证 j 合法不越界），说明不仅当前重量 i 可以被 data[0…j] 加出来，还能保证 i+1…i+data[j] 的重量也被 data[0…j] 加出来。因为使用砝码 data[j] 和能够计算出重量 i 的砝码，就可以算出重量 i＋data[j] ，就可以至少保证重量即可算出因此重量 i+1 到 i+data[j] 也解决了。综上所述，此时已经解决重量 1…i+data[j]，所以下一次 i 直接从 i+data[j]开始遍历即可。

若当前砝码（ data[j] ）大于当前重量（ i＋1 ）时，或者 j 已经越界时（相当于此时已经没有砝码可用，只能增添新的砝码），加入新的砝码。问题等价于重量 1…i 都可以是使用已有的砝码 data[0…j] 加出来，到 i+1 就不行了，因此要引入一个能接受的最小的砝码，保证局部最优，因此加入一个质量为 i+1 的砝码，至少保证重量 i+1 能被加出来，因此在数组 data 后面加入砝码 i+1 ，并且排序，使 i+1 被放在正确的位置。

维持这个循环，直到遍历到 m 为止， cnt 就是加入的砝码的数量，而 data[0…cnt+n-1]

就是最终状态加入 cnt 个新砝码的砝码质量序列。输出 cnt 即可。

要额外注意，这道题看似 int 可以解决，但是在运算的时候可能会溢出，因此使用 long long 。

时间复杂度 O(mlgn)。

参考代码

```
#include <cstdio>
#include <algorithm>
#define ll long long
using namespace std;

ll n,m,cnt,i,j;
ll data[100007];

int main() {
    while(~scanf("%d%d",&n,&m)) {
       for(i=0;i<n;i++) {
          scanf("%d",&data[i]);
       }
       cnt=0;
       sort(data,data+n);
       if(n==0 || data[0]!=1) {
          data[n+cnt]=1;
          cnt++;
       }
       sort(data,data+cnt+n);
       i=1;
       j=1;
       while(i<m) {
          if(data[j]<=i+1 && j<=n+cnt-1) {
             i+=(data[j]);
             j++;
          }
          else {
             data[n+cnt]=i+1;
             cnt++;
             sort(data,data+cnt+n);
          }
```

```
        }
        printf("%lld\n",cnt);
    }
}
```

E. Bamboo and the Ancient Spell

思路分析

本题考查 LCS，求给定两个串的最长公共子序列的长度。

算法分析

先来讨论一下传统的无附加条件的 LCS 问题。考虑引入数组 L[0…n1][0…n2]，定义 L[i][j]为串 s1 前 i 个字符组成的子串和串 s2 前 j 个字符组成的子串的 LCS 长度，易知 L[n1][n2]即为所求。要注意 s1 前 i 个字符组成的子串是 s1[0…n-1]，字符串从下标 0 开始。

$O(n^2)$遍历两个串 s1[0…n1-1]和 s2[0…n2-1]的所有字符，对比每两个字符的配对情况。令 s1[0…i-1]和 s2[0…j-1]，且 str[0…k-1]是 s1[0…i-1]和 s2[0…j-1]的任意 LCS。则有以下两种情况：

情况 1，配对成功：

若 s1[i-1]=s2[j-1]，则 str[k-1]=s1[i-1]=s2[j-1]且 str[0…k-2]是 s1[0…i-2]和 s2[0…j-2]的一个 LCS。则 L[i][j]可从 L[i-1][j-1]转移过来，由于配对成功，LCS 的长度增加一。

因此有 L[i][j]= L[i-1][j-1]+1，当且仅当 s1[i-1]=s2[j-1]。

情况 2，配对失败：

若 s1[i-1]!=s2[j-1]，则 str[k-1]!=s1[i-1]且 str[k-1]!=s2[j-1]，说明 str[0…k-1]是 s1[0…i-2]和 s2[0…j-1]的一个 LCS，也是 s1[0…i-1]和 s2[0…j-2]的一个 LCS，因此 L[i][j]可从 L[i-1][j]和 L[i][j-1]转移过来，由于配对失败，LCS 的长度保持不变，因此 L[i][j]=L[i-1][j]或 L[i][j]=L[i][j-1]，由于 L[i][j]是“最长”公共子序列，因此选择两者较大的。

因此有 L[i][j]=max(L[i-1][j], L[i][j-1])，当且仅当 s1[i-1]!=s2[j-1]。

额外注意边界情况，L[0][j]=L[i][0]=0，因为长度为 0 的串没有子序列。至此可以求解传统的 LCS 问题。

此题要对遍历到的字符进行特判，判定#和?的情况：

只要遍历到的两个字符，有一个是#，说明必定配对失败，归为情况 2。

若遍历到的字符，有一个是?，那么只要另一个不是#，即可算作配对成功，归为情况 1。

此题时间复杂度 $O(n^2)$。

参考代码

```
#include <cstdio>
#include <cstring>
#include <string>
#include <set>
#include <algorithm>
#include <iostream>
using namespace std;

char s1[105];
char s2[105];
int L[105][105];
int len1,len2,len;

int main() {
    while(~scanf("%s%s",s1,s2)) {
        memset(L,0,sizeof(L));
        len1=strlen(s1);
        len2=strlen(s2);
        for(int i=0;i<=len1;i++) {
            for(int j=0;j<=len2;j++) {
                if(i==0 || j==0) L[i][j]=0;
                else if(s1[i-1]==s2[j-1] && s1[i-1]!='#' &&
                        s2[j-1]!='#') {
                    L[i][j]=L[i-1][j-1]+1;
                }
                else if((s1[i-1]=='?' && s2[j-1]!='#') ||
                        (s1[i-1]!='#' && s2[j-1]=='?')) {
                    L[i][j]=L[i-1][j-1]+1;
                }
                else {
                    if(L[i-1][j]>L[i][j-1]) {
                        L[i][j]=L[i-1][j];
                    }
                    else {
                        L[i][j]=L[i][j-1];
                    }
                }
            }
```

```
        }
        len=L[len1][len2];
        printf("%d\n",len);
    }
}
```

F. AlvinZH 的最“长”公共子序列

思路分析

本题考查字符串编辑距离，属于经典 dp 问题。

编辑距离，指一个字符串通过增、删、改三种操作，转换成另一字符串所需要的最少操作次数。

算法分析

建立 dp 模型。考虑引入数组 ans[0…n1][0…n2]，n1、n2 分别为待测字符串 s1[0…n1-1]和 s2[0…n2-1]的长度，定义 ans[i][j]表示 s1 前 i 位（即 s1[0…i-1]）和 s2 前 j 位（即 s2[0…j-1]）的字符串编辑距离，因此 ans[n1][n2]即为所求。注意，字符串下标从 0 开始。

对模型进行初始化。考虑边界情况，对于 ans[i][j]，当 i=0 时，s1 的长度为 0，s2 的长度为 j，因此 s1 需要 j 次操作（增加字符）达到转换成 s2，因此有 ans[0][j]=j，同理 ans[i][0]=i。当然，ans[0][0]=0，两个串都是空串，不需要操作。

接下来考虑状态转移方程。值得注意的是，对 s1 进行删除操作，等价于对 s2 进行增添操作，由于 dp 的自底向上的性质，仅考虑删除操作即可。O(n^2)遍历两字符串，与 LCS 相同，在已知 ans[0…i-1][0…j-1]的情况下，推出 ans[i][j]，会遇到两种情况：

情况一，字符配对成功：

s1[i-1]=s2[j-1]，则针对两个串 s1[0…i-2]、s2[0…j-2]，分别增添 s1[i-1]和 s2[j-1]，不需要针对这个字符进行额外的编辑操作，那么 s1[0…i-2]、s2[0…j-2]和 s1[0…i-1]、s2[0…j-1]具有相同的编辑距离。

因此，ans[i][j]=ans[i-1][j-1]当且仅当 s1[i-1]=s2[j-1]。

情况二，字符配对失败：

s1[i-1]!=s2[j-1]，那么我们可以花费 1 个操作，更改 s1[j-1]为 s2[j-1]，使得 s1[i-1]=s2[j-1]，那么根据情况一，还需 ans[i-1][j-1]次操作，即 ans[i][j]=ans[i-1][j-1]+1。我们也可以花费 1 个操作，删掉 s1[i-1]，剩下的还需要 ans[i-1][j]次操作，即 ans[i][j]=ans[i-1][j]+1。同理，也可以花费 1 个操作，删掉 s2[j-1]，剩下的还需要 ans[i][j-1]次操作，即 ans[i][j]=ans[i][j-1]+1。由于要计算的最少的操作次数，因此要选取三种情况的最小值。

因此，ans[i][j]=min{ans[i-1][j-1]+1, ans[i-1][j]+1,ans[i][j-1]+1}。

最终，输出 ans[n1][n2]即可。时间复杂度 O(n1*n2)。

参考代码一

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define ll long long
using namespace std;

char s1[1007];
char s2[1007];
int ans[1007][1007];
int n1,n2;

int solve(char s1[],char s2[]){
    memset(ans,0,sizeof(ans));
    int i=0,j=0;
    for(j=0;j<=n2;j++){
        ans[0][j]=j;
    }
    for(i=0;i<=n1;i++){
        ans[i][0]=i;
    }
    for(i=1;i<=n1;i++){
        for(j=1;j<=n2;j++){
            if(s1[i-1]==s2[j-1]){
               ans[i][j]=ans[i-1][j-1];
            } else {
               ans[i][j]=min(ans[i][j-1],min(ans[i-1][j],
                         ans[i-1][j-1]))+1;
            }
        }
    }
    return ans[n1][n2];
}


int main() {
    while(~scanf("%s%s",s1,s2)) {
```

```
        n1=strlen(s1);
        n2=strlen(s2);
        printf("%d\n",solve(s1,s2));
    }
}
```