1. Shortest Travel Sequence

```
def LCS(f1,f2):
  l = [][]
  for j = 0; j <= f2.length; ++j:
    l[0][j] = 0
  for i = 0; i <= f1.length; ++i:
    l[i][0] = 0
  for i = 1; i <= f1.length; ++i:
    for j = 1; j <= f2.length; ++j:
      if f1[i] == f2[j]:
        l[i][j] = l[i-1][j-1]+1
      else:
        l[i][j] = max(l[i][j-1],l[i-1][j])
  return l[f1.length][f2.length]

def Get_Shortest_Length(f1,f2):
  return f1.length + f2.length - LCS(f1,f2)
```

> 时间复杂度为O(mn),dp求最长公共子串的时间为O(mn),得到结果只需要2次常数项操作，所以总的时间复杂度为O(mn)

正确性证明:

若f1和f2的最长公共子串为$s_1, s_2, ... s_k$,那么f1和f2字符串中位于s_i和s_{i+1}中间的字符串合并后采取任意的拓扑排序即可，只要是按照f1和f2中字符的拓扑顺序都能让$s_i ... s_{i+1}$间的字符串满足要求，所以根据此算法最后得到的字符串一定是满足顺序要求的，不过为什么一定是最短的呢?

若f1中的所有字符可以分为x,a两大类，x表示f1和f2的公共最长子序列，a表示剩下的字符。同理，f2中的所有字符也可以分为x,b两大类。所以根据上面的证明可知满足顺序要求的字符串长度为a+x+b，而a+x=f1.length,b+x=f2.length。所以结果字符串的长度为f1.lenth+f2.length-x,所以当x最大的时候，结果字符串长度最小，为f1.lenth+f2.length-x

2. Big Bang

```
def get_min_set(dic,str):
  dp = []
  dic = set(dic)
  str_set = []
  for i = 0; i < str.length; ++i:
    dp[i] = MAX_VALUE-1
    if(dic.contains(str[0:i+1])):
      dp[i] = 1
      str_set[i] = [str[0:i+1]]
    last = ""
    for j = i-1; j >= 0; --j:
     last = str[j+1:i+1]
```

```
    if(dic.contains(last) && dp[j] + 1 < dp[i]):
        dp[i] = dp[j] + 1
        str_set[i] = str_set[j].add(last)
  if(dp[str.length-1] == MAX_VALUE-1):
   print("Incomplete Dict!")
  else
   for words in str_set[str.length-1]:
     print(words)
```

> 通过动态规划，用dp[i]表示str[0:i]间能分解为字典单词的最小个数，当求解dp[i]时，从后往前遍历j=i-1...0。每次搜索到str[j:i]在字典中时，就判断dp[j]+1是否小于dp[i]，若是，则更新dp[i]，所以两次遍历时间复杂度为 :$O(n^2)$,n为目标字符串的长度

正确性证明:

这道题用动态规划求解

初始化dp为

$$dp[i] = \begin{cases} +\infty & str[0:i]\ not\ in\ dic \\ 1 & str[0:i]\ in\ dic \end{cases}$$

状态转移方程为:

$$dp[i] = \min_{0\le j\le i-1\ \&\ str[j:i]\ in\ dic} \{dp[i], dp[j]+1\}$$

所以每次求解的dp[i]都是前i个字符能分解为字典中单词的最小集合数量,

3. Shortest Path in Grid Space

```
    public static int [][]next = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    public static int dijkstra(int[][] grid){
        int n = grid.length;
        ArrayList<int[]> []w = new ArrayList[n*n];
        for (int i = 0; i < n*n; ++i){
            w[i] = new ArrayList<>();
        }
        for (int i = 0; i < n; ++i){
            for (int j = 0; j < n; ++j){
                for (int k = 0; k < 4; ++k){
                    int x = i + next[k][0],y = j + next[k][1];
                    if (x >= 0 && x < n && y >= 0 && y < n){
                        w[i*n+j].add(new int[]{x*n+y,grid[x][y]});//w[i]记录第i
个节点的邻居节点(节点数，路径消耗)
                    }
                }
            }
        }
```

```
        PriorityQueue<int[]> priorityQueue = new PriorityQueue<>((o1, o2) ->
o1[1]-     o2[1]);
        HashSet<Integer> hashSet = new HashSet<>();
        int []min_dis = new int[n*n];//记录未访问节点i到v0的最短路径
        Arrays.fill(min_dis,1,n*n,0x3333ffff);
        priorityQueue.offer(new int[]{0,0});
        while (hashSet.size() < n*n && !priorityQueue.isEmpty()){
            int []first = priorityQueue.poll();
            if (hashSet.contains(first[0])) continue;
            hashSet.add(first[0]);
            for (int[] edge : w[first[0]]){
                if (!hashSet.contains(edge[0]) && min_dis[first[0]] + edge[1] <
min_dis[edge[0]]){  //进行权值缩放
                    min_dis[edge[0]] = min_dis[first[0]] + edge[1];
                    priorityQueue.offer(new int[]{edge[0],min_dis[edge[0]]});
                }
            }
        }
        return min_dis[n*n-1]+grid[0][0];
    }
```

> 因为采用的DIJKSTRA单元最短路径的方法，所以如果最小优先队列采用的是二叉堆实现的话，时间复杂度约为O(ElgV),若采用的是斐波那契堆实现的话，时间复杂度约为O(VlogV+E),因为此题中G为n^2的图，所以$V=n^2$,$E=c*n^2$(c为常数)，所以时间复杂度约为$O(n^2lgn^2)$

正确性证明:

此题要求从最左上的点到最右下的点的路径中最小的消耗值，因为并没有规定只能向右或向下走所以不能使用动态规划，可以转换为图的最小路径问题，一共有n*n个点，每个点有4n条边，w[i,j] = G[i,j]。所以问题就转换成了单元最短路径问题，因为每个边上的权值都是正数，所以采用DIJKSTRA算法即可求出最左上点到最右下点的最小路径消耗。

4. Fish fish eat eat fish

(a)

我们的子问题可以转换为在 第i天第j个湖可以获得的最大鱼的数量。

递归关系为:

$$dp[i][j]=\begin{cases}\max\limits_{1\le k\le N\ \&\ k!=j}\{dp[i][j],dp[i-1][k]-w[k][j]+S[i][j]\} & dp[i-1][k]\ge w[k][j]\\ dp[i-1][j]+S[i][j] & else\end{cases}$$

初始值:$dp[1][x]=S[1][x],dp[i][y]=-\infty(y\neq x)$

证明:

$dp[i][j]$表示第i天在第j个湖的捕获鱼的最大值，他只有两种可能：1是第i-1天的时候从其他湖k捕获完后然后支付了$w[k][j]$后过来继续捕捞，最大值就是$dp[i-1][k]-w[k][j]+S[i][j]$或者是第i-1天的时候就是在第j个湖，因此不需要路费，因此最大值是$dp[i-1][j]+S[i][j]$

(b)

```
def get_max_fish(S,x,C):
  for i = 1; i <= m; ++i:
    dp[1][i] = MIN_VALUE
  dp[1][x] = S[1][x]
  for i = 2; i <= n; ++i:
    for j = 1; j <= m; ++j:
      for k = 1; k <= m; ++k:
        if(k != j && dp[i-1][k] >= C[k][j]):
          dp[i][j] = max(dp[i][j],dp[i-1][k]-C[k][j]+S[i][j])
        else:
          dp[i][j] = max(dp[i][j],dp[i-1][j]+S[i][j])
  max_fish = 0;
  for i = 1; i <= m; ++i:
    max_fish = max(max_fish,dp[n][i])
  return max_fish
```

> 正确性:根据第一问的状态转移方程可知，最后的$dp[n][i]$表示在第n天第i个湖能捕获的最多的鱼的数量,所以$\max\limits_{1=<i<=m} dp[n][i]$就是最后一天能在某个湖获得最多鱼的数量。时间复杂度为:$O(nm^2)$