

利用分段深度搜算的方式进行钢琴指法选择

一、研究背景

在钢琴弹奏中，对于难度极高的有大量高速连续弹奏音符的曲目，能否选取最有效的指法往往是弹奏能否成功的重要因素。本课题就是主要针对如何根据音符选取最有效的弹奏指法进行的研究。

二、变量说明

为了方便说明，现在用 $Note[i]$ 来表示全曲第 i 个音符的音调。下面是具体的标记说明：

1. 以中央 C 位的标准音 Do 作为 1，Re 就是 2，Mi 就是 3。Re 和 Do 之间的黑键作为 1.5。
2. 若是出现和弦，则将其拆分为几个单音，单音的顺序按高音在前，低音在后的顺序排序。
3. 由于左手和右手的对称性，以上的标记方式均是右手的标记方式，之后关于指法选择的讨论，也都将关注右手进行展开。

对于每一个音符 i ，它都对应一个弹奏指法，标记为 $Finger[i]$ 。

三、基础指法

钢琴有几种基础指法。如果已知选用哪一种基本指法，上一个音符的音调 ($Note[x]$)，当前音的音调 ($Note[x+1]$)，上一个音符的所选指法 ($Finger[x]$)，就可以给出当前音符的对应指法 ($Finger[x+1]$)。以下是几种基本指法：

1. 基本指法：弹奏的时候，5 个手指与五个相邻琴键一一对应。
比如，已知： $Note[x]=1, Note[x+1]=3, Finger[x]=1$ ，则 $Finger[x+1]=Finger[x]+Note[x+1]-Note[x]$ 。
使用该种指法的基本条件为：
上行时： $Finger[x] \leq Finger[x+1] \leq 5$ ；
下行时： $1 \leq Finger[x+1] \leq Finger[x]$ ；
2. 扩指：有别与基本指法，弹奏的时候，可以适当选择扩指，利用比基本指法更靠近上一个弹奏音符所用指法的指法进行目前音符的弹奏。
如果记扩指程度为 k ，则：
上行时扩指的表达式为：
 $Finger[x+1]=Finger[x]+Note[x+1]-Note[x]-k$ ；
下行时扩指的表达式为：
 $Finger[x+1]=Finger[x]+Note[x+1]-Note[x]+k$ ；
使用该种指法的基本条件为：
上行时： $Finger[x] \leq Finger[x+1] \leq 5, \Delta Note \leq K_Lim(Finger[x], Finger[x+1])$ ；
下行时： $1 \leq Finger[x+1] \leq Finger[x], \Delta Note \leq K_Lim(Finger[x+1], Finger[x])$ ；
其中， $K_Lim(m, n)$ ， $1 \leq m \leq n \leq 5$ 表示在前一音符指法使用 m 指，后一音符指法使用 n 指时所允许的两个音之间的最大音距
3. 缩指：与扩指相反，用比基本指法更远离上一个弹奏音符所用指法的指法进行目前音符的弹奏。
如果记缩指程度为 s ，则：

上行时缩指的表达式为: $Finger[x+1]=Finger[x]+Note[x+1]-Note[x]+s$;

下行时缩指的表达式为: $Finger[x+1]=Finger[x]+Note[x+1]-Note[x]-s$;

使用该种指法的基本条件为:

上行时: $Finger[x] \leq Finger[x+1] \leq 5$;

下行时: $1 \leq Finger[x+1] \leq Finger[x]$;

4. 轮指: 轮指的运用可以说是钢琴中最考究的部分。对已刚才上述的三种指法来说,使用它们所需要的上述条件都是一致的——除了要满足最基本的人类手指从 1 到 5 的限制以外,还需要保证在上行的过程中所用指法依次递增,在下行的过程中所用指法依次递减。对于轮指来说,它的基本要求与上述 3 种指法正好相反:它在上行时需要使用递减的指法,在下行的时候使用递增的指法。形象来说,它在钢琴弹奏时,就是利用某一手指“钻过”其他手指来弹奏。

对于轮指来说,因为受限于人类的手指结构,它在上下行时的基本指法并不显示出十分明显的对称性。

对于上行时, 设函数 $Turn_Up (Note[x], Note[x+1], Finger[x])$ 。

则:

$$\begin{aligned} & Turn (Note[x], Note[x+1], Finger[x]) = 1, \\ & \begin{cases} Finger[x] = 4, |Note[x] - Note[x+1]| \leq Turn_Lim[4] \\ Finger[x] = 3, |Note[x] - Note[x+1]| \leq Turn_Lim[3] \\ Finger[x] = 2, |Note[x] - Note[x+1]| \leq Turn_Lim[2] \end{cases} \end{aligned}$$

其中, $Turn_Lim[x]$ 表示前一指法为 x 的扩指的最大音距。

对于下行来说, $Turn_Down (Note[x], Note[x+1])$ 是 $Turn_Up$ 函数在已知等式成立, $Note[x]$ 和 $Note[x+1]$ 时求解 $Finger[x]$ 的一个反向映射。

由于同一组输入可以得到不同的可能解, 将其输出 $Turn (Note[x], Note[x+1])$ 定义为一个从大到小排列的 $Finger[x+1]$ 的可能指法的有序集合: $\{ Turn (Note[x], Note[x+1]) [1], Turn (Note[x], Note[x+1]) [2], Turn (Note[x], Note[x+1]) [3] \}$ 。

需要注意的是, 该集合的元素数量不一定是 3, 只是元素的最大数量可以等于 3。

5. 跳指: 以上 4 种基本指法都有标准的使用范围, 如果出现一组不满足任何一组限定条件的输入, 则采用跳跃的形势完成弹奏

四、难度定义

难度定义: 由于各类指法在弹奏方式上本质上的不同, 我们的讨论将分为两个部分:

1. 各指法内部的相对难度定义

基本指法, 缩指, 扩指, 跳指: 它们都属于指法走向与音调走向一致的顺序指法, 其弹奏难度可以统一定义为: $D[x+1]=|Finger[x+1] - (Finger[x] + Note[x+1] - Note[x])|$,即基本指法与实际所用指法的差值。(试想: 弹奏没有转位的 C 大调琶音是几乎没有难度要求的)

轮指: 轮指的动作的主要组成部分是手腕的转动, 因此轮指的难度定义应该是与手腕的转动角度有关的。在进行最大程度的转指时 ($\Delta Note = Turn_Lim$), 手腕的转动角度是一致的, 与前一个音的所用指法无关。我们最后得到的难度定义方式是:

$$D[x+1] = \frac{|Note[x+1] - Note[x]|}{Turn_Lim[Finger[x]]} \quad (\text{上行时的表达式})$$

$$D[x+1] = \frac{|Note[x+1] - Note[x]|}{Turn_Lim[Finger[x+1]]} \quad (\text{下行时的表达式})$$

2. 各指法之间的相对难度:

选择指法的第一步就是选取合适的基础指法（弹奏方式），有时就必须在扩指和轮指中进行取舍。只有在清楚各指法之间的相对难度关系后，才能合理地选择出最有效的弹奏方式。接下来讨论的是会关系到选择原则的几种指法之间的相对难度：

跳指和其它指法：毫无疑问，跳指的相对难度应该是远高于任何一种其它基础指法的。跳指是在前四种指法均无法满足弹奏需求时的最后手段跳指应该被尽量避免，因为它往往会干扰弹奏的连续性。一套优秀的指法应该尽量避免跳指的出现。

轮指和扩指：轮指和扩指之间的相对难度可以通过比较两者需要的手腕转动程度来比较。轮指所需要的手腕转动角度往往是大于 45° 接近 90° ；相比之下，扩指所需要的手腕转动角度往往极小，就算连续几次接近极限的扩指，叠加起来的转动角度也是小于一般轮指的。因此，我们的得到结论：虽然无法精确定义轮指与顺序指法之间难度的比例，但是单次轮指的难度应是大于单次甚至是多次叠加的扩指难度的。在指法分配中，应该尽可能多地使用前三种指法，来换取最少的轮指数量。

缩指，扩指和基本指法：鉴于基本指法可以说是缩指和扩指的一个分界点，它们之间的相对难度可以被统一为刚才用于分别描述三种指法难度的公式。

3. 顺序指法和非顺序指法:

对于扩指，基本指法，缩指，由于它们难度计算方式基本相同，并且都拥有指法走向与音符走向的一致性，我们统一规定它们为顺序指法。而轮指，则被视为非顺序指法。

刚才，我们已经将钢琴弹奏规范为了几种基础指法：基本指法，扩指，缩指，轮指，跳指。每一种基础指法都有其对应的函数，可以通过上一个音的弹奏指法以及已知乐谱导出当前音的弹奏指法。同时，关于几种弹奏方式的相对难度定义也导出了弹奏的基本原则：

1. 最高原则：保证弹奏的连续性
2. 次要原则：尽量减少轮指的使用次数

五、音域网络建立

对于一个由 i 个音符组成的乐谱，建立一个由 i 层，每层 5 个元素组成的加权网络。

每一层代表一个音符，5 个元素分别对应 5 个指法（1,2,3,4,5）。每一层的每一个元素都分别于上一层以及下一层的共十个元素相连接，它们之间的权的计算方式如下：


对于第 i 层第 X 个元素和第 $i+1$ 层第 Y 个元素之间的权，视为第 i 个音符用 X 指法弹奏，第 $i+1$ 个音符用 Y 指法弹奏时的弹奏难度，即 $D[x+1]$ 。

如果该种弹奏方式是顺序指法，即：

$$(Note[x+1] - Note[x]) \times (Y - X) \geq 0$$

则接下来判定该种指法是否符合缩指，扩指或基本指法的要求，在上行的情况下即：

$$Y < X + Note[i+1] - Note[i], \Delta Note \leq K_Lim(Finger[x], Finger[x+1])$$


 扩指

$Y = X + \text{Note}[i+1] - \text{Note}[i]$, $\Delta \text{Note} \leq K_Lim(\text{Finger}[x], \text{Finger}[x+1])$ 基本指法
 $Y > X + \text{Note}[i+1] - \text{Note}[i]$, $\Delta \text{Note} \leq K_Lim(\text{Finger}[x], \text{Finger}[x+1])$ 缩指
 $\Delta \text{Note} > K_Lim(\text{Finger}[x], \text{Finger}[x+1])$ 两个元素不相连

判断出模式后，就可以根据相对应的方式求解权值：

$$D[x+1] = |\text{Finger}[x+1] - (\text{Finger}[x] + \text{Note}[x+1] - \text{Note}[x])|$$

如果该种弹奏方式是非顺序指法，即：

$$(\text{Note}[x+1] - \text{Note}[x]) \times (Y - X) < 0$$

则判断它是否符合轮指条件，即是否符合 Turn_Up 或是 Turn_Down 所规定的输入与输出的映射关系。只有在符合映射关系时，才说明两个元素间的连线成立。

在成立的情况下，两者间的权重计算方式需要在原来的难度计算方式上进行修正。为了体现轮指远大于其他顺序指法的难度的事实，我们将原来计算的弹奏难度乘以 100，这样可以保证绝对不会出现多次连续顺序指法叠加后难度大于单次轮指的情况：

$$D[x+1] = \frac{|\text{Note}[x+1] - \text{Note}[x]|}{\text{Turn_Lim}[\text{Finger}[x]]} \times 100 \quad (\text{上行时的表达式})$$

$$D[x+1] = \frac{|\text{Note}[x+1] - \text{Note}[x]|}{\text{Turn_Lim}[\text{Finger}[x+1]]} \times 100 \quad (\text{下行时的表达式})$$

这样，我们就成功建立出了一个与乐谱对应的加权网络了。如果将每个元素理解为一个点，两个元素之间的权理解为两个点之间的距离，则选取最优指法就可以被转化成一个寻找从第一层通向最后一层的最短距离的问题。

六、问题求解（确定节点）

如果网络的层数小于 10，那么通过深度搜索算法就可以暴力求解出最优路径，也就是最理想的指法。

但在绝大多数情况下，需要连续弹奏的音符数量远大于 10。为了减小算法的复杂度，就需要进行适当地剪枝。具体来讲，我们可以通过在这个网络中设置必须要经过的节点来达到对整个网络进行分段求解，最终减小算法复杂度的目的。接下来，我们就将关于如何设置节点进行具体的讨论。

在几种基础指法中，轮指可以有效地起到设置节点的作用。对于上行的轮指来说，它需要满足弹奏的前一个音使用的指法是 1 指；对于下行的轮指来说，它需要满足当前弹奏的音使用指法为 1 指。这样，如果我们能够确定在第 i 层使用轮指，就能将第 i 层的第一个元素设置成为节点。而确定轮指使用位置的方法，则应该遵循“尽量少地使用轮指”的原则。

在正常情况下，迫不得已使用轮指的情况有两种：

1. 两个音的间距超出了所允许的最大扩指程度，即：

$$\Delta \text{Note} \leq K_Lim(\text{Finger}[x], \text{Finger}[x+1])$$

2. 上行或下行时候手指不够用，即：

$$\text{Finger}[x]=5, \text{Note}[x+1] > \text{Note}[x] \quad \text{或}$$

$$\text{Finger}[x]=1, \text{Note}[x+1] < \text{Note}[x]$$

体现在音域网络上，就是已经“无路可走”。此时，就需要搜寻从上一个节点到当前音符之间的所有音符，检查是否有符合转指条件的音符，然后回溯，将其定制成为转点。

对于正常人来说，扩指的极限随着所用指法的递增而减小，随着所用指法差的增大而增大，即对于任何 $1 \leq M, N \leq 5$ 满足下述规律：

$$\begin{aligned} K_Lim(M,N) &< K_Lim(M,N+1) \\ K_Lim(M,N) &> K_Lim(M+1,N+1) \end{aligned}$$

这两个公式的意义对于网络来说，就是：

在上行时，越靠右的元素越容易出现“无路可走”的情况。

在下行时，越靠左的元素越容易出现“无路可走”的情况。

因此，为了避免转指的出现，在初始选择指法的时候，应该尽量地采用极限状态的扩指（比如在上行的时候，能用 2 指就不用 3 指）。体现在网络上，就是尽量地避免水平方向上的位移。

按照这种原则进行选择，我们仍就会碰到“无路可走”的情况。从上一个节点到按照这种选择原则得到的断点，就是至少会有一次轮指发生的音符范围。

搜索这个范围，往往会发现很多层符合轮指的条件。还是根据“尽量较少转指数量”的原则，我们可以得到 1 个轮指的筛选条件，即尽量使用 4 指和 1 指之间的轮指。通过这种方式，可以创造出最大的“负水平位移”来抵消之前通过顺序指法积累的水平位移，达到尽量避免水平位移的目的，也因此能够有效减少转指数量。虽然 4,1 之间的轮指难度要大于 3,1 或 2,1 之间的轮指难度，但是两次低难度轮指的难度之和，是绝对会大于一次高难度轮指的难度的。在经过以上方法的筛选后，我们就能够找到目前看来最有效的使用轮指，也就是设定节点的位置了。

接下来，在不考虑和弦弹奏的情况下，我们面临的问题还有两个：

1. 如何确定初始音的指法
2. 对于有些靠近上行下行的交界点的音符，使用 4,1 之间的轮指并没能成为最有效的指法。原因在于：我们之所以使用 4,1 之间的轮指，是建立在 4,1 的轮指可以有效减少轮指数量的基础上的。但在面临节点时，由于音调的走向改变，3,1 指法并没有比 4,1 指法“更容易导致无路可走的局面”。
3. 与问题 2 类似，对于靠近上行下行交界点的音符，“最小水平位移”的定义将变得有些暧昧。极限扩指不再成为了最好的途径，因为上行时的扩指对于之后的下行可能会起到负面作用。

解决第一个问题，我们当然可以分别尝试每一种指法，然后计算每一种初始指法对应的最小路径（弹奏难度），然后互相比，得出最优结论。但实际上还有一种更加巧妙的方法：人在选择初始指法时，总是会尝试着将手拜访在能够令之后弹奏的音尽量靠近自己的手的地方。当然，所谓“之后弹奏的音”是有限制的，如果在之后的弹奏中出现了两个绝对不可能出现在同一个节点范围里连续弹奏的音（两个音之间的音距超过了手的最大跨度，比如说 10），那之后的音就完全不用考虑了。

具体来讲，在所选指法确定以后，就会有一个人手在自然状态下（默认为基本指法）的覆盖范围。之后计算考虑在内的音符中落在范围外的音符到覆盖范围的距离之和，当它值最小，那对应的就应该是理想指法。

判定考虑音符范围的代码如下：

```
up=down=Finger[current];
i=current;
While(up-down<=10)
{ i++;
```

```

        if(Finger[i]>up)
            up=Finger[i];
        if(Finger[i]<down)
            down=Finger[i]
        }
        return i-1;

```

i-1 即是需要考虑范围。具体的计算过程是：

```

        for(x=1;x<=5;x++)
        {   num=0;
            for(y=current+1;y<=i;y++)
            {
                if (Note[y]<Note[current]-(x-1))
                    num=num+(Note[current]-(x-1))-Note[y];
                if (Note[y]> Note[current]+(5-x))
                    num=num+(Note[y]- (Note[current]+(5-x)));
            }

```

通过打擂台，即可得带最终的理想指法。

对于第二个问题，我们需要适当地修正之前的轮指选择策略，得到最优的轮指位置。难题在于如何解决上行与下行接口处的问题。一个有效的方式是利用我们刚才用来求解第一个音符理想指法的方式。更实际来讲，这个“理想指法”实际上是最方便之后的音符弹奏的理想指法。如果我们将它运用到之后的每一个音符上，就可以得到之后每一个音符关于那个音符之后音符的理想指法。由于这个指法对于上行下行都是兼容的，它的输出是会受上下行的接口影响的。

以上行为例，对于远离交界点的音符，判断理想指法的过程的输出将始终保持为 1，因为一直需要保证之后的上行音符尽可能多地落在手的覆盖范围内。对于交界点的指法，输出则正好相反，总是会维持在 5。对于不断接近交界点的音符，理想指法的过程的输出将逐渐经历一个从 1 到 5 的过度过程。

这种过程，正好可以用来修正之前我们的轮指筛选条件：以上行为例，只要该指法的理想指法仍旧是 1，那选取它作为轮指位置就可以被考虑为能够有效较少之后的轮指次数（试想，如果有 5 个连续的音符，分别是 1,2,3,4,5。此时，第一个音符的理想指法应该是 1 指。如果选择在第一个音之前进行轮指，那弹奏第一个音的实际指法就会大于 1，这样就会在之后的弹奏中出现因为“手指不够用”引发的断路。）

因此，我们需要在“尽量靠后”的原则上加上一条优先级别更高的原则：轮指后的指法应该尽量接近该指法对应的理想指法。在保证该条原则的基础上再执行“尽量靠后”原则，就可以得到最优的轮指位置，也就是最佳的节点。

最后一个问题，解决方法仍旧是理想指法的运用。我们可以计算交界点出的理想指法，然后将其定义为一个新的节点。这样，就可以有效地避免上行的扩指对之后的下行弹奏起到相反作用的情况了。

在解决以上几个问题后，我们就可以很清晰的定义出我们所需要的节点，然后分段利用深度搜索算法进行局部的求解了。

七、和弦弹奏初探

对于和弦的研究，分为和弦与单音的连接弹奏和和弦内部的连接弹奏。

对于第一种情况，做法是根据音走势的上下行模式，按照其他音的走势对和弦进行拆分。之后再按照单音的方式进行指法的选择。唯一的一点不同在于：对于由一个和弦拆分出来的音内部，不允许出现轮指。在回溯搜索可能出现的轮指时，应该去除出现在和弦内部的可能轮指位置。

而对于和弦之间的连续弹奏，因为四种基础指法不再试用，所以对应单音的指法选择模式无法直接推广到和弦之间的连续弹奏上。对于这一部分的问题，还有待继续探究。

八、算法实现（C 语言）

由于和弦之间的连续弹奏模式还在研究中，所以该算法只包括了单音连续弹奏和单音与和弦之间的弹奏。输入方式：

1. 首先输入总音符数量。
2. 依次输入以中央 C 位为 1 的简谱形式乐谱。

```
#include<stdio.h>
#include<math.h>
void enterin(int n, double *pitch) //输入指法
{
    int i;
    for (i=1;i<=n;i++)
    {
        scanf("%lf",pitch+i);
    }
}

int turn(int start,double *pitch,int *turningpoint, int n )//预处理转折点
{
    int a,b;
    double diff;
    int z=1; //统计个数
    a=start;
    b=a+1;
    diff=(*(pitch+a))-(*(pitch+b));
    for(a=start+1;a<=n-1;a++)
    {
        b=a+1;
        if(diff*((*(pitch+a))-(*(pitch+b)))<0)
        {
            z++;
            *(turningpoint+z)=a;
        }
        diff=(*(pitch+a))-(*(pitch+b));
    }
```

```

    }
    return z;
}

```

int idealfinger(int *finger,double *pitch, int now, int **storeidealfinger,int end,int mode) //测定理想指法

```

{
    int i=now;
    double up=*(pitch+i);
    double down=*(pitch+i);
    int up_num=0;
    int down_num=0;
    int result;
    if (now==end)
    {result=(*(finger+now-1)+*(pitch+now)-*(pitch+now-1));
        if (result>=1&&result<=5)
            return result;
        if(result<1)
            return 1;
        if(result>5)
            return 5;
    }
    while(up-down<=8)
    {
        i++;
        if (i>end)
            break;

        if(*(pitch+i)>up)
        {
            up=*(pitch+i);
            up_num++;
        }
        if(*(pitch+i)<down)
        {
            down=*(pitch+i);
            down_num++;
        }
    }
    i--;
}

```



```

int x;
int y;
int min=100;
double num;

for(x=1;x<=5;x++)
{
    num=0;
    for(y=now+1;y<=i;y++)
    {
        if ((* (pitch+y))<(* (pitch+now)-(x-1)))
            num=num+(* (pitch+now)-(x-1))-(* (pitch+y));
        if ((* (pitch+y))>(* (pitch+now)+(5-x)))
            num=num+(* (pitch+y))-(* (pitch+now)+(5-x));
    }
    if (num<min)
    {
        min=num;
        result=x;
    }
}
int z=0;
for(x=1;x<=5;x++)
{
    num=0;
    for(y=now+1;y<=i;y++)
    {
        if ((* (pitch+y))<(* (pitch+now)-(x-1)))
            num=num+(* (pitch+now)-(x-1))-(* (pitch+y));
        if ((* (pitch+y))>(* (pitch+now)+(5-x)))
            num=num+(* (pitch+y))-(* (pitch+now)+(5-x));
    }

    if (num==min)
    {
        z++;

        *(* (storeidealfinger+now)+z)=x;
    }
}

if (mode==0)
return result; //返回得到的第一个理想指法

```

```

        if(mode==1)    //返回理想指法的个数
        return z;

    }

int difficulty_add(double *pitch,int pitch_now,int *finger,int finger_now,int **D)
{
    double z;
    int difficulty;
    if
    ((*pitch+pitch_now)-*(pitch+pitch_now-1))*(*(finger+finger_now)-*(finger+finger_now-1))>0||
    (*(pitch+pitch_now)-*(pitch+pitch_now-1))*(*(finger+finger_now)-*(finger+finger_now-1))>0==0)
    {
        z=*(pitch+pitch_now)-*(pitch+pitch_now-1);
        int a,b,temp;
        a=finger[finger_now-1];
        b=finger[finger_now];
        if (a>b)
        {temp=a;
        a=b;
        b=temp;}
        difficulty=*(D+a)+b);
        if (z<0)
        {
            z=-z;
        }
        if (z>difficulty)
        return 1;
        else
        return 0;
    }
    else
    {
        return 0;
    }
}

```

//基于搜索找到理想节点实行转指

```

int ThreeRuleTurn_Up(int end, int start, int last,double *pitch, int *finger,int **storeidealfinger )
{
    int rawresult[10];
    int processresult[10];
    int finalresult=-1;
    int diff=100;

```

```

int max=0;
int x;
int y=0;
if(*(finger+start)!=1)
{
    if (last<end)
    {
        for(x=start;x<=last;x++) //粗略筛选出结果 (rawresult ( ) 储存的是转位组的第一个
音
        {
            if (*(pitch+x+1)-*(pitch+x)<=3&&*(finger+x)!=1&&*(finger+x)!=5)
            {
                if(floor(*(pitch+x))<*(pitch+x))
                {y++;
                rawresult[y]=x;
                }
            else
            {
                if(floor(*(pitch+x+1))==*(pitch+x+1))
                {
                    y++;
                    rawresult[y]=x;
                }
            }
        }
    }
}
else
{
    for(x=start;x<=last-1;x++) //粗略筛选出结果 (rawresult ( ) 储存的是转位组的第一
个音
    {
        if (*(pitch+x+1)-*(pitch+x)<=3&&*(finger+x)!=1&&*(finger+x)!=5)
        {
            if(floor(*(pitch+x))<*(pitch+x))
            {y++;
            rawresult[y]=x;
            }
        else
        {
            if(floor(*(pitch+x+1))==*(pitch+x+1))
            {
                y++;
                rawresult[y]=x;}
        }
    }
}

```

```

    }
    }
    }
    }
}
else
{
    if (last<end)
    {
        for(x=start+1;x<=last;x++) //粗略筛选出结果 (rawresult ( ) 储存的是转位组的第一
一个音
        {
            if (*(pitch+x+1)-*(pitch+x)<=3&&*(finger+x)!=1&&*(finger+x)!=5)
            {
                if(floor(*(pitch+x))<*(pitch+x))
                {y++;
                rawresult[y]=x;
                }
            }
            else
            {
                if(floor(*(pitch+x+1))==*(pitch+x+1))
                {
                    y++;
                    rawresult[y]=x;}
            }
        }
    }
}
else
{
    for(x=start+1;x<=last-1;x++) //粗略筛选出结果 (rawresult ( ) 储存的是转位组的第一
一个音
    {
        if (*(pitch+x+1)-*(pitch+x)<=3&&*(finger+x)!=1&&*(finger+x)!=5)
        {
            if(floor(*(pitch+x))<*(pitch+x))
            {y++;
            rawresult[y]=x;
            }
        }
        else
        {
            if(floor(*(pitch+x+1))==*(pitch+x+1))
            {
                y++;
            }
        }
    }
}

```

```

        rawresult[y]=x;}
    }
}
}
}
int z=0;
for(x=1;x<=y;x++)
{
    if((idealfinger(finger,pitch,rawresult[x]+1,storeidealfinger,end,0)-1)<=diff)
    {
        diff= (idealfinger(finger,pitch,rawresult[x]+1,storeidealfinger,end,0)-1);
    }
}
for(x=1;x<=y;x++)
{
    if((idealfinger(finger,pitch,rawresult[x]+1,storeidealfinger,end,0)-1)==diff)
    {
        z++;
        processresult[z]=rawresult[x];
    }
}
int min=0;
for(x=1;x<=z;x++)
{
    if (processresult[x]>min)
        min=processresult[x];
}
if (min==0)
{
    return 0;
}
else
    return min;
}
//基于搜索的段落优化
int Search(double *R_D,int *finger,int level,double *pitch,int point,int now,int n,int **R,int
storey,int i,int *last,int **D,int **storeidealfinger_,int mode)
{
    int x,y;
    storey++;
    for(x=point+1;x<=now;x++)
    {
        i++;

```

```

int *the=*(R+i);
if(storey==1)
{for(y=point;y<=x-1;y++)
*(the+y-point+1)=*(last+y);
if (mode==1)
*(the+x-point+1)=*(last+x)+1;
else
*(the+x-point+1)=*(last+x)-1;
}
else
{for(y=point;y<=x-1;y++)
*(the+y-point+1)=*(last+y-point+1);
if(mode==1)
*(the+x-point+1)=*(last+x-point+1)+1;
else
*(the+x-point+1)=*(last+x-point+1)-1;
}
y=x+1;
int success=1;
int difficulty=0;
while(y<=now)
{
if (mode==1)
{
if(*(pitch+y)>*(pitch+y-1))
*(the+y-point+1)=*(the+y-1-point+1)+1;
if(*(pitch+y)==*(pitch+y-1))
*(the+y-point+1)=*(the+y-1-point+1);
if (*(the+y-point+1)>5)
{
success=0;
break;
}
}
if(difficulty_add(pitch,y,the,y-point+1,D_)==1)
{
while(difficulty_add(pitch,y,the,y-point+1,D_)==1 && *(the+y-point+1)<5)
{
*(the+y-point+1)=*(the+y-point+1)+1;
}
if (difficulty_add(pitch,y,the,y-point+1,D_)==1)
{
success=0;
break;
}
}
}

```

```

    }
}
else
{
    if(*(pitch+y)<*(pitch+y-1))
*(the+y-point+1)=*(the+y-1-point+1)-1;
    if(*(pitch+y)==*(pitch+y-1))
*(the+y-point+1)=*(the+y-1-point+1);
    if (*(the+y-point+1)<1)
    {
        success=0;
        break;
    }
    if(difficulty_add(pitch,y,the,y-point+1,D_)==1)
    {
        while(difficulty_add(pitch,y,the,y-point+1,D_)==1 && *(the+y-point+1)>1)
        {
            *(the+y-point+1)=*(the+y-point+1)-1;
        }
        if (difficulty_add(pitch,y,the,y-point+1,D_)==1)
        {
            success=0;
            break;
        }
    }
}
y++;
}
y--;
if (success!=0)
{
    if (mode==1)
    level=idealfinger(finger,pitch,now,storeidealfinger_,n,0)-*(the+now-point+1);
    else
    level=-(idealfinger(finger,pitch,now,storeidealfinger_,n,0)-*(the+now-point+1));
}
else
level=level-100;

if (level==0)
{
    for(y=1;y<=now-point+1;y++)    //y change to the number of the series
    {
        difficulty=difficulty+the[1]+(pitch[y+point-1]-pitch[point])-the[y];
    }
}

```

```

    }
    *(R_D+i)=difficulty;
}
if (level<0)
{
    *(R_D+i)=-1;
}
if (level>0)
{
    i=Search(R_D,finger,level,pitch,point,now,n,R,storey,i,last,D_,storeidealfinger_,mode);
}
}
return i;
}

```

```

int main(void)
{
    double pitch[100];
    int finger[100];
    int storeidealfinger[100][6];
    int *storeidealfinger_[100];
    int D[6][6];
    int *D_[6];
    int R[3200][6];
    int *R_[3200];
    double R_D[3200];
    int turningpoint[100];
    int turningpoint_num;
    int n;

    int x,y,z,i;
    int difficulty;
    for(x=1;x<100;x++)
    {
        storeidealfinger_[x]=storeidealfinger[x];
    }
    for(x=1;x<=5;x++)
    {
        D_[x]=D[x];
    }
    for(x=1;x<=3200;x++)
    {
        R_[x]=R[x];
    }
}

```



```
}
```

```
D[1][5]=9;D[2][5]=6;D[3][5]=3;D[4][5]=2;D[5][5]=0;D[1][4]=8;D[2][4]=3;D[3][4]=2;D[4][4]=0;D[1][3]=6;D[2][3]=3;D[3][3]=0;D[1][2]=5;D[2][2]=0;D[1][1]=0;
```

```
int numidealfinger[100];
```

```
scanf("%d",&n);
```

```
enterin(n,pitch);
```

```
turningpoint_num=turn(1,pitch,turningpoint,n)+1;
```

```
turningpoint[1]=1;
```

```
turningpoint[turningpoint_num]=n;
```

```
difficulty=0;
```

```
int point=1,min,result;
```

```
int next=2;
```

```
int level;
```

```
finger[1]=idealfinger(finger,pitch,1,storeidealfinger_,n,0);
```

```
x=2;
```

```
int h;
```

```
while(x<=n)
```

```
{
```

```
    y=0;
```

```
    if(pitch[x]>pitch[x-1])
```

```
        finger[x]=finger[x-1]+1;
```

```
    if(pitch[x]==pitch[x-1])
```

```
        finger[x]=finger[x-1];
```

```
    if(pitch[x]<pitch[x-1])
```

```
        finger[x]=finger[x-1]-1;
```

```
if (turningpoint_num>0)
```

```
{
```

```
    for(y=1;y<=turningpoint_num;y++)
```

```
    {
```

```
        if(x<=*(y+turningpoint))
```

```
            break;
```

```
    }
```

```
}
```

```
if (y!=0)
```

```
{
```

```
    if(point<*(turningpoint+y))
```

```
    point=*(turningpoint+y-1);
```

```
}
```

```
if (finger[x]>5)
```

```

{
    z=ThreeRuleTurn_Up(n,point,x-1,pitch,finger,storeidealfinger_);
    if(z!=0)
    {finger[z+1]=1;
    x=z+1;
    point=x;
    }
    else
    {
        finger[x]=idealfinger(finger,pitch, x,storeidealfinger_,n,0);
        point=x;
    }
}

```

```

if(finger[x]<1)
{
    finger[x]=idealfinger(finger,pitch,x,storeidealfinger_,n,0);
    z=pitch[x-1]-pitch[x];
    if (z<3)
    {
        if (finger[x]>4)
            finger[x]=4;
    }
    else
    {

```

```

        if (z<4)
        {
            if(finger[x]>3)
                finger[x]=3;
        }
        else
        {
            if (z<5)
            {
                if (finger[x]>2)
                    finger[x]=2;
            }
        }
    }
    point=x;
}

```

```

if (difficulty_add(pitch,x,finger,x,D_)==1)

```

```

{
    if (pitch[x]-pitch[x-1]>=0)
    {
        while(difficulty_add(pitch,x,finger,x,D_)==1 && finger[x]<5)
        {
            finger[x]=finger[x]+1;
        }
        if (difficulty_add(pitch,x,finger,x,D_)==1)
        {
            z=ThreeRuleTurn_Up(n,point,x-1,pitch,finger,storeidealfinger_);
            if (z!=0)
            {finger[z+1]=1;
            x=z+1;
            point=x;
            }
        else
        {
            finger[x]=idealfinger(finger,pitch, x,storeidealfinger_,n,0);
        }
    }
    else
    {
        while(difficulty_add(pitch,x,finger,x,D_)==1 && finger[x]>1)
        {
            finger[x]=finger[x]-1;
        }
        if (difficulty_add(pitch,x,finger,x,D_)==1)
        {
            z=ThreeRuleTurn_Up(n,point,x-1,pitch,finger,storeidealfinger_);

            if (z!=0)
            {finger[z+1]=1;
            x=z+1;
            point=x;
            }
        else
        {
            finger[x]=idealfinger(finger,pitch, x,storeidealfinger_,n,0);
        }
    }
}

```

```

if (x==turningpoint[next])
{
    if (pitch[x]-pitch[x-1]>=0)
    {
        level=idealfinger(finger,pitch,x,storeidealfinger_,n,0)-finger[x];;
        if (level>0)
        {
            i=Search(R_D,finger,level,pitch,point,x,n,R_,0,0,finger,D_,storeidealfinger_,1);
            min=100;
            for(y=1;y<=i;y++)
            {
                if (*(R_D+y)<min)
                {
                    result=y;
                    min=*(R_D+y);
                }
            }
            for(y=point;y<=x;y++)
            {
                finger[y]=R[result][y-point+1];
            }

        }
    }
    else
    {
        level=-(idealfinger(finger,pitch,x,storeidealfinger_,n,0)-finger[x]);

        if (level>0)
        {
            i=Search(R_D,finger,level,pitch,point,x,n,R_,0,0,finger,D_,storeidealfinger_,0);

            int difficulty;
            for(y=1;y<=x-point+1;y++)    //y change to the number of the series
            {
                difficulty=difficulty+finger[point]+(pitch[y+point-1]-pitch[point])-finger[y];
                R[0][y]=finger[y+point-1];
            }
            R[0][0]=difficulty;
            min=100;
            for(y=0;y<=i;y++)
            {
                if (R[y][0]<min)
                {

```

```

        result=y;
        min=R[y][0];
    }
}

for(y=point;y<=x;y++)
{
    finger[y]=R[result][y-point+1];
}
}
}
if (next<turningpoint_num)
    next++;

}
x++;

}
printf("\n\n\n\n\n");
for(x=1;x<=n;x++)
printf("%d\n",finger[x]);
scanf("%d",&x);
return 0;
}

```