算法分析与设计第四次作业

黄丛宇 2010212439

October 21, 2010

1 实验环境

- CPU: Intel(R) Core(TM)2 Duo CPU T5870 2.00GHz
- MEM: 1GB
- OS : Debian 5.0 （1GB swap）
- Java: java version "1.6.0_21"

2 Exercise 15.4-4

由算法可知，在算法执行过程中，算法每次循环只使用当前行和前一行的数据。因此，可以使用一个只有两行的二维滚动数组来存储数据。另外，使用一个变量，标记那一行是当前行，则另一行是前一行。由于算法中两个循环谁在外边谁在里面不应想算法的正确性，因此可以min(m,n)放在内循环中。这样，滚动数据只需要2*min(n,m) 的长度，外加一个$O(1)$的标记变量。

可以进一步将两行的二维数组变成一个一维的数组。观察算法可得，内循环的每次运算中，假如当前要计算的元素是c[i,j]，那么计算只使用了c[i-1, j-1]，c[i-1, j]和c[i, j-1]。当使用一维数组的时候，假设当前计算的元素是c'[i]，那么c'[i - 1]存放的就是原来的c[i-1, j]，而当前的c'[i]中存放的是c[i, j-1] 的值，现在只缺少c[i-1, j-1]，这个值恰好就是c[i-1]中存放的前一个值。因此，可以使用一个变量，在计算c[i-1]的时候，将c[i-1]的前一个值保存起来，给c[i]使用。同样，计算c[i]的时候，在覆盖c[i]之前，将其值保存在这个变量中。同理，数组的长度为min(n,m)，因此，算法只需要min(n,m)长度的数组外加一个 $O(1)$的标记变量。

3 Exercise 15.4-6

定义b[k]表示以s[k]结尾的最长递增子序列的长度，则状态转移方程如下：

$$b[k] = max(max(b[j]|s[j] < s[k], j < k) + 1, 1);$$

在a[k]前面找到满足a[j]<a[k]的最大b[j],然后把a[k]接在它的后面，可得到以a[k]结尾的最长递增子序列的长度，或者a[k]前面没有比它小的a[j]，那么这时a[k]自成一序列，长度为1。最后整个数列的最长递增子序列即为max(b[k]|0<=k<=n-1);

在寻找最大的b[j]的时候，如果使用顺序查找，则算法复杂度为$O(n^2)$，因此使用二分查找降低时间复杂度。

引入一个新的数组c。c中元素满足$c[b[k]] = a[k]$,即当递增子序列的长度为b[k]时子序列的末尾元素为 $c[b[k]] = a[k]$。算法中对c的修改可以保证c是有序的。如果有多个相同长度的递增子列，那么对应的位置存放的是最后出现的那个子列的最后一个元素。c[1]=s[0]，c[0]=0。c[0]作为二分查找的哨兵使用。

核心代码如下：

```
public static int getLISLen(final int[] s, int[] lis)
{
    if (null == s) {
        return -1;
    }
    c = new int[s.length + 1];
    cindex = new int[s.length + 1];
    pre = new int[s.length];

    //初始化
        cindex[0] = -1;
        for(int i = 0; i < s.length; ++i){
            pre[i] = -1;
            cindex[i + 1] = -1;
        }

        c[0] = 0; //这个元素作为一个哨兵。在二分查找中使用。
        c[1] = s[0];
        cindex[1] = 0;
        len = 1; //此时只有c[1]求出来，最长递增子序列的长度为1.
    int j;
    for(int i = 1; i < s.length; ++i){
        j = binarySearch(c, len, s[i]);
        c[j] = s[i];
        cindex[j] = i;
        /*
         * 以s[i]结尾的最长子串的倒数第二个元素是c[j-1]。
         */
        pre[i] = cindex[j - 1];
        if(len < j){
            len = j;
```

```
            lastIndex = i;
        }

    }
    getSubsquence(s, lis);
    return len;
}
/**
 * 二分查找。返回值表示n在数组a中的位置。如果在数组中有元素等于n
 * 那么返回最后一个等于n的元素的下一个位置。
 * @param a 数组a
 * @param len 数组a中数据的个数
 * @param n 需要查找的元素
 * @return
 */
private static int binarySearch(final int[] a, int len,
     int n)
{
    if (n < 0) {
        return -1;
    }

    int left = 0, right = len;
    int mid = (left + right) / 2;

    while (left <= right) {
        /*
         * 等于是为了处理"两个相等的元素"也是递增序列的情况
         */
        if (n >= a[mid]){
            left = mid + 1;
        }
        else if (n < a[mid]){
            right = mid - 1;
        }

        mid = (left + right) / 2;
    }
    return left;
}

/**
 * 构造其中一个最长递增子列。
 * @param s 原始序列。
```

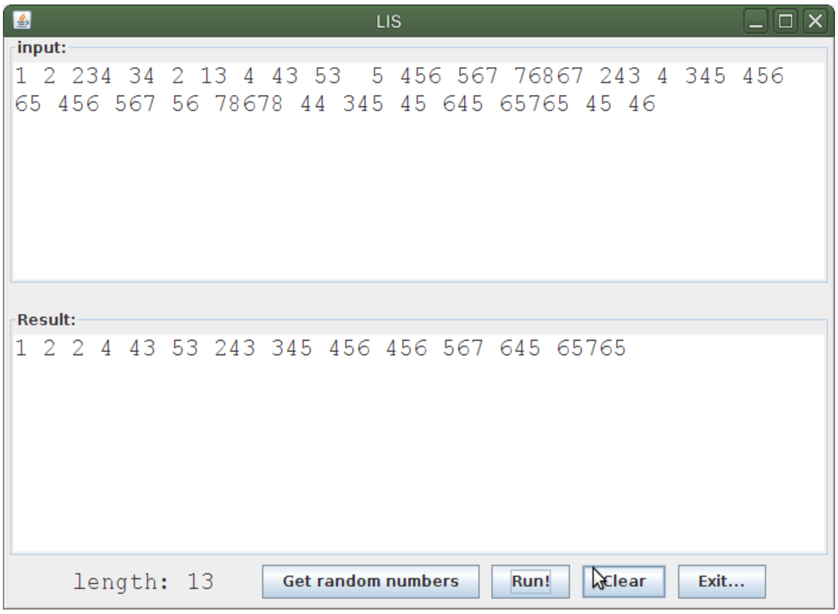
```java
 * @param lis 最长子列
 */
private static void getSubsquence(final int[] s, int[] lis
    )
{
   int pr;
   int index = len;
   pr = lastIndex;
   do{
      lis[--index] = s[pr];
      pr = pre[pr];
   }while(pr != -1);
}

//最长递增子列的长度
private static int len = 0;
//最长递增子列最后一个元素的位置。
private static int lastIndex = -1;
/*
 * c[i]=a[j，]表示c[i]中存储的是长度为i的最长递增子列的最后一个元
     素。
 * 并且，c中存放的就是最长递增子列。
 * c从1开始，c[0]最为哨兵在二分搜索中使用
 */
private static int[] c;
/*
 * cindex[i]存储c[i]对应的元素在序列中的位置。
 */
private static int[] cindex;
/*
 * pre[i]表示s[i]所在的最长递增子列的前一个元素的位置。
 * 注，这个最长子列可能不是s的最长子列，只是包含s[i]中所有
 * 递增子列最长的。
 */
private static int[] pre;
```

使用一个pre数组保存每个元素所在的最长递增子列的前一个元素的位置。pre[i]表示s[i]所在的最长递增子列中，s[i]前一个元素的下标。使用pre数组，可以在$O(n)$的时间内构造出一个最长的递增子列。

运行结果如下：

Figure 1 运行结果



4 Problem 15-1 bitonic tours

对所有点按x坐标排序。$T(i)$表示前i个点的最短路径，那么$T(i)$可以从$T(i-1)$构造出。从 $T(i-1)$中选出一条边，使得从$T(i-1)$删除这条边并加上这条边的两个端点到点i的两条边之后，路径距离最短。从$T(i-1)$删除这条边并加上两条新边后就得到了$T(i)$。

算法如下：

```
对所有的点按x坐标排序，将排好序的点存入数组p。
path存储当前最短路径。
path_len为当前最短路径的长度

边(p[0],p[1])存入path中。
path_len等于边(p[0],p[1])的长度

for(i从1到n-1)
    for(path中的每一条边(p[x], p[y]))
        计算从path删除边(p[x], p[y]),并加上边(p[x], p[i])和
        边(p[i], p[y])后的路径长度。
        记录这个路径的最小值和要删除的边。
    从path中删除这个边，并加上新的两条边。
    更新path_len的值。
```

最终，path中存储最短路径的所有边，path_len存储最短路径长度。

path可以使用一个双向连表，插入和删除的时间可以在$O(1)$内实现。排序算法使用快速排序，时间 $O(nlgn)$。外循环需要n-1次，内循环要i - 2次，因此，算法的时间复杂度是$O(n^2)$

5 Problem 15-6 checker

用一个n×n的二维数组c表示checkerboader。那么，对于格子c[i][j]，checker只可能从c[i][j-1]，c[i-1][j-1]和c[i+1][j-1]三个格子移动过来。用一个二维数组b表示到达每个格子所能获得的最大钱数。b[i][j]表示checker到达c[i][j]时所能获得的最大钱数。每一个格子的标号用i*n+j表示。可以得到如下的递推方程：

$$\begin{aligned} b[i][j] = max(&b[i-1][j] + p((i-1)*n+j, i*n+j) \\ &, b[i-1][j-1] + p((i-1)*n+j-1, i*n+j) \\ &, b[i-1][j+1] + p((i+1)*n+j+1, i*n+j)) \end{aligned}$$

算法如下：

```
maxValue = -1
b存储checker到达每个格子所得到的最大值。
for(i=1; i < n; ++i)
    for(j=0; j < n; ++j)
        b[i][j] = max(b[i-1][j]+p((i-1)*n+j, i*n+j)
                    ,b[i-1][j-1]+p((i-1)*n+j-1, i*n+j)
                    ,b[i-1][j+1]+p((i+1)*n+j+1, i*n+j));

for(i=0; i < n; ++i)
    if(maxValue < b[n-1][i])
    maxValue = b[n-1][i]
```

外层循环和内层循环分别是n-1次和n次，因此算法的复杂度是$O(n^2)$