算法分析与设计第四次作业

黄丛宇 2010212439

October 20, 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字符是c[j]。-1
       */
      pre[i] = cindex[j - 1];
      if(len < j){
         len = j;
```

```
            lastIndex = i;
        }

    }
    getSubsquence(s, lis);
    return len;
}
/**
 * 二分查找。返回值表示在数组中的位置。如果在数组中有元素等于nan
 * 那么返回最后一个等于的元素的下一个位置。n
 * @param a 数组a
 * @param len 数组中数据的个数a
 * @param n 需要查找的字符
 * @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;
}

//最长递增子列的长度
private static int len = 0;
//最长递增子列最后一个字符的位置。
```

```
private static int lastIndex = -1;
/*
 * c[i]=a[j，表示]c[i中存储的是长度为]的最长递增子列的最后一个字符。
     i
 * 并且，中存放的就是最长递增子列。c
 * 从开始，c1c最为哨兵在二分搜索中使用[0]
 */
private static int[] c;
/*
 * cindex[i存储]c[i对应的字符在字符串中的位置。]
 */
private static int[] cindex;
/*
 * pre[i表示]s[i所在的最长递增子列的前一个字符的位置。]
 * 注，这个最长子列可能不是的最长子列，只是包含ss[i中所有]
 * 递增子列最长的。
 */
private static int[] pre;
```

运行结果如下：

Figure 1 运行结果