算法基础 2023 春 Homework 3

任课老师：陈雪 & 邵帅 due: April 20, 23:59

作业要求：清晰简洁的伪代码，必要的时间复杂度分析，和必要的正确性分析。可以直接调用基本的数据库和已讨论过的算法/程序（如排序、找中位数、二分查找等）。

问题 1 (30 分)**.** 给定一个序列 $a_1, \cdots, a_n$, 设计不同的算法来找到最长的递增子序列 $s_1, \cdots, s_m$ 满足如下要求。

1. 如果有多个长度相同的子序列，输出第一个元素最小的序列；如果有多个序列同时有相同的长度和第一个元素，输出第二个元素最小的；以此类推。
2. 如果有多个长度相同的子序列，输出第一个元素最大的序列；如果有多个序列同时有相同的长度和第一个元素，输出第二个元素最大的；以此类推。

answer1:

1. 最长递增子序列中用到 3 个额外数组保存中间变量分别是 $leng, order, prefix$。
   其中 $leng[i]$ 表示以 $X[i]$ 结尾的最长子序列的长度，$order[i]$ 表示长为 i 的子序列的最后元素，$prefix[i]$ 表示打印最长子序列时 $X[i]$ 的前一个元素的下标。
   程序从前往后遍历 X 数组，并且利用 $order[j]$ 判断 $X[i]$ 是否大于长度为 j 的子序列的最后一个元素 ($order$ 数组始终保存相同长度下最后一个元素最小值，可以利用二分查找在 $\log(n)$ 时间内找到 j)，于是以 $X[i]$ 结尾的子序列长度最大为 $j+1$。
   利用 $prefix[i]$ 记录 $order[j]$ 的值, 利用栈逆向输出 $order$ 中最后一个元素记录在 $prefix$ 中的序列即为题目要求的序列。
   正确性：首先可以保证这样的输出使得在子序列后缀相同时，前面的元素符合最小的要求。这是因为每次发现长度相同而结尾元素更小的序列时，会更新 $order$。接着可以保证在子序列后缀不同时，也符合要求。这是因为 $order$ 中最后一个元素是长度最长的序列中，结尾元素最小的元素。而对于 X 中存在的多个长度相同的子序列，考虑他们的结束元素，可以发现元素小的在 X 中排在元素大的后面 (否则会有更长的子序列)。而对于在 X 后面的元素，他的前缀的选择是依据更新后的 $order$，可以保证在相同长度下元素不大于之前相同长度下的元素。所以正确性成立。
   程序设计如下：
2. 类似前一问，我们先将 X 翻转，即求最长递减序列，使相同长度序列的最后一个元素是最大的。需要修改 $order$ 中的元素，使他保存相同长度下的最大元素。

Algorithm 1 LIS1

```
function LIS1(X)
    n = X.length
    let leng[1...n], order[1...n] and prefix[1...n] be new tables
    for i = 1...n do
        leng[i] = 1
        order[i] = max
        prefix[i] = 0
    order[1] = X[1]
    for i = 1...n do
        L = 1, R = i
        while L ≤ R do
            m = (L+R)/2
            if order[m] < X[i] then
                prefix[i] = order1[m]
                L = m + 1
            else
                R = m − 1
        leng[i] = leng[prefix[i]] + 1
        if order[leng[i]] > X[i] then
            order[leng[i]] = X[i]
    for i = n...1 do
        if order1[i] > min then
            a = i
            let output be a stack
            output.push(order1[a])
            push elements in prefix[·] into stack inorder
            break
    print output
```

问题 2 (20 分)**.** 给定一个树 T 和覆盖半径 r，其中 T 的根节点为 $root$，每个节点 u 的孩子保存在 $c.child$ 序列中。找出 T 中最小的覆盖集（使得每个点到该集合的距离都 $\leq r$）。

answer2: 注意到每选择一个结点加入覆盖集，对于该结点下面的结点：他的子树中深度不超过 r 的结点全部可以被覆盖。对于该结点上面的结点：离该结点不超过 r 的结点会被覆盖。我们先找到深度最大的结点，并且选择可以把这个结点覆盖的深度最小的点。接着我们标记被覆盖的结点，去寻找未被覆盖的最深节点，重复上述过程，即可得到线性时间的算法。

下面证明贪心算法正确性：

1. 结点存在：因为一定有结点去把深度最大的结点覆盖，所以这样的结点存在；

2. 贪心选择包含在某些最优解中：对于某一最小覆盖集，如果他选择深度最小的结点去覆盖最深结点，那么我们的选择就是最优覆盖集的选择；如果他不选择深度最小的结点去覆盖最深结点，那么我们把这个结点从覆盖集中去掉，用深度最小的结点去覆盖最深结点，这时可以发现新的结点覆盖的范围比原结点大，新的集合也是覆盖集且大小不变，所以也是最优解；

3. 原问题和子问题有相同结构，可以归纳：子问题覆盖了除原问题贪心选择去掉结点以外的所有结点，因此子问题加上原问题的贪心选择可以合并出原问题的解；

4. 子问题的最优解可以还原原问题的最优解：令原问题为 P，假设贪心选择为 $\hat{c}$，P' 为子问题。由子问题最优解 π' 还原得到原问题的解 $\pi = \pi' \cup \hat{c}$。假设 π 不是最优解，并且存在最优解 π^*，那么由前知贪心选择也可以在与 π^* 相同规模的覆盖集中 (所以就不妨设为在 π^* 中)，所以存在子问题 P' 上有更优解 π^*。而这与 π' 为子问题 P' 上最优解矛盾。故子问题最优解可以还原原问题最优解。

Algorithm 2 Cover Set

function CoverSet(T, r)
    if $T.h \le r$ **then** return T
    else
        let d, *set*, *cover* be new tables
        add all nodes into $d[\cdot]$ in depth order by BFS
        for $i = n \dots 1$ **do**
            if $cover[i] == 0$ **then** // has not been covered
                add the deepest node's rth parent into the cover set // use the highest node to cover
                for all nodes around the rth parent in distance r , its $cover[\cdot]$ to be 1

问题 3 (25 分)**.** 给定一个背包其重量上限为 M 和 n 个物品，其中每个物品重量为 w_i，价值为 v_i。请选择一些物品使得其总重量不超过 M 的同时总价值最大。

同时回答：算法的时间是否为多项式时间吗？给出必要的理由。

answer3: 考虑将物品一个一个装入背包，可以有递推式：

$$val[i][j] = \max(val[i-1]\,[j - weight[i]] + value[i], val[i-1][j])$$

从而可以设计算法如下. 算法循环 $n \times M$ 次，故时间复杂度为 $O(nM)$。可是 M 循环的时间复杂度取决于 M 和物品重量的相对关系。比如在 M 大于全部物品重量之和时，就算只计算 M 循环中物品重量和的那些点 (原本是从 1 遍历到 M)，也需要遍历 2^n 个点 (为选择数的累加，即 $\Sigma_{i=0}^{n}\binom{n}{i}$)，故不为多项式级别的。算法如下：

Algorithm 3 Knapsack

function Knapsack(M, n, $weight$, $value$)
  let $val[0\ldots n][0\ldots M]$ be a new table
  for $j = 0\ldots M$ **do**
    $val[0][j] = 0$
  for $i = 1\ldots n$ **do**
    for $j = 1\ldots M$ **do**
      if $weight[i] \le j$ **then**
        if $val[i-1]\,[j - weight[i]] + value[i] \ge val[i-1][j]$ **then**
          $val[i][j] = val[i-1]\,[j - weight[i]] + value[i]$
        else
          $val[i][j] = val[i-1][j]$

问题 4 (25 分)**.** 为了将一个文本串 $x[1,\cdots,m]$ 转换为目标串 $y[1,\cdots,n]$，我们可以使用多种变换操作。我们的目标是，给定 x 和 y，求将 x 转换为 y 的一个变换操作序列。我们使用一个数组 z 保存中间结果，假定它足够大，可存下中间结果的所有字符。初始时，z 是空的，结束时，应有 $z[j] = y[j], j = 1, 2, \cdots, n$。我们维护两个下标 i 和 j，分别指向 x 中位置和 z 中位置，变换操作允许改变 z 的内容和这两个下标。初始时，$i = j = 1$。在转换过程中应处理 x 的所有字符，这意味着在变换操作结束时，应有 $i = m + 1$。

我们可以使用如下变换操作：

1. 复制：从 x 复制一个字符到 z，即进行赋值 $z[j++] = x[i++]$

2. 替换：将 x 中的一个字符替换为另一个字符 c，即 $z[j++] = c, i++$

3. 删除：删除 x 中一个字符，即 $i++$

4. 插入：将字符 c 插入中 z，即 $z[j++] = c$

每种变换操作每次执行都有一定的代价 $c_{copy}, c_{replace}, c_{delete}, c_{insert}$，而 x 到 y 的编辑距离是将 x 转换为 y 的最小的变换代价之和。设计动态规划算法，输入为文本串 x, 目标串 y 和每种操作的代价 $cost$，求 $x[1, \cdots, m]$ 到 $y[1, \cdots, n]$ 的编辑距离并打印最优操作序列。

answer4: 类似求最长公共子序列,我们用二维数组来保存求解过程。二维数组第一行/列代表空串，之后分别代表 x 和 y。将二维数组 $a[x.length + 1, y.length + 1]$ 的第一行和第一列初始化为 $c_{insert} \times n$(假设 c_{insert} 是从空串到 x y 编辑距离最小的方式)。四种变换操作对应二维数组上的操作如下:

1. 复制：从 $a[i, j]$ 到 $a[i + 1, j + 1]$;

2. 替换：从 $a[i, j]$ 到 $a[i + 1, j + 1]$;

3. 删除：从 $a[i, j]$ 到 $a[i + 1, j]$;

4. 插入：从 $a[i, j]$ 到 $a[i, j + 1]$;

最终 $a[x.length + 1, y.length + 1]$ 即为完成从 x 到 y 的操作;

Algorithm 4 algorithm4

function ALGORITHM4(X, Y)
    let $n = X.length$, $m = Y.length$
    let $a[1 \dots n+1, 1 \dots y+1]$, $b[1 \dots n+1, 1 \dots y+1]$ be new tables
    for $i = 1 \dots n+1$ **do**
        $a[i, 1] = c_insert \times i$
    for $j = 1 \dots m+1$ **do**
        $a[1, j] = c_insert \times j$
    for $i = 2 \dots n+1$ **do**
        for $j = 2 \dots m+1$ **do**
            if $X[i] == Y[j]$ **then**
                $d_1 = a[i-1, j-1] + c_{copy}$ // copy
            if $X[i]! = Y[j]$ **then**
                $d_1 = a[i-1, j-1] + c_{replace}$ // replace
            $d_2 = a[i-1, j] + c_{delete}$ // delete
            $d_3 = a[i, j-1] + c_{insert}$ // insert
            $a[i, j] = \min\{d_1, d_2, d_3\}$
            depending on the choice of $a[i, j]$, let $b[i, j]$ to be $\{\swarrow, \dot{\swarrow}, \downarrow, \rightarrow\}$
    output $a[n+1, m+1]$ and a sequence from $b[n+1, j+1]$ with a stack