算法基础HW4

PB19071535徐昊天

问题1

答:

对四种操作分别作分析:

1. 复制：对于下标i,j，皆后移一位，且直接对z数组赋x数组的元素，由于要求结束时有$z[j]=y[j]$，故当$x[i]=y[j]$时才可使用。
2. 替换：对于下标i,j，皆后移一位。
3. 删除：对于下标i，后移一位；下标j不变化。
4. 插入：对于下标j，后移一位；下标i不变化。

令$c(i,j)$为处理x,y字符串的下标分别为i,j时变换操作的总代价。

则存在关系式:

$$c(i,j)=min\begin{cases}c(i-1,j)+cost(delete)\\c(i,j-1)+cost(insert)\\c(i-1,j-1)+cost(replace)\\\begin{cases}c(i-1,j-1)+cost(copy) & x[i]=y[j]\\\infty & x[i]\neq y[j]\end{cases}\end{cases}$$

设计算法伪代码如下:

```
EDIT(x,y)
let K[0..x.length][0..y.length],T[0..x.length][0..y.length] be a new table
    for i=0 to x,length
        K[i,0]=i*cost(delete)
    for j=1 to y.length
        K[0,j]=j*cost(insert)
    for i=1 to x.length
        for j=1 to y.length
            if x[i]=y[j]
                K[i,j]=min{K[i-1,j-1]+cost(copy),K[i-1,j-1]+cost(replace),K[i-1,j]+cost(delete),K[i,j-1]+cost(insert)}
                T[i,j]=edit of K[i,j]       //若K[i,j]=K[i-1,j-1]+cost(copy),则T[i,j]='copy'；其他同理
            else
                K[i,j]=min{K[i-1,j-1]+cost(replace),K[i-1,j]+cost(delete),K[i,j-1]+cost(insert)}
                T[i,j]=edit of K[i,j]       //若K[i,j]=K[i-1,j-1]+cost(replace),则T[i,j]='replace'；其他同理
    return K,T
```

由以上**EDIT**算法可知，$K[x.length][y.length]$即为所求编辑距离。T矩阵用于储存最优变换操作选择，矩阵中的元素储存了到达i,j该值时经历的上一个操作。

时间复杂度分析：

第3行for循环时间复杂度为$\Theta(x.length)$，第5行for循环时间复杂度为$\Theta(y.length)$，第7、8行两层for循环时间复杂度为$\Theta(x.length \times y.length)$，故以上**EDIT**算法时间复杂度为$\Theta(x.length \times y.length)$。

正确性分析：

用数学归纳法证明：$K[i'][j']$储存了维护的下标$i=i',j=j'$时$x[1..i']$到$y[1..j']$的编辑距离。

证明如下：

①$i=j=0$时

显然有$K[0][0]=0$，此时x,y字符串为空，显然正确。

②$i \neq 0$且$j=0$时

根据算法有：$K[i,0]=i*cost(delete)$。由于此时y字符串为空，故复制、替换、插入操作均不可用，只可执行i次删除操作，故算法正确。

③$i=0$且$j \neq 0$时

根据算法有：$K[0,j]=j*cost(insert)$。由于此时x字符串为空，故复制、替换、删除操作均不可用，只可执行j次插入操作，故算法正确。

④$i>0,j>0$时

令$K[i-1][j-1],K[i-1][j],K[i][j-1]$=y[j]

$x[i] \neq y[j]$时，由于要求结束时有$z[j]=y[j]$，故不可执行复制操作。只可执行替换、删除、插入操作。若执行替换操作，则$cost=K[i-1,j-1]+cost(replace)$;若执行删除操作，则$cost=K[i-1,j]+cost(delete)$;若执行插入操作，则
$cost=K[i,j-1]+cost(insert)$。由于$K[i-1][j-1],K[i-1][j],K[i][j-1]$分别储存对应下标的编辑距离，故以上三个等式的结果的最小值即为$K[i][j]$对应下标的编辑距离。

$x[i]=y[j]$时，由于要求结束时有$z[j]=y[j]$，故可以执行复制操作。若执行复制操作，则
$cost=K[i-1,j-1]+cost(copy)$;若执行替换操作，则
$cost=K[i-1,j-1]+cost(replace)$;若执行删除操作，则
$cost=K[i-1,j]+cost(delete)$;若执行插入操作，则

$cost=K[i,j-1]+cost(insert)$。由于$K[i-1][j-1],K[i-1][j],K[i][j-1]$分别储存对应下标的编辑距离，故以上四个等式的结果的最小值即为$K[i][j]$对应下标的编辑距离。

综上所述，可证该算法正确。

问题2

答：

采样算法导论中介绍的左孩子右兄弟表示法描述树，并在树节点结构体相对于算法导论中内容添加了value(权重),id(标识符)两项。设计算法伪代码如下：

```
VALUE_1(root,A,B)      // include root
    value=root.value+VALUE_2(root.left-child,A,B)+VALUE_2(root.right-sibling,A,B)
    A[root.id]=value
    return value

VALUE_2(root,A,B)      // exclude root
    value=max(VALUE_1(root.left-child,s),VALUE_2(root.left-child,s))+max(VALUE_1(root.right-sibling,s),VALUE_2(root.right-sibling,s))
    B[root.id]=value
    return value

PRINT(T.root,s,flag)
    if(root!=NIL)
        if(A[root.id]>=B[root.id] && flag=1)
            s=s∪root
            PRINT(root.left-child,s,0)
            PRINT(root.right-sibling,s,0)
        else
            PRINT(root.left-child,s,1)
            PRINT(root.right-sibling,s,1)

MAX_VALUE(T.root)
    let s a new set
    let A[1..n],B[1..n] be new array
    value=max(VALUE_1(T.root,A,B),VALUE_2(T.root,A,B))
    root=T.root
    PRINT(root,s,1)
    return value,s
```

VALUE_1函数返回以root为根的子树的最优顶点独立子集(包括root)，于是与root相邻的左孩子和右兄弟皆不可加入该子集中；**VALUE_2**函数返回root为根的子树的最优顶点独立子集(不包括root)，于是可以将与root相邻的左孩子和右兄弟考虑加入子集，并分别对这两个结点针对以上两种情况做选择。

MAX_VALUE函数中建立数组A、B分别用于储存树中结点的**VALUE_1**函数、**VALUE_2**函数返回值，数组下标由树节点id确定，树节点id唯一。

MAX_VALUE函数中建立新的集合s，并作为参数传入**PRINT**函数中，利用递归的方式将最优顶点独立子集存入s中。其中flag用于判断是否有相邻结点已经放入s集合中，若有则设置为0，该结点即不会加入s中；否则设置为1。

MAX_VALUE函数最后将最优顶点独立子集及其权重返回。

时间复杂度分析：

首先分析代码24行递归调用**VALUE_1**函数及**VALUE_2**函数，该操作对每个结点皆执行常数次，故时间为$\Theta(n)(n\text{为结点数})$。

分析**PRINT**函数，该函数对树中每个结点皆遍历一次，故时间为$\Theta(n)$。

综上，该算法时间复杂度为$\Theta(n)$。

正确性分析：

用数学归纳法证明：**VALUE_1**函数及**VALUE_2**函数分别得到包含与不包含root结点时子树的最优顶点独立子集的权重。

①root的左孩子和右兄弟皆为空时

此时显然**VALUE_2**函数返回值为0，**VALUE_1**函数返回值为该结点的权重，显然正确。

②root的左孩子为空，右兄弟不为空时

显然左孩子无需考虑。对于**VALUE_1**函数，由于root已经加入集合中，则右兄弟不可加入，故只可调用**VALUE_2**函数；对于**VALUE_2**函数，由于root未加入集合中，故是否将右孩子加入集合都可行，故要在**VALUE_1**函数与**VALUE_2**函数中选择最大值。故显然符合假设情况。

③root的左孩子不为空，右兄弟为空时

显然右兄弟无需考虑。对于**VALUE_1**函数，由于root已经加入集合中，则左孩子不可加入，故只可调用**VALUE_2**函数；对于**VALUE_2**函数，由于root未加入集合中，故是否将左孩子加入集合都可行，故要在**VALUE_1**函数与**VALUE_2**函数中选择最大值。故显然符合假设情况。

④root的左孩子和右兄弟皆不为空时

对于**VALUE_1**函数，由于root已经加入集合中，则左孩子和右兄弟不可加入，故只可调用**VALUE_2**函数并将两者相加(由于两者不相邻，故可以直接相加)；对于**VALUE_2**函数，由于root未加入集合中，故是否将左孩子和右兄弟加入集合都可行，故要在**VALUE_1**函数与**VALUE_2**函数中选择最大值并将两者相加。故显然符合假设情况。

综上所述，可证算法正确。

问题3

答：

将问题转化为判断问题，即原问题是否存在不大于某个距离 r 的解。用贪心算法解决该判定问题，然后使用二分搜索找到最优距离。树节点中与父结点距离用变量 `distance` 储存。

设计伪代码如下：

```
Longest_Path(T,b,c)
  length=max(Longest_Path(T.root.left-child)+T.root.left-child.distance,Longest_Path(T.root.right-sibling)+T.root.right-sibling.distance-T.root.distance)
  let b[T.root] be the better successor of T.root,c[T.root] be the distance with b[T.root]
  return length

A(T,k,r,s)
  if(T==NIL)
    return 1
  else if (k==0&&T!=NIL)
    return 0
  else
    let b[1..n],c[1..n] be new arrays
    max=Longest_Path(T,b,c)
    go up from the deepest node on the longest path, let N be the node that the last one whose distance from the deepest node is less than r
    s=s∪N
    return A(T-node,k-1,r,s)    // node包含N及距离N小于r的所有节点

BINARY_SEARCH(T,k)
    let max be the largest distance between two nodes of tree,s be a new set
    low=0
    high=max
    while low<=high
      mid=(low+high)/2
      if A(T,k,mid,s)==1
        high=mid-1
      else
        low=mid+1
    return low
```

如上，算法原理如下：

对于**BINARY_SEARCH**函数，首先令max储存树中距离最大的两个节点的距离，作为r的上界，而后使用二分法在0~max的范围内寻找最佳的r值：若r值满足条件，则向较小的方向折半，寻找最小r值；若r值不满足条件，则向较大的方向折半，寻找满足条件的r值。直至二分结束，即获得了最佳r值。

对于**Longest_Path**函数，即通过递归的方式获得树的根结点能到达的最长路径并用数组储存最长路径上的结点集合以及相邻结点的距离。

对于A函数，将其作为一个判定问题，判断是否存在k个结点使得这些结点在r值的范围内覆盖树中所有结点。设计贪心算法，首先调用**Longest_Path**函数获得根结点能到达的最长路径，而后从结点集合中最底层的结点向上遍历，寻找最后一个距离最底层结点小于r的结点并将其作为中心，并将中心存入s集合中。而后递归调用A函数：$A(T-node,k-1,r,s)$。递归调用时中心个数减一，并将之前获得的一个中心及树中所有和该中心距离小于r的结点从树中删去(并非删除结点，而是在后续执行递归函数时不再考虑这些结点)。若最后树中所有节点皆被覆盖则返回1；若k=0，即中心设置完毕后树中依然存在未被覆盖的结点则返回0。

时间复杂度分析：

首先分析A函数，由于A函数中调用Longest_Path函数需将树中的每一个结点遍历一次，故时间复杂度为$\Theta(n)(n为树结点个数)$；对最长路径上的结点集合的寻找操作不会超过n次，故时间复杂度为$O(n)$；由于该函数需递归调用最多k次，故该函数时间复杂度为$O(kn)=O(n)$。

分析**BINARY_SEARCH**函数，由于要在max的范围内进行折半操作，故需进行$log(max)$次折半，每次折半执行一次A函数。

综上可知，算法时间复杂度为$O(nlog(max))$。

正确性分析：

用数学归纳法证明：在**BINARY_SEARCH**函数二分循环结束后，low是[0..max]中满足$A(T,k,r,s)=1$的最小值,$high$是[0..max]中满足$A(T,k,r,s)=0$的最大值。

证明如下：

①$low=0,high=max$时

显然存在$A(T,k,0,s)=0,A(T,k,max,s)=1$。

②$low<=high$时

（情况一）

若上一次循环结束后满足$A(T,k,low',s)=0,A(T,k,high',s)=1$。

若$A(T,k,mid',s)=1$，则有$high''=mid'-1$，则此次循环结束后可能处于情况一或情况三。

若$A(T,k,mid',s)=0$，则有$low''=mid'+1$，则此次循环结束后可能处于情况一或情况二。

在满足情况一的条件下，low小于[0..max]中满足$A(T,k,r,s)=1$的最小值,$high$大于[0..max]中满足$A(T,k,r,s)=0$的最大值。

（情况二）

若上一次循环结束后满足$A(T,k,low',s)=A(T,k,high',s)=1$。

则必有$A(T,k,mid',s)=1$，$high$会向左折半，继续满足情况二。

假设low'不是满足$A(T,k,r,s)=1$的最小值，即满足$A(T,k,low'-1,s)=1$，则之前的循环中应向左折半，即执行$high=mid-1$，而并非对low作修改，则显然假设不正确。故应当满足low'是满足$A(T,k,r,s)=1$的最小值。

（情况三）

若上一次循环结束后满足$A(T,k,low',s)=A(T,k,high',s)=0$。

则必有$A(T,k,mid',s)=0$，low会向右折半，继续满足情况三。

假设$high'$不是满足$A(T,k,r,s)=0$的最大值，即满足$A(T,k,high'-1,s)=0$，则之前的循环中应向右折半，即执行$low=mid+1$，而并非对$high$作修改，则显然假设不正确。故应当满足$high'$是满足$A(T,k,r,s)=0$的最大值。

③$low>high$时

根据②可知，循环结束前将在以上三种情况中切换，易得循环结束后必满足$A(T,k,low,s)=1,A(T,k,high,s)=0$。由于显然存在$low=high+1$，故满足$low$是[0..max]中满足$A(T,k,r,s)=1$的最小值,$high$是[0..max]中满足$A(T,k,r,s)=0$的最大值。故函数返回$low$即为所求的最小化的通信价值最大值。

综上所述，可证该算法正确。

问题4

答：

通过遍历序列并合理调用二分查找，设计算法伪代码如下：

```
FIND_LIS(a)
    let b[1..n],c[1..n] be new arrays
    /*
    c[i]储存数组a[1..i]中以a[i]为最后一个元素的最长递增子序列长度
    b[i]储存数组a数组中长度为i的最长递增子序列的最后一个元素的最小值。
    */
    length=1
    b[1]=a[1]
    c[1]=1
    for i=2 to n
        if a[i]<b[1]
            b[1]=a[i]
            c[i]=1
        else if(a[i]>b[length])
```

```
15            length++
16            b[length]=a[i]
17            c[i]=length
18        else
19            j=BINARY_SEARCH(b,length,a[i])  // b[j-1]<a[i]≤b[j]
20            b[j]=a[i]
21            c[i]=j
22     let s[1..length] be a new array
23     j=length
24     for i=n to 1   // 写入s数组
25         if(c[i]=j)
26             s[j]=a[i]
27             j--
28     for i=1 to length
29         print(s[i])
30     return length
31
32 BINARY_SEARCH(b,len,a)
33     low=1
34     high=len
35     while low≤high do
36        mid=(low+high)/2    //折半查找
37        if A[mid]==key then
38            return mid
39        else if A[mid]>key then     //向左折半
40            high=mid-1
41        else       //向右折半
42            low=mid+1
43     return low
```

由上，算法原理如下：

FIND_LIS算法中，c数组中$c[i]$储存数组$a[1..i]$中以$a[i]$为最后一个元素的最长递增子序列长度，b数组中$b[i]$储存a数组中长度为i的最长递增子序列的最后一个元素的最小值。

BINARY_SEARCH算法用于通过二分法查找b数组前len个元素中满足$b[j-1] < a[i] \leq b[j]$的j值。

FIND_LIS算法第10行循环结束后，通过c数组中的元素将a数组中的最长递增子序列存入s数组中并打印出来，最后返回最长递增子序列长度。

时间复杂度分析：

首先对**BINARY_SEARCH**算法作分析，由于算法采用折半查找法，首次查找时数组长度为 `len`，故k次查找后数组长度为$\frac{len}{2^k}$。

有：$\frac{len}{2^k} \geq 1 => k \leq log(len) \leq logn$。

即可证得该算法的时间复杂度为$O(logn)$。

对**FIND_LIS**算法作分析，由于第10行的for循环执行n次，故该for循环时间复杂度为$O(nlogn)$；第24、28行的for循环分别执行n、$length$次，时间复杂度分别为$O(n)$、$O(length)$，故**FIND_LIS**算法时间复杂度为$O(nlogn)$。

正确性分析：

循环不变式：

（性质一）**FIND_LIS**算法在10行的循环结构每次迭代之后，满足$c[i]$储存数组$a[1..i]$中以$a[i]$为最后一个元素的最长递增子序列长度；

（性质二）对$0 < i' \leq i$有$b[i']$储存数组$a[1..i]$中长度为i'的最长递增子序列的最后一个元素的最小值；

（性质三）**length**储存数组$a[1..i]$中最长递增子序列的长度。

初始化：

循环开始之前，**length=1**是$a[1]$中最长递增子序列的长度；$b[1] = a[1]$，即长度为1的最长递增子序列的最后一个元素最小值；$c[1] = 1$，即以$a[1]$为最后一个元素的最长递增子序列的长度。故满足循环不变式。

保持：

某次循环迭代之后，满足循环不变式，令次轮循环时$i = i' - 1$。

则下一轮循环时$i = i'$。

①若$a[i] < b[1]$

显然$a[i]$是$a[1..i]$中的最小元素，故$a[i]$为最后一个元素的最长递增子序列长度只能为1，故$c[i] = 1$以满足性质一；

由于该元素小于$b[1]$，故应该将$b[1]$更新为$a[i]$，从而保持性质二；

由于$a[i]$是$a[1..i]$中的最小元素，故不会对已知的最长递增子序列产生影响，**length**不会发生改变。由于上一轮循环结束后满足性质三，故该轮循环结束后依旧满足性质三。

②若$a[i] > b[length]$

由于$b[length]$储存数组$a[1..i]$中长度为$length$的最长递增子序列的最后一个元素的最小值，故$a[i]$可作为最后一个元素加入已知的最长递增子序列，且长度为**length+1**，故$c[i] = length + 1$，满足性质一；

由于在这一轮循环中第一次找到长度为$length + 1$的最长递增子序列，故最后一个元素的最小值即为$a[i]$，满足性质二；

显然$a[i]$作为最后一个元素加入了已知的最长递增子序列从而得到了新的序列，且长度为**length+1**，满足性质三。

③若$b[1] \leq a[i] \leq b[length]$

由于得到j满足$b[j-1] < a[i] \leq b[j]$，即$a[i]$大于数组$a[1..i]$中长度为$j-1$的最长递增子序列的最后一个元素的最小值，且小于数组$a[1..i]$中长度为j的最长递增子序列的最后一个元素的最小值。故长度为j的最长递增子序列的最后一个元素的最小值可更新为$a[i]$，满足性质二;

由于$a[i]$为最后一个元素的最长递增子序列长度为j，故执行$c[i]=j$，满足性质一;

由于未找到更长的最长递增子序列，故**length**不变。由于上一轮循环结束后满足性质三，故该轮循环结束后依旧满足性质三。

终止:

终止时$i=n$，易得循环结束后满足$c[n]$储存数组$a[1..n]$中以$a[n]$为最后一个元素的最长递增子序列长度，即满足性质一;

循环结束后，对$0 < i' \leq n$有$b[i']$储存数组$a[1..n]$中长度为i'的最长递增子序列的最后一个元素的最小值，即满足性质二;

循环结束时，**length**储存数组$a[1..n]$中最长递增子序列的长度，即题中所求。

综上所述，该算法正确。

问题5

答:

(a)设计动态规划算法:

假设我们已知解决方案中存在一个重量为 w 的特定项目。然后我们必须解决具有最大权重 $W-w$ 的 $n-1$ 个项目的子问题。因此需解决所有物品和可能的重量小于 W 的问题，自底向上求解。

令$A(j,Y)$：在总重量不超过Y的前提下，前j种物品的总价格所能达到的最大值。

故$A(j,Y)$的递推关系式为:

(1)$A(0,Y)=0$

(2)若$w_j > Y$，则$A(j,Y)=A(j-1,Y)$

(3)若$w_j \leq Y$，则$A(j,Y)=max\{A(j-1,Y), v_j + A(j-1, Y-w_j)\}$

通过计算$A(n,W)$即可得到最大总价值。

伪代码如下：

```
1  KNAPSACK(n,W,w,v)   // w,v为物品的重量和价值
2     let K[0..n][0..W] ,T[0..n][0..W] be new table
3     for i=1 to n
4        K[i,0]=0
5     for j=0 to W
6        K[0,j]=0
7     for i=1 to n
8        for j=1 to W
9           if j<w[i]
10             K[i,j]=K[i-1,j]
11             T[i][j]=0
12          else
13             K[i,j]=max{K[i-1,j],K[i-1,j-w[i]]+v[i]}
14             if (K[i-1,j]>K[i-1,j-w[i]]+v[i])
15                T[i][j]=0
16             else
17                T[i][j]=1
18 return K,T
```

由以上伪代码可知，$K[n][W]$即为所求最优价值；T矩阵用于储存最优选择，矩阵中为1的元素构成了最优选择。

时间复杂度分析：

第3行for循环时间复杂度为$\Theta(n)$，第5行for循环时间复杂度为$\Theta(W)$，第7、8行两层for循环时间复杂度为$\Theta(nW)$，故以上算法时间复杂度为$\Theta(nW)$。

正确性分析：

利用数学归纳法证明：算法中第7、8行的二维循环中每执行结束一次，$K[i][j]$中都储存了在总重量不超过j的前提下，前i种物品的总价格所能达到的最大值。

①$i=0$或$j=0$时

由于物品数或重量上限为0，故最优值一定为0，显然正确。

②i,j皆不为0时

假设此轮循环之前的循环结束后皆满足证明条件。

则分情况讨论：

I.若第i个物品重量大于重量上限，则显然不可将该物品加入背包中，物品数应当减一，即$K[i][j]=K[i-1][j]$。由假设可知$K[i-1][j]$符合证明条件，则$K[i][j]$也符合证明条件。

II.若加入第i个物品后的价值小于加入之后对应重量上限的最优价值，则也不可将该物品加入背包中，物品数应当减一，即$K[i][j] = K[i-1][j]$。由假设可知$K[i-1][j]$符合证明条件，则$K[i][j]$也符合证明条件。

III.若加入第i个物品后的价值大于加入之后对应重量上限的最优价值，则需将该物品加入背包中并更新$K[i][j]$，更新后的$K[i][j]$即满足证明条件。

③$i = n, j = W$时

根据以上的归纳结论，可知$K[n][W]$即储存了在总重量不超过W的前提下，前n种物品的总价格所能达到的最大值。即为题中所求的最优价值。

综上所述，以上算法正确。

(b)

由于算法时间复杂度为$\Theta(nW)$，即为$\Theta(n^1 \times W^1)$，故时间复杂度是多项式级别的。