

# 算法设计与分析作业六

作者：吴润泽      学号：181860109

Email: [181860109@smail.nju.edu.cn](mailto:181860109@smail.nju.edu.cn)

2020 年 5 月 22 日

## 目录

<b>Chapter 13</b>	<b>2</b>
problem 13.1 . . . . .	2
problem 13.2 . . . . .	3
problem 13.7 . . . . .	4
<b>Chapter 14</b>	<b>4</b>
problem 14.2 . . . . .	4
problem 14.3 . . . . .	5
problem 14.6 . . . . .	6
problem 14.7 . . . . .	8
problem 14.11 . . . . .	8
problem 14.13 . . . . .	10
problem 14.14 . . . . .	11

## Chapter 13

## problem 13.1

1)

```

1 FloydNext(G):
2   D(0):=W /*初始情况下，两点间的最短路径长度就是它们的边权*/
3   for(i:=1 to n):
4     for(j:=1 to n):
5       if map[i][j]!=Inf: /*如果ij间有边连接，则i的后继即为j*/
6         GO[i][j]=j
7   for(k:=1 to n):
8     for(i:=1 to n):
9       for(j:=1 to n):
10        if D[i][j]>D[i][k]+D[k][j]:
11          GO[i][j]=k /*如果i到j通过k的路径更短，则更新路由表
12          */
          D[i][j]=D[i][k]+D[k][j]

```

FloydNext.func

2) 算法更新路径  $i \rightarrow k \rightarrow j$ ，所选择的  $j$  的前驱与从选择的节点  $k$  到节点  $j$  的一条中间节点取自集合  $\{1, 2, \dots, k-1\}$  的最短路径上的前驱是一样的。

```

1 FloydPrev(G):
2   D(0):=W
3   for(i:=1 to n):
4     for(j:=1 to n):
5       if map[i][j]!=Inf: /*如果ij间有边连接，则j的前驱即为i*/
6         GO[i][j]:=i
7   for(k:=1 to n):
8     for(i:=1 to n):
9       for(j:=1 to n):
10        if D[i][j]>D[i][k]+D[k][j]:
11          GO[i][j]:=GO[k][j] /*如果i到j通过k的路径更短，则前
12          缀为Back[k][j]*/
          D[i][j]:=D[i][k]+D[k][j]

```

FloydPrev.func

## problem 13.2

1) 仿照 Dijkstra 算法，记录每条路径的最小值即可。

```

1 DijkstraCap(G):
2   initialize the priority queue Fringe as empty
3   insert some node v to Fringe
4   cap:=[] /*cap 原来存放s到达其他点的吞吐量*/
5   while Fringe!=empty:
6       v:=Fringe.EXTRACT-MIN()
7       cap[v]:=v.priority /*存放到达v的结果*/
8       UPDATE-FRINGE(v,Fringe)
9   return cap
10 UPDATE-FRINGE(v,Fringe):
11   for neighbor w of v:
12       w.priority:=min(vw.weight,v.priority)
13       /*取路径的最小值，即当前路径已知吞吐量*/
14       if w is UNSEEN: /*如果是第一次遍历到*/
15           Fringe.INSERT(w,w.priority) /*将w加入队列*/
16       else:
17           Fringe.decrease(w,w.priority)
18           /*将w的权值更新为w.priority*/

```

DijkstraCap.func

2) 仿照 Floyd 算法，对路径  $i \rightarrow k \rightarrow j$  从  $ij, ik, kj$  中选取最小权边即可。

```

1 FloydCap(G):
2 D(0):=W /*开始的吞吐量记为相连边的边权值*/
3 for(k:=1 to n):
4     for(i:=1 to n):
5         for(j:=1 to n):
6             D[i][j]:=min(D[i][j],D[i][k],D[k][j]) /*选取3者的最小值，
              作为最大吞吐量*/

```

FloydCap.func

## problem 13.7

计算出所有点到达  $v_0$  的最短路径和  $v_0$  到达其他顶点的最短路径。则  $uv$  间的最短路径即为  $u$  到  $v_0$  的最短路径加上  $v$  到  $v_0$  的最短路径。

```

1 ShortByV(G,v):
2   D:=W /*初始化所有最短路径为起始结点权值*/
3   GT=reverse(G) /*得到G的转置图*/
4   D[][v]:=dijkstra(GT,v) /*对GT使用dijkstra算法*/
5   /*计算所有节点到达v的最短路径，结果存入D[][v]中*/
6   dijkstra(G,v) /*对G使用dijkstra算法*/
7   /*计算v到达所有节点的最短路径，结果存入D[v][]中*/
8   for(i:=1 to n):
9     for(j:=1 to n):
10      D[i][j]:=D[i][v]+D[v][j]
11      /*最短路径为i到v加v到j的最短路径*/

```

ShortByV.func

## Chapter 14

## problem 14.2

解 取子问题为  $dp[i][j]$ ，含义为使用前  $i$  个数能否满足元素和为  $j$ 。对于  $dp[i][j]$  要么使用了第  $i$  个元素，要么使用了前  $i-1$  个元素。故其递推式为：

$$dp[i][j] = (dp[i-1][j]) || (dp[i][j-A[i]]).$$

基于此得到整数子集和算法，时间复杂度  $O(nS)$ ，空间复杂度  $O(nS)$ ：

```

1 SubSeqSum(A,S):
2   dp:=[n+1][S+1] /*dp数组大小为(n+1)(S+1)的二维数组*/
3   for i:=1 to n do: /*初始化布尔dp数组*/
4     dp[i][0]:=dp[i][A[i]]:=true
5   for i:=2 to n do:
6     for j:=1 to S do:
7       if A[i]>j: /*如果A[i]已经大于j，必不能包含A[i]*/
8         dp[i][j]:=dp[i-1][j]
9       else:
10        dp[i][j]:=(dp[i-1][j]) || (dp[i][j-A[i]])
11   return dp[n][s] /*返回使用n个数能否达到s即可*/

```

SubSeqSum.func

## problem 14.3

解 取子问题  $dp[i]$  为正整数  $i$  变为 1 需要的最少操作数。根据上述规则，可根据子问题  $i$  与 2, 3 的倍数关系，分成三种情况列出递推式：

$$\begin{cases} i = 3k \&\& i = 2k, & dp[i] = \min(d[i-1], d[i/2], d[i/3]) + 1, \\ i = 2k \&\& i! = 3k, & dp[i] = \min(d[i-1], d[i/2]) + 1, \\ i = 3k \&\& i! = 2k, & dp[i] = \min(d[i-1], d[i/3]) + 1, \\ i! = 3k \&\& i! = 2k, & dp[i] = d[i-1] + 1. \end{cases}$$

基于此得到算法伪代码，时间复杂度  $O(n)$ ，空间复杂度  $O(n)$ ：

```

1 MiniOp(n):
2   dp:= [n+1] /*dp数组大小为n+1的一维数组*/
3   dp[1]:=dp[2]:=dp[3]:=1 /*1,2,3均只需要1次操作*/
4   for i:=4 to n do:
5     if i%6=0:
6       dp[i]:=min(d[i-1],d[i/2],d[i/3])+1
7     else if i%3=0:
8       dp[i]:=min(d[i-1],d[i/3])+1
9     else if i%2=0:
10      dp[i]:=min(d[i-1],d[i/2])+1
11    else:
12      dp[i]:=d[i-1]+1
13  return dp[n] /*返回子问题规模为n的最少操作数即可*/

```

MiniOp.func

## problem 14.6

1) 取子问题  $dp[i][j]$  记为  $X$  的前  $i$  个字符与  $Y$  的前  $j$  个字符的 LCS 长度。基于  $X[i]$  与  $Y[j]$  是否相等得到递推方程:

$$\begin{cases} X[i] == Y[j], & dp[i][j] = dp[i-1][j-1] + 1, \\ X[i] \neq Y[j], & dp[i][j] = \max(dp[i-1][j], dp[i][j-1]). \end{cases}$$

基于此得到算法伪代码, 时间复杂度  $O(mn)$ , 空间复杂度  $O(mn)$ :

```

1 LCS(X,Y):
2   dp:=[m+1][n+1] /*dp数组大小为(m+1)(n+1)的二维数组*/
3   for i:=1 to m do:
4     dp[i][0]:=0 /*X子串与空字符串的LCS为0*/
5   for j:=1 to n do:
6     dp[0][j]:=0 /*Y子串与空字符串的LCS为0*/
7   for i:=1 to m do:
8     for j:=1 to n do:
9       if X[i]=Y[j]:
10        dp[i][j]:=dp[i-1][j-1]+1
11      else:
12        dp[i][j]:=max(dp[i-1][j], dp[i][j-1])
13  return dp[m][n] /*返回子问题规模为X和Y的LCS即可*/

```

LCS.func

2)  $X$  中的字符可以重复出现, 当  $X[i] = Y[j]$  时, 对于子问题  $dp[i][j]$ , 其等价于  $X$  的前  $i$  个字符与  $Y$  前  $j-1$  个字符的 LCS 加 1。递推方程修改为:

$$\begin{cases} X[i] == Y[j], & dp[i][j] = dp[i][j-1] + 1, \\ X[i] \neq Y[j], & dp[i][j] = \max(dp[i-1][j], dp[i][j-1]). \end{cases}$$

```

1 ReLCS(X,Y):
2   dp:=[m+1][n+1] /*dp数组大小为(m+1)(n+1)的二维数组*/
3   for i:=1 to m do:
4     dp[i][0]:=0 /*X子串与空字符串的LCS为0*/
5   for j:=1 to n do:
6     dp[0][j]:=0 /*Y子串与空字符串的LCS为0*/
7   for i:=1 to m do:
8     for j:=1 to n do:
9       if X[i]=Y[j]:

```

```

10         dp[i][j]:=dp[i][j-1]+1 /*递推式进行修改*/
11     else:
12         dp[i][j]:=max(dp[i-1][j],dp[i][j-1])
13     return dp[m][n] /*返回子问题规模为X和Y的LCS即可*/

```

ReLCS.func

时间复杂度与空间复杂度不变。

3) 在循环中记录重复次数, 当  $X[i] = Y[j]$  且  $X[i]$  重复次数超过  $k$  后,  $dp[i][j]$  的 LCS 等价于  $X$  的前  $i-1$  个字符和  $Y$  前  $j-1$  个字符的 LCS 加 1。开辟三维数组记录,  $dp[i][j][1]$  记录原始的 LCS 结果,  $dp[i][j][2]$  记录  $X[i]$  重复使用的次数。

```

1 LimReLCS(X,Y,k):
2     dp:=[m+1][n+1][2] /*dp数组大小为2(m+1)(n+1)的三维数组*/
3     for i:=1 to m do:
4         dp[i][0][1]:=0 /*X子串与空字符串的LCS为0*/
5     for j:=1 to n do:
6         dp[0][j][1]:=0 /*Y子串与空字符串的LCS为0*/
7     dp[0][0][2]:=0 /*初始化重复次数均为0*/
8     for i:=1 to m do:
9         for j:=1 to n do:
10            if X[i]=Y[j]:
11                if dp[i][j-1][2]>=k:
12                    dp[i][j][2]:=dp[i-1][j-1][2]
13                    dp[i][j][1]:=dp[i-1][j-1][1]+1
14                else:
15                    if dp[i][j-1][1]>dp[i-1][j-1][1] /*即需要使用X[i]*/
16                        dp[i][j][1]:=dp[i][j-1][1]+1
17                        dp[i][j][2]:=dp[i][j-1][2]+1 /*重复次数加1*/
18                    else:
19                        dp[i][j][1]:=dp[i-1][j-1][1]+1
20                        dp[i][j][2]:=0 /*没有使用X[i]*/
21            else:
22                dp[i][j]:=max(dp[i-1][j],dp[i][j-1]) /*重复次数与LCS结
                果均被复制*/
23     return dp[m][n][1] /*返回子问题规模为X和Y的LCS即可*/

```

LimReLCS.func

## problem 14.7

解 取子问题  $dp[i][j]$  记为以  $T[i]$  为结尾的前向串, 以  $T[j]$  为开始的向后串的最大连续子串长度。为了满足两子串不重叠, 易得  $i < j$ 。根据  $T[i]$  与  $T[j]$  得递推方程:

$$\begin{cases} T[i] == T[j] \&\& i < j, & dp[i][j] = dp[i-1][j+1] + 1, \\ T[i] != T[j] \&\& i < j, & dp[i][j] = 0. \end{cases}$$

基于此得到算法伪代码, 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n^2)$ :

```

1 FroLas(T):
2   dp:=[n+1][n+1] /*dp数组大小为(m+1)(n+1)的二维数组*/
3   res:=-Inf /*初始化为最小值, 结果保存最大的连续长度*/
4   for i:=1 to n do:
5     dp[i][0]:=dp[0][i]:=0 /*子串与空字符串的重叠长度为0*/
6   for i:=1 to n do: /*前向串从头向后扩充*/
7     for j:=n to i+1 do: /*前向串从尾部向前扩充*/
8       if T[i]=T[j]:
9         dp[i][j]:=dp[i-1][j+1]+1
10      else:
11        dp[i][j]:=0
12      res:=max(res, dp[i][j]) /*更新结果*/
13   return res

```

FroLas.func

## problem 14.11

1) 与 problem 14.7类似, 取子问题  $dp[i][j]$  为子串  $T[i \cdots j]$  的最长回文子序列。同样根据  $T[i]$  与  $T[j]$  得到递推方程:

$$\begin{cases} T[i] == T[j] \&\& i < j, & dp[i][j] = dp[i+1][j-1] + 2, \\ T[i] != T[j] \&\& i < j, & dp[i][j] = \max(dp[i][j-1], dp[i+1][j]). \end{cases}$$

基于此得到算法伪代码, 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n^2)$ :

```

1 PalindromSeq(T):
2   dp:=[n+1][n+1] /*dp数组大小为(n+1)(n+1)的二维数组*/
3   for i:=1 to n do:
4     dp[i][i]:=1 /*单个字符的回文串长度为1*/
5   for i:=1 to n do:

```



```

6      for j:=i to n-i-1 do: /*枚举i-j的子串*/
7          if T[i]=T[j]:
8              dp[i][j]:=dp[i+1][j-1]+2
9          else:
10             dp[i][j]:=max(dp[i][j-1],dp[i+1][j])
11 return dp[0][n-1] /*返回原字符串的最大回文序列长度*/

```

PalindromSeq.func

2) 采用类似的算法,将字符串中的所有回文字串划分。取子问题  $IsPalind[i][j]$  子串从  $i$  到  $j$  是否为回文子串,递推方程为:

$$IsPalind[i][j] = (IsPalind[i+1][j-1]) \& (T[i] == T[j]).$$

取子问题  $dp[i]$  为划分前  $i$  个字符串需要的划分次数,如果  $j-i$  为回文子串,那么  $dp[i] \geq dp[j] + 1$ 。因此递推方程为:

$$dp[i] = \min(dp[i], (IsPalind[j][i]) \& (dp[j] + 1)), j = 1, 2, \dots, i-1.$$

基于此得到算法伪代码,时间复杂度  $O(n^2)$ , 空间复杂度  $O(n^2)$ :

```

1 FindPalind(T):
2     IsPalind[n+1][n+1] /*IsPalind数组大小为(n+1)(n+1)的二维数组*/
3     for i:=1 to n do:
4         IsPalind[i][i]:=true /*单个字符均为回文串*/
5     for i:=1 to n do:
6         for j:=1 to i do: /*枚举j-i的子串*/
7             IsPalind[i][j]:=(IsPalind[i+1][j-1])&(T[i]=T[j])
8
9 PalindSplit(T):
10    FindPalind(T) /*将字符串中所有的回文子串进行标记*/
11    dp:=[n+1] /*dp数组大小为(n+1)的一维数组*/
12    for i:=1 to n do:
13        dp[i]:=i-1 /*初始化划分i个字符需要i-1次*/
14    for i:=1 to n do:
15        for j:=1 to i do: /*枚举j-i的子串*/
16            if IsPalind[i][j]:
17                dp[i]:=min(dp[i],dp[j]+1)
18    return dp[n-1]+1 /*返回原字符串最小划分次数+1,即回文串数量*/

```

PalindSplit.func

## problem 14.13

1) 取子问题  $dp[i]$  记录是否可以表示金额  $i$ 。枚举使用每个硬币且不超过金额  $v$  的情况, 如果  $dp[j]$  为真, 则  $dp[x_k]$  也为真, 递推方程为:

$$dp[j + x_k] := dp[j], k = 1, 2, \dots, n.$$

```

1 RepCoin(X, v):
2   dp := [v+1] /*dp数组大小为(v+1)的一维数组*/
3   for k:=1 to n do: /*所有硬币整数倍的均可以拼凑*/
4     j:=0
5     while (j*X[j]<=v):
6       dp[j*X[j]] := true
7       j++
8   for k:=1 to n do:
9     for j:=0 to v do:
10      if dp[j]=true:
11        dp[j+X[k]] := true
12  return dp[v]
```

RepCoin.func

2) 与 problem 14.2 类似, 取子问题为  $dp[i][j]$ , 含义为使用前  $i$  个数能否满足金额和为  $j$ 。对于  $dp[i][j]$  要么使用了第  $i$  个元素, 要么使用了前  $i-1$  个元素。故其递推式为:

$$dp[i][j] = (dp[i-1][j]) || (dp[i][j - A[i]]).$$

3) 即找最少的硬币来兑换目标金额。取子问题  $dp[i]$  为组成金额  $i$  所需最少的硬币数量, 则对应的递推方程为:

$$dp[i] = \min(dp(i - x_k)) + 1, k = 0, 1, \dots, n.$$

```

1 MinRepCoin(X, v):
2   dp := [v+1] /*dp数组大小为(v+1)的一维数组*/
3   for i:=0 to v do: /*枚举0~v金额*/
4     for j:=1 to n do: /*枚举使用每一枚硬币*/
5       if X[j]<=i: /*硬币面值不超过所需金额*/
6         dp[i] := min(dp[i], dp[i-X[j]]+1)
7   return dp[v]
```

MinRepCoin.func

## problem 14.14

解 每个点设有两种状态：

$dp[i][0]$  表示点  $i$  属于点覆盖集，并且以点  $i$  为根的子树中所连接的边都被覆盖的情况下，点覆盖集中所包含最少点的个数。

$dp[i][1]$  表示点  $i$  不属于点覆盖集，并且以点  $i$  为根的子树中所连接的边都被覆盖的情况下，点覆盖集中所包含最少点的个数。

对于第一种状态，其最小个数等于所有子节点两种状态中最小值之和并加其本身。

对于第二种状态，因为点  $i$  所连接的边都应被覆盖，故点  $i$  的子节点都应被加入点覆盖集。故其最小个数等于所有子节点的第一种状态之和。

$$\begin{cases} dp[i][0] = \sum \min(dp[j][0], dp[j][1]), parent[j] = i, \\ dp[i][1] = \sum (dp[j][0]), parent[j] = i. \end{cases}$$

```

1 MiniVertexCover(u):
2     dp[u][0] := 1;
3     dp[u][1] := 0;
4     for child v of u do:
5         MiniVertexCover(v)
6         dp[u][0] += min(dp[v][0], dp[v][1])
7         dp[u][1] += dp[v][0]
8
9 MiniVertexCoverWrapper(T, root):
10    dp := [V+1][2] /*dp数组大小为2(V+1)的一维数组*/
11    dp(root)
12    return min(dp[root][0], dp[root][1])

```

MiniVertexCover.func