7. Dynamic Programming

注：以下答案中的所有伪代码均已编程验证。

Problem 7.1 (整数子集合问题)

用$d[k][i] = true/false$表示自然数 k 是否可以用自然数集合$A\{s_1, s_2, \dots, s_i\}$的子集和构成，当计算$d[k][i]$时，考虑是否将$s_i$加入子集和，我们可以得到如下状态转移方程：

$$d[k][i] = d[k][i-1] \vee d[k-s_i][i-1], s_i \le k$$

初始时设置$d[k][i] = false$， $d[0][i] = true(i = 1,2,\dots,k)$，表示任意集合的某个子集和（空集）都可以构成 0，$d[s_i][1] = true$，表示集合$A\{s_1\}$可以组成和s_i。

伪代码如下：

```
algorithm Subarray-Sum(S, A[1..n]):
    initialize array d[0..S][1..n] as false
    for i = 1 to n:
        d[0][i] = true
    d[A[1]][1] = true
    for k = 1 to S:
        for i = 2 to n:
            if A[i] <= k:
                d[k][i] = d[k][i - 1] || d[k - A[i]][i - 1]
            else:
                d[k][i] = d[k][i - 1]
    return d[S][n]
```

Problem 7.2

记$d[k]$表示将正整数 k 变为 1 需要的最少操作，状态转移方程为：

$$d[k] = \min\left\{d[k-1]+1, d\left[\frac{k}{2}\right]+1 \middle| k\text{是 2 的倍数}, d\left[\frac{k}{3}\right]+1 \middle| k\text{是 3 的倍数}\right\}$$

伪代码如下：

```
initialize array d[1..n]
d[1] = 0
for k = 2 to n:
    d[k] = d[k - 1] + 1
    if k % 2 == 0:
        d[k] = min(d[k], d[k/2] + 1)
    if k % 3 == 0:
        d[k] = min(d[k], d[k/3] + 1)
return d[n]
```

该算法的时间复杂度为$O(n)$。

Problem 7.3

记$d[k]$表示数组$A[1 \dots n]$中以$A[k]$结尾的最长非递减子序列长度，当我们计算$d[k+1]$时，我们只要从$A[k]$开始向前遍历，找出满足$A[i] \le A[k+1]$的$d[i]$的最大值$d[m]$，那么$d[k+1]$的值就为$d[m]+1$，状态转移方程为：

$$d[k] = \max\{d[i]+1 | \, d[i] \le d[k], i = 1 \dots k-1\}$$

因此$A[1 \dots n]$的最长非递减子序列就是$\max\{d[i] | i = 1 \dots n\}$。

伪代码如下：

```
initialize array d[1..n] as 0
d[1] = 1
```

```
maxLength = 1
for k = 2 to n:
    for i = k - 1 downto 1:
        if A[i] <= A[k]:
            d[k] = max(d[k], d[i] + 1)
    maxLength = max(maxLength, d[k])
return maxLength
```

该算法的时间复杂度为：

$$T(n) = O\left(\sum_{i=1}^{n-1} i\right) = O(n^2)$$

Problem 7.4

以下算法可以同时处理三小题。

记$\mathrm{d1[k]}$表示数组$\mathrm{A[1 \dots n]}$中以$\mathrm{A[k]}$结尾的子数组**最大乘积值**，记$\mathrm{d2[k]}$表示以$\mathrm{A[k]}$结尾的子数组**最小乘积值**，我们很容易得到如下状态转移方程：

$$\mathrm{d1[k]} = \max\{\mathrm{A[k]}, \mathrm{A[k]} \times \mathrm{d1[k-1]}, \mathrm{A[k]} \times \mathrm{d2[k-1]}\}$$
$$\mathrm{d2[k]} = \min\{\mathrm{A[k]}, \mathrm{A[k]} \times \mathrm{d1[k-1]}, \mathrm{A[k]} \times \mathrm{d2[k-1]}\}$$

伪代码如下：

```
initialize array d1[1..n] and d2[1..n]
d1[1] = A[1]
d2[1] = A[1]
maxValue = A[1]
for i = 2 to n:
    d1[i] = max(max(A[i] * d1[i - 1], A[i] * d2[i - 1]), A[i])
    d2[i] = min(min(A[i] * d1[i - 1], A[i] * d2[i - 1]), A[i])
    maxValue = max(max(d1[i], d2[i]), maxValue)
return maxValue
```

Problem 7.5

1. 用$\mathrm{d[i][j]}$表示两个字符子串$\mathrm{X_i} =< x_1, x_2, \dots, x_i >$和$\mathrm{Y_j} =< y_1, y_2, \dots, y_j >$的最长公共子序列长度。当$\mathrm{x_i = y_j}$时，显然有$\mathrm{d[i][j] = d[i-1][j-1] + 1}$；当$\mathrm{x_i \neq x_j}$时，有$\mathrm{d[i][j]} = \max\{\mathrm{d[i-1][j]}, \mathrm{d[i][j-1]}\}$。
   因此我们得到如下伪代码：

```
algorithm LCS(str1[1..m], str2[1..n]):
    initialize array d[0…m][0…n] as 0
    for i = 1 to m:
        for j = 1 to n:
            if str1[i] == str2[j]:
                d[i][j] = d[i - 1][j - 1] + 1
            else
                d[i][j] = max(d[i - 1][j], d[i][j - 1])
    return d[m][n]
```

注意到在上述代码中，我们定义的二维数组 d 维度为$\mathrm{(m+1, n+1)}$，并且初始化为 0，这样做的目的是让访问$\mathrm{d[i-1][j-1]}$的代码更简洁，不然需要判断i和j是否大于 0。

2. 用$\mathrm{d[i][j]}$表示两个字符子串$\mathrm{X_i} =< x_1, x_2, \dots, x_i >$和$\mathrm{Y_j} =< y_1, y_2, \dots, y_j >$的最长公共子序列长度。
   当$\mathrm{x_i = y_j}$时，因为 X 中字符可以重复出现，因此我们得到：

$$\mathrm{d[i][j]} = \max\{\mathrm{d[i-1][j-1]} + 1, \mathrm{d[i][j-1]} + 1\}$$

   当$\mathrm{x_i \neq x_j}$时：

$$d[i][j]=\max\{d[i-1][j],d[i][j-1]\}$$

伪代码如下：

```
algorithm LCS_2(str1[1…m], str2[1…n]):
    initialize array d[0…m][0…n] as 0
    for i = 1 to m:
        for j = 1 to n:
            if str1[i] == str2[j]:
                d[i][j] = max(d[i - 1][j - 1] + 1, d[i][j - 1] + 1)
            else
                d[i][j] = max(d[i - 1][j], d[i][j - 1])
    return d[m][n]
```

3. 用$d[i][j][h]$表示两个字符子串$X_i=<x_1,x_2,\dots,x_i>$和$Y_j=<y_1,y_2,\dots,y_j>$的最长公共子序列长度，且字符串X_i的字符x_i可以重复 h 次。

当$x_i=y_j$时，因为 X 中字符可以重复出现 k 次，因此我们得到：

$$d[i][j][h]=\max\{d[i-1][j-1][k]+1,d[i][j-1][h-1]+1\}$$

当$x_i\neq x_j$时：

$$d[i][j][h]=\max\{d[i-1][j][k],d[i][j-1][h]\}$$

伪代码如下：

```
algorithm LCS_3(str1[1..m], str2[1..n], k):
    initialize array d[0..m][0..n][0..k] as 0
    for i = 1 to m:
        for j = 1 to n:
            for h = 1 to k:
                if str1[i] == str2[j]:
                    d[i][j][h] = max(d[i - 1][j - 1][k] + 1, d[i][j – 1][h - 1] + 1)
                else:
                    d[i][j][h] = max(d[i - 1][j][k], d[i][j - 1][h])
    return d[m][n][k]
```

时间复杂度为$O(kmn)$。

Problem 7.6

用$d[i][j]$表示两个字符子串$A[1\dots i]$和$B[1\dots j]$的最短公共超序列长度。

初始化有$d[0][i]=i(i=0,1,\dots,n)$和$d[i][0]=i(i=0,1,\dots,m)$。

当$x_i=y_j$时：

$$d[i][j]=d[i-1][j-1]+1$$

当$x_i\neq x_j$时：

$$d[i][j]=\min\{d[i-1][j]+1,d[i][j-1]+1\}$$

伪代码如下：

```
algorithm Shorted-Super-Sequence(str1[1..m], str2[1..n]):
    initialize array d[0…m][0…n]
    for i = 0 to m:
        d[i][0] = i
    for i = 0 to n:
        d[0][i] = i
    for i = 1 to m:
        for j = 1 to n:
            if str1[i] == str2[j]:
                d[i][j] = d[i - 1][j - 1] + 1
```

```
            else
                d[i][j] = min(d[i - 1][j] + 1, d[i][j - 1] + 1)
    return d[m][n]
```

Problem 7.7

1. 错误，反例：$X =< AB >, Y =< BA >, Z =< ABAB >$，则有$LCS(X, Z) =< AB >, Z' = Z - LCS(X, Z) =< AB >$，算法出错。
2. 用$d[i][j] = true/false$表示两个字符子串$X_i =< x_1, x_2, \dots, x_i >$和$Y_j =< y_1, y_2, \dots, y_j >$是否可以组成长度为$i + j$的字符串$Z_{i+j} =< z_1, z_2, \dots, z_{i+j} >$。我们可以得到如下状态转移方程：
$$\begin{aligned} &d[i][j] = false && if\ \ x_i \neq z_{i+j}, y_j \neq z_{i+j} \\ &d[i][j] = d[i-1][j] && if\ \ x_i = z_{i+j}, y_j \neq z_{i+j} \\ &d[i][j] = d[i][j-1] && if\ \ x_i \neq z_{i+j}, y_j = z_{i+j} \\ &d[i][j] = d[i][j-1] \vee d[i-1][j] && if\ \ x_i = z_{i+j}, y_j = z_{i+j} \end{aligned}$$
初始时设置$d[0][0] = true$。
伪代码如下：
```
algorithm(X[1..m], Y[1..n], Z[1..k]):
    if k != m + n:
        return false
    initialize array d[0..m][0..n] as false
    d[0][0] = true
    for i = 0 to m:
        for j = 0 to n:
            if i > 0 && X[i] == Z[i + j]:
                d[i][j] = d[i - 1][j]
            if j > 0 && Y[j] == Z[i + j]:
                d[i][j] = d[i][j] || d[i][j - 1]
return d[m][n]
```
3. 用$d[i][j][h]$表示从$X_i =< x_1, x_2, \dots, x_i >$，$Y_j =< y_1, y_2, \dots, y_j >$，$Z_h =< z_1, z_2, \dots, z_h >$中删除元素使得合并成立的最小数目。通过判断$x_i, y_j, z_h$的关系，我们可以得到如下状态转移方程：
$$\begin{aligned} &d[i][j][h] = \min\{d[i-1][j][h-1], d[i][j-1][h]+1, d[i][j][h-1]+1\} && if\ \ x_i = z_h, y_j \neq z_h \\ &d[i][j][h] = \min\{d[i][j-1][h-1], d[i-1][j][h]+1, d[i][j][h-1]+1\} && if\ \ y_j = z_h, x_i \neq z_h \\ &d[i][j][h] = \min\{d[i][j-1][h-1], d[i-1][j][h-1]\} && if\ \ x_i = z_h, y_j = z_h \end{aligned}$$
初始时设置$d[0][0][h] = h, d[0][j][0] = j, d[i][0][0] = i$。
为了输出删除元素的集合，我们需要另一个数组$p[i][j][h]$记录删除的元素是哪个序列的最后元素。
c++代码如下：不想改成伪代码了╮(╯▽╰)╭
```
enum { eX = 0x1, eY = 0x2, eZ = 0x4 }; // 记录删除元素所属序列

void print_Deleted_Char(vector<vector<vector<int>>> &p, int m, int n, int k) {
    vector<int> X_result, Y_result, Z_result;
    while (!(m == 0 && n == 0 && k == 0)) {
        switch (p[m][n][k]) {
        case eX: // 删除 X 序列最后一个元素
            X_result.push_back(m - 1); // 记录删除的元素下标
            m--;
            break;
        case eY: // 删除 Y 序列最后一个元素
            Y_result.push_back(n - 1); // 记录删除的元素下标
            n--;
            break;
```

```
        case eZ: // 删除 Z 序列最后一个元素
            Z_result.push_back(k - 1); // 记录删除的元素下标
            k--;
            break;
        case eX | eZ: // 删除 X 和 Z 序列的最后元素，只会发生在 X 和 Z 序列最后元素相同时，此时不记删除数。
            m--;
            k--;
            break;
        case eY | eZ: // 删除 Y 和 Z 序列的最后元素，只会发生在 Y 和 Z 序列最后元素相同时，此时不记删除数。
            n--;
            k--;
            break;
        }
    }
    // 输出删除元素的下标
    cout << "X: ";
    for (int i = 0; i < X_result.size(); i++) {
        cout << X_result[i] << " ";
    }
    cout << endl;

    cout << "Y: ";
    for (int i = 0; i < Y_result.size(); i++) {
        cout << Y_result[i] << " ";
    }
    cout << endl;

    cout << "Z: ";
    for (int i = 0; i < Z_result.size(); i++) {
        cout << Z_result[i] << " ";
    }
    cout << endl;
}

int Min_Deletion_Count(string &X, string &Y, string &Z) {
    int m = X.size(), n = Y.size(), k = Z.size();
    vector<vector<vector<int>>> d(m + 1, vector<vector<int>>(n + 1, vector<int>(k + 1,
std::numeric_limits<int>::max())));
    vector<vector<vector<int>>> p(m + 1, vector<vector<int>>(n + 1, vector<int>(k + 1, 0)));
    for (int i = 0; i <= m; i++) {
        d[i][0][0] = i;
        p[i][0][0] = eX;
    }
    for (int i = 0; i <= n; i++) {
        d[0][i][0] = i;
        p[0][i][0] = eY;
    }
    for (int i = 0; i <= k; i++) {
        d[0][0][i] = i;
```

```
        p[0][0][i] = eZ;
    }
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            for (int h = 0; h <= k; h++) {
                if (i > 0 && h > 0 && X[i - 1] == Z[h - 1] && d[i - 1][j][h - 1] < d[i][j][h]) {
                    d[i][j][h] = d[i - 1][j][h - 1];
                    p[i][j][h] = eX | eZ;
                }
                if (j > 0 && h > 0 && Y[j - 1] == Z[h - 1] && d[i][j - 1][h - 1] < d[i][j][h]) {
                    d[i][j][h] = d[i][j - 1][h - 1];
                    p[i][j][h] = eY | eZ;
                }
                if (i > 0 && d[i - 1][j][h] + 1 < d[i][j][h]) {
                    d[i][j][h] = d[i - 1][j][h] + 1;
                    p[i][j][h] = eX;
                }
                if (j > 0 && d[i][j - 1][h] + 1 < d[i][j][h]) {
                    d[i][j][h] = d[i][j - 1][h] + 1;
                    p[i][j][h] = eY;
                }
                if (h > 0 && d[i][j][h - 1] + 1 < d[i][j][h]) {
                    d[i][j][h] = d[i][j][h - 1] + 1;
                    p[i][j][h] = eZ;
                }
            }
        }
    }

    print_Deleted_Char(p, m, n, k);
    return d[m][n][k];
}
```

若输入$X =< ABCo >, Y =< DEFo >, Z =< ooADBECF >$，则输出为：

```
X: 3
Y: 3
Z: 1 0
min count: 4
Press any key to continue . . .
```

Problem 7.8

记$d[i][j]$表示字符子串$T[i \dots j]$中包含前向子串包含$T[i]$，后向子串包含$T[j]$的最长子串长度，因为前向和后向子串不能重叠，所以有$d[i][j] = 0 (i \geq j)$。状态转移方程为：

$$
\begin{aligned}
d[i][j] &= d[i+1][j-1] + 1 \quad && \mathit{if} \;\; T[i] = T[j], i < j \\
d[i][j] &= 0 && \mathit{if} \;\; T[i] \neq T[j], i < j
\end{aligned}
$$

注意到$d[i][j]$依赖于$d[i+1][j-1]$，因此填表的顺序需要注意，伪代码如下：

```
initialize array d[0…m][0…m] as 0
maxValue = 0
for i = m downto 1:
    for j = i + 1 to m:
        if X[i] == X[j]:
```

```
            d[i][j] = d[i + 1][j - 1] + 1
        maxValue = max(maxValue, d[i][j])
return maxValue
```

注意到在上述代码中，我们定义的二维数组 d 维度为$(m+1, m+1)$，并且初始化为 0，这样做的目的是让访问$d[i+1][j-1]$的代码更简洁，不然需要判断j是否大于 0。

时间复杂度为$O(n^2)$。

Problem 7.9

1. 用$d[k] = true/false$表示字符子串$s[1 \ldots k]$是否可以重建，为方便起见记$d[0] = true$，我们可以得到如下状态转移方程：

$$d[k] = \bigvee_{i=0}^{k-1}\left(d[i] \wedge dict(s[i+1 \ldots k])\right)$$

判断$d[k]$的值我们需要进行$O(k)$时间，因此该算法的时间复杂度为：

$$T(n) = O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

伪代码如下：

```
initialize booleam array d[0…n] as false
d[0] = true
for k = 1 to n:
    for i = k-1 downto 0:
        d[k] = d[k] || (d[i] && dict(s[i+1…k]))
return d[n]
```

2. 为了得到单词序列，我们需要记录每个分割点的位置，用数组$x[1 \ldots n]$表示，其中$x[k] = i$表示分割字符串$s[1 \ldots k]$的分割点在$s[i]$，且$s[i+1 \ldots k]$是一个完整单词。为了输出单词，我们可以从$x[n]$开始反向查找，不过注意到这样输出的单词是反序的，为了得到正确的序列，我们可以用一个递归过程来实现，伪代码如下：

```
initialize boolean array d[0…n] as false
initialize array x[1…n] as -1
d[0] = true
for k = 1 to n:
    for i = k-1 downto 0:
        if (d[i] && dict(s[i+1…k])) == true:
            d[k] = true
            x[k] = i
output(n, x[1…n])

procedure output(k, x[1…n]):
    if x[k] == -1:
        return
    output(x[k], x[1…n])
    print(s[x[k]+1…k])
```

Problem 7.10

1. 记$d[i][j]$表示字符串$s[i \ldots j]$的最长回文子序列长度。我们可以得到如下状态转移方程：

$$\begin{aligned}
&d[i][j] = 0 && if\ \ i > j \\
&d[i][j] = d[i+1][j-1] + 2 && if\ \ s_i = s_j, i \le j \\
&d[i][j] = \max\{d[i+1][j], d[i][j-1]\} && if\ \ s_i \ne s_j, i \le j
\end{aligned}$$

伪代码如下，注意填表顺序：

```
algorithm Longest-Palindromic-Substring(s[1…n]):
    initialize array d[1…n+1][0…n] as 0
    maxValue = 0
    for i = n downto 1:
        for j = i to n:
            if i < j && s[i] == s[j]:
                d[i][j] = d[i + 1][j - 1] + 2;
            else if i == j && s[i - 1] == s[j - 1]:
                d[i][j] = d[i + 1][j - 1] + 1;
            else:
                d[i][j] = max(d[i + 1][j], d[i][j - 1]);
            maxValue = max(maxValue, d[i][j]);
    return maxValue
```

2. 记$\mathrm{d[i][j]}$表示字符串$\mathrm{s[i \dots j]}$可以拆分的最少的回文数量，为了计算$\mathrm{s[i \dots j]}$可以拆分的回文数量，这有两种情况：如果$\mathrm{s[i \dots j]}$本身就是一个回文串，则$\mathrm{d[i][j] = 1}$；否则，我们计算$\min\{\mathrm{d[i][k] + d[k+1][j]}\ \mid \mathrm{k = i \dots j - 1}\}$。该算法的时间复杂度为：

$$\mathrm{T(n)} = \sum_{k=1}^{n-1} k \times (n-k) = \mathrm{O(n^3)}$$

伪代码如下：

```
algorithm Min-Division-Count(s[1..n]):
    initialize array d[1..n][1..n] as infinity
    for i = n downto 1:
        for j = i to n:
            if is_palindromic(s, i, j):
                d[i][j] = 1
            else:
                for k = i to j-1:
                    d[i][j] = min(d[i][j], d[i][k] + d[k + 1][j])
    return d[1][m]

procedure is_palindromic(s, l, r):
    while l <= r:
        if s[l] != s[r]:
            return false
        l = l + 1
        r = r - 1
    return true
```

Problem 7.11

假设字符串为$\mathrm{s[1 \dots n]}$，分割点位置由$\mathrm{L[1 \dots m]}$给出，$\mathrm{L[i]}$表示第i个分割点将字符串分成了长度为$\mathrm{L[i]}$和$\mathrm{n - L[i]}$的两部分，在这里我们假设$\mathrm{L[1 \dots m]}$给出的分割点是单调递增且互不相同的。

记$\mathrm{d[i][j]}$表示字符串$\mathrm{s\big[L[i] \dots L[j]\big]}$分割的最小代价，显然有$\mathrm{s\big[L[i], L[i+1]\big] = 0}$。为了方便起见，我们在$\mathrm{L[1 \dots m]}$中额外添加两个分割点$\mathrm{L[1] = 0, L[m] = n}$，这样最终结果就可以表示为$\mathrm{d[0][m]}$。

我们可以得到如下状态转移方程：

$$\mathrm{d[i][j] = L[j] - L[i] + \min\{d[i][k] + d[k][j] \mid k = i, i+1, \dots, j\}}$$

伪代码如下：

```
algorithm Min-Division-Price(n, L[1..m]):
    L.push_front(0)  // 在字符串开头和末尾添加两个分割点，便于编写代码
    L.push_back(n)
    m = L.size()     // m = m + 2
    initialize array d[1..m][1..m] as infinity
    for i = 1 to m - 1:
        d[i][i + 1] = 0
    for i = m downto 1:
        for j = i + 1 to m:
            for k = i + 1 to j - 1:
                d[i][j] = min(d[i][j], d[i][k] + d[k][j] + L[j] - L[i])
    return d[1][m]
```

该算法的时间复杂度为$\mathrm{O}(\mathrm{n}^2)$。

Problem 7.12

1. 记$\mathrm{d}[\mathrm{k}] = true/false$表示金额 k 是否可以兑换成硬币，$\mathrm{x}[1 \dots \mathrm{n}]$表示 n 种硬币面额，我们很容易得到如下状态转移方程：

$$\mathrm{d}[\mathrm{k}] = \bigvee_{i=1,2,\dots,n}^{x[i]\le k} d\big[k - x[i]\big]$$

判断$\mathrm{d}[\mathrm{k}]$的值需要进行$\mathrm{O}(\mathrm{n})$时间，因此该算法的时间复杂度为：

$$\mathrm{T}(\mathrm{n}) = O\left(\sum_{i=1}^{v} n\right) = O(nv)$$

伪代码如下：

```
algorithm(v, x[1..n]):
    initialize array d[0..v] as false
    d[0] = true
    for k = 1 to v:
        for i = 1 to n:
            if x[i] <= k:
                d[k] = d[k] || d[k - x[i]]
    return d[v]
```

2. 用$\mathrm{d}[\mathrm{k}][\mathrm{i}] = true/false$表示金额 k 是否可以用硬币$\mathrm{x}[1 \dots \mathrm{i}]$兑换，我们可以得到如下状态转移方程：

$$\mathrm{d}[\mathrm{k}][\mathrm{i}] = \mathrm{d}[\mathrm{k}][\mathrm{i}-1] \vee \mathrm{d}\big[\mathrm{k}-\mathrm{x}[\mathrm{i}]\big][\mathrm{i}-1], \mathrm{x}[\mathrm{i}] \le \mathrm{k}$$

初始时设置$\mathrm{d}[\mathrm{k}][\mathrm{i}] = \mathrm{false}$，$\mathrm{d}[0][\mathrm{i}] = \mathrm{true}(\mathrm{i} = 1,2,\dots,\mathrm{k})$, 表示用任意硬币都可以兑换金额 0, $\mathrm{d}\big[\mathrm{x}[1]\big][1] = \mathrm{true}$, 表示只用一个硬币$\mathrm{x}[1]$可以兑换金额$\mathrm{x}[1]$。

伪代码如下：

```
algorithm_2(v, x[1..n]):
    initialize array d[0..v][1..n] as false
    for i = 1 to n:
        d[0][i] = true
    d[x[1]][1] = true
    for k = 1 to v:
        for i = 2 to n:
            if x[i] <= k:
                d[k][i] = d[k][i - 1] || d[k - x[i]][i - 1]
            else:
```

```
            d[k][i] = d[k][i - 1]
    return d[v][n]
```

3. 用$\mathrm{d[i]}[k] = true/false$表示金额 i 是否可以用 k 枚硬币兑换，我们可以得到如下状态转移方程：

$$\mathrm{d[i][k]} = \bigvee_{j=1,2,\dots,n}^{x[j]\le i} d\big[i - x[j]\big][k-1]$$

设定初始值$\mathrm{d[i][k] = false}$，$\mathrm{d[0][k] = true(k = 0,1,\dots,k)}$

```
algorithm_3(v, x[1..n], k):
    initialize array d[0..v][0..k] as false
    for i = 0 to k:
        d[0][i] = true
    for i = 1 to v:
        for j = 1 to k:
            for h = 1 to n:
                if x[h] <= i:
                    d[i][j] = d[i][j] || d[i - x[h]][j - 1]
    return d[v][k]
```

该算法的时间复杂度为$O(knv)$。

Problem 7.13　　此题未验证

首先从根节点开始用一次 BFS 为顶点编号为$1,\dots,n$，用$\mathrm{Vertex[1\dots n]}$记录，$\mathrm{Vertex[i]}$表示编号为 i 的顶点，采用 BFS 算法我们可以得到一个性质：顶点 k 的所有子节点的编号都比 k 大。

用$\mathrm{d[k]}$表示以 k 为顶点的子树的最小顶点覆盖数目。当计算$\mathrm{d[k]}$时，考虑是否将顶点 k 加入顶点覆盖：

若加入顶点 k，可以得到递推式为：

$$\mathrm{d[k]} = 1 + \sum\nolimits_{v\in Vertex(k).children} d[v]$$

若不加入顶点 k，则为了达到最小顶点覆盖，我们需要加入顶点 k 的所有子节点，递推式为：

$$\mathrm{d[k]} = |\mathrm{Vertex(k).children}| + \sum\nolimits_{v\in Vertex(k).children.children} d[v]$$

因此我们可以得到如下状态转移方程：

$$\mathrm{d[k]} = \min\left\{1 + \sum\nolimits_{v\in Vertex(k).children} d[v], |\mathrm{Vertex(k).children}| + \sum\nolimits_{v\in Vertex(k).children.children} d[v]\right\}$$

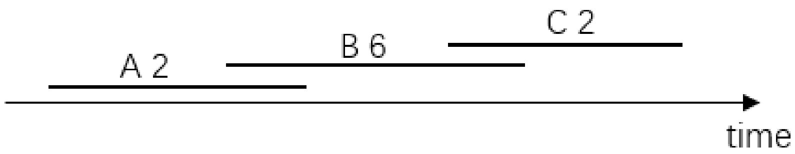
伪代码如下：

```
algorithm Min-Coverage(G(V, E)):
    n = V.size()
    initialize array d[1..n] as infinity
    use BFS(G) to compute Vertex[1..n]
    for i = n downto 1:
        // 如果顶点 i 加入覆盖集
        tmp = 1
        for each v in Vertex[i].children:
            tmp = tmp + d[v]
        d[i] = tmp
        // 如果顶点 i 不加入覆盖集
        tmp = 0
        for each v in Vertex[i].children:
            tmp = tmp + 1
            for each w in v.children:
```

```
                tmp = tmp + d[w]
        d[i] = min(d[i], tmp)
    return d[1]
```

Problem 7.14　此题未验证

1. 反例如下：



若用贪心算法，则应该选择课程 A 和 C，得到 4 学分，但是答案应该是选择课程 B，这样可以得到 6 个学分。

2. 假设课程存储于$\mathrm{course}[1 \dots n]$中，课程开始时间，结束时间，学分分别为$\mathrm{course}[i].\mathrm{startTime}$，$\mathrm{course}[i].\mathrm{finishTime}$，$\mathrm{course}[i].\mathrm{credit}$。我们需要按课程结束时间对 n 个课程进行排序。

用$\mathrm{d}[i]$表示在课程 i 已经被选中的前提下，我们在课程$\mathrm{course}[1 \dots i]$中可以得到的最多学分。为了计算$\mathrm{d}[i]$，我们需要找出与课程 i 不冲突的序号最大的课程$\mathrm{course}[k]$，$k < i$，则：

$$\mathrm{d}[i] = \mathrm{course}[i].\mathrm{credit} + \mathrm{d}[k]$$

伪代码如下：

```
algorithm Select-Course(course[1..n]):
    sort course[1..n] based on course[i].finishTime
    initialize array d[1..n]
    for i = 1 to n:
        d[i] = course[i].credit
        for k = i - 1 downto 1:
            if course[k].finishTime <= course[i].startTime:
                d[i] += d[k]
                break;
    return d[n]
```

该算法的时间复杂度为：

$$\mathrm{T(n)} = O\left(\sum_{i=1}^{n-1} i\right) = O(n^2)$$

3. 第二题就是一个动态规划算法。

Problem 7.15

用$\mathrm{d}[i]$表示一段时间以后旅行到旅店a_i得到的最小总惩罚，为了计算$\mathrm{d}[i]$，我们需要考察前一天可能住宿的旅店位置，用状态转移方程表示为：

$$\mathrm{d}[i] = \min\left\{\left(200 - (a_i - a_k)\right)^2 + \mathrm{d}[k] \middle| a_i - a_k \le 200, k = 1,2,\dots,i-1\right\}$$

为了记录旅店序列，我们需要另一个数组$\mathrm{p}[1 \dots n]$，$\mathrm{p}[k]$记录了在旅店 k 之前入住的旅店。为了按序打印出旅店序列，我们需要一个递归过程。伪代码如下：

```
algorithm Min-Traveling-Price(a[1..n]):
    initialize array d[1..n] as infinity
    initialize array p[1..n] as -1
    d[1] = 0
    for i = 2 to n:
        for k = i - 1 downto 1:
            if a[i] - a[k] <= 200 && ((200 - a[i] + a[k]) * (200 - a[i] + a[k]) + d[k]) < d[i]:
                d[i] = ((200 - a[i] + a[k]) * (200 - a[i] + a[k]) + d[k])
                p[i] = k
```

```
    Print-Hotel(p, n)

procedure Print-Hotel(p[1..n], k):
    if k == -1:
        return
    Print-Hotel(p, p[k])
    print(k)
```

该算法的时间复杂度为：

$$T(n) = O\left(\sum_{i=1}^{n-1} i\right) = O(n^2)$$

Problem 7.16

用$d[i]$表示在修建第 i 个酒店的前提下，距离高速公路入口m_i公里范围内，修建酒店的最大利润。考虑酒店之间的距离限制，我们可以得到如下状态转移方程：

$$d[i] = \max\{p_i + d[j] \ | m_i - m_j \geq k, j = 1,2, \ldots, i-1\}$$

伪代码如下：

```
algorithm Max-Hotel-Profit(m[1..n], p[1..n], k):
    initialize array d[1..n] as 0
    d[1] = p[1]
    for i = 2 to n:
        for j = i - 1 downto 1:
            if m[i] - m[j] >= k:
                d[i] = max(d[i], p[i] + d[j])
    return d[n]
```

该算法的时间复杂度为$T(n) = O(n^2)$。

Problem 7.17

假设棋盘格子分数记录在$board[i][j]$中。

用$d[i][j]$表示以第 i 行，第 j 列为起点，可以得到的最大得分。因为只能向下或者向右移动一个格子，因此我们可以得到如下状态转移方程：

$$d[i][j] = board[i][j] + \max\{d[i+1][j], d[i][j+1]\}$$

这题需要注意填表顺序，伪代码如下：

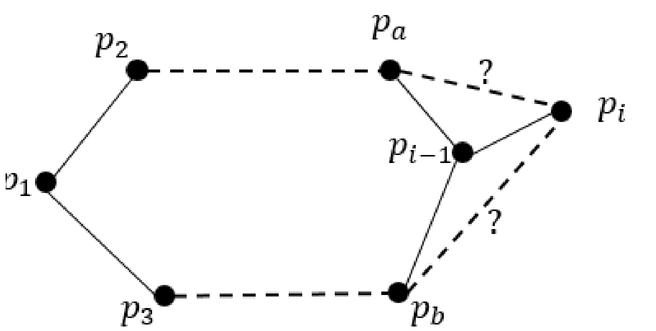
```
algorithm Max-Board-Score(board[1..n][1..n]):
    initialize array d[1..n+1][1..n+1] as 0
    maxValue = -infinity
    for i = n downto 1:
        for j = n downto 1:
            d[i][j] = board[i][j] + max(d[i + 1][j], d[i][j + 1])
            if d[i][j] > maxValue:
                maxValue = d[i][j]
    return maxValue
```

注意到我们设置的$d[n+1][n+1]$有$n+1$维，且初始化为 0，这样是为了方便编写代码，不然还需要判断$i < n$和$j < n$。

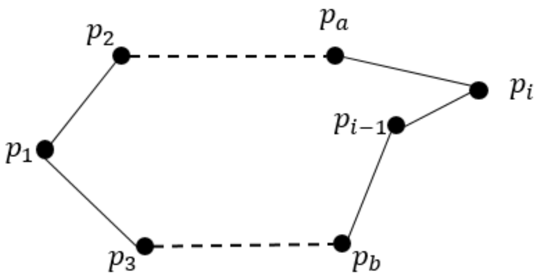
该算法的时间复杂度为$O(n^2)$。

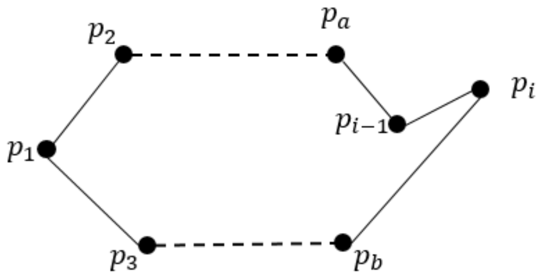
Problem 7.18　此题未验证

用$d[i]$表示区 i 个目的地$\{d_1, d_2, \ldots, d_i\}$送餐需要的最短外卖路线。$d[1], d[2], d[3]$都是简单情形，可以直接计算。当我们从$d[i-1]$计算$d[i]$时，如下图路线图所示：



我们可以证明$\mathrm{p_i}$和$\mathrm{p_{i-1}}$一定是相连的，$\mathrm{p_i}$插入的位置要么是在$\mathrm{p_a}$和$\mathrm{p_{i-1}}$之间，要么是在$\mathrm{p_b}$和$\mathrm{p_{i-1}}$之间，其中$\mathrm{p_a}$和$\mathrm{p_b}$是与$\mathrm{p_{i-1}}$直接相连的节点。这样就形成了如下两种情况：





我们只要从中选择一个较小的即构成了d[i]的解。

伪代码如下：

```
algorithm Min-Deliver-Distance(p[1..n]):
    initialize array d[1..n]
    d[1] = 0
    d[2] = dist(1, 2)
    d[3] = dist(1, 2) + dist(2, 3) + dist(1, 3)
    pa = 1
    pb = 2
    for i = 4 to n:
        // 选择第一种情况
        if dist(pa, i) - dist(a, i - 1) < dist(pb, i) - dist(b, i - 1):
            d[i] = d[i - 1] + dist(i - 1, i) + dist(pa, i) - dist(a, i - 1)
            pb = i – 1
        // 选择第二种情况
        else:
            d[i] = d[i - 1] + dist(i - 1, i) + dist(pb, i) - dist(b, i - 1)
            pa = i - 1
    return d[n]
```

姓名：陆依鸣
学号：151220066
邮箱：luyimingchn@gmail.com