Problem 1.

See the zip file for source code.

Problem 2. All-Pairs Shortest-Paths Problem(15%)

1. $\forall i \in V, j \in V, d_{ij}^{(0)} = w(i,j)$
2. $\forall i \in V, j \in V, k \in V, d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
3. $\forall i \in V, j \in V, d_{ij} = d_{ij}^{(n)}$
4.

```
1 FLOYD-WARSHALL(V, w)
2     n = |V|
3     for i = 1 to n
4         for j = 1 to n
5             d[i][j]=w(i,j)
6     for k = 1 to n
7         for i = 1 to n
8             for j = 1 to n
9                 d[i][j] = min{d[i][j], d[i][k]+d[k][j]}
```

The space required for V is $O(n)$. The space required for w and d are $O(n^2)$. Therefore, it only requires $O(n^2)$ space.

The implementation is still correct if `d[i][j]` is the same as $d_{ij}^{(k-1)}$ and `d[i][k]+d[k][j]` is the same as $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ right before running line 9 of the code. After leaving for-loop in line 3, `d[i][j]` would be $w(i,j)$ which is the same as $d_{ij}^{(0)}$. Before running line 9 of the code, `d[i][k]+d[k][j]` may be either $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ or $d_{ik}^{(k)} + d_{kj}^{(k)}$ and `d[i][j]` must be $d_{ij}^{(k-1)}$ because we are about going to do the k-th update. Recall the definition of $d^{(k-1)}$ and $d^{(k)}$. It's easy to see that $d_{ik}^{(k-1)}$ must be the same as $d_{ik}^{(k)}$ and $d_{kj}^{(k-1)}$ must be the same as $d_{kj}^{(k)}$. Therefore, our implementation is still correct.

Problem 3. (15%)

a. (8%)

```
int howManyCoin(int M, int *L, int n)
    // the number of coins never bigger than M
    int maxCoinNum=M+1, dp[max_M], i, j;
    for(j=0; j<=M; j++)
        dp[j]=maxCoinNum;
    for(i=1; i<=n; i++)
        for(dp[L[i]]=1, j=L[i]+1; j<=M; j++)
            dp[j]=min(dp[j-L[i]]+1, dp[j]);
    return (dp[M]==maxCoinNum) ? (-1) : (dp[M]);
```

Basic Idea:

當 i=1 時，dp[x] 代表「若有 L[1] 這種幣值，就能組合出 x 元之最少錢幣數」

當 i=2 時，dp[x] 代表「若有 L[1] 和 L[2] 之幣值，能組合出 x 元之最少錢幣數」

以此類推，當 i=n 時，dp[M] 即為「若有 L[1...n] 這些幣值，能組合出 M 元之最少錢幣數」

Running time:

做 n 次，每次都跑 $O(M)$ 個陣列元素 => 整個演算法的複雜度是 $O(Mn)$

b. (7%)

```
int howManyCoin_advanced(int M, int *L, int *N, int n)
    int maxCoinNum=M+1, dp[max_M], i, j, cnt[max_M];
    for(j=0; j<=M; j++)
        dp[j]=maxCoinNum;
        cnt[j]=0;
    for(i=1; i<=n; i++)
        memset(cnt, 0, (M+1)*sizeof(int));
        for(dp[L[i]]=1, cnt[L[i]]=1, j=L[i]+1; j<=M; j++)
            if(cnt[j-L[i]]<N[i] && dp[j-L[i]]+1 < dp[j])
                dp[j]=dp[j-L[i]]+1;
                cnt[j]=cnt[j-L[i]]+1;
    return (dp[M]==maxCoinNum) ? (-1) : (dp[M]);
```

多了一個 cnt 陣列，紀錄用了多少個幣值為 L[i] 的錢幣

=> 所以每次 i 增加時，cnt 陣列都要重新歸零)

=> 因為「dp[j-L[i]]+1」的「+1」代表「多用了一個 L[i] 幣值的錢幣」，所以在讓他多用之前，要檢查 cnt 是不是超過上限了，所以才要寫「cnt[j-L[i]]<N[i]」

Running time:

做 n 次，每次都跑 $O(M)$ 個陣列元素 => 整個演算法的複雜度是 $O(Mn)$

Problem 4. (15%)

1. (5%)

   To find the longest palindrome sebsequence in string str, we just need to reverse str to be str' and find the longest common sebsequence of str and str'.

```
int table[str.len][str.len] = {0}
int LPS(string str)
    string str' = reverse(str)
    for i = 1 to str.len-1
        for j = 1 to str.len-1
            if str[i] == str'[j]
                table[i][j] = table[i-1][j-1]+1
            else
                table[i][j] = max(table[i-1][j], table[i][j-1])
    return table[str.len-1][str.len-1]
```

   From the pseudo code we need to travel i and j from 0 to str.len-1, so the time complexity would be $O(\text{str.len}^2) = O(n^2)$.

2. (10%)

```
//table[i][j] = the minimum partition of str.substring(i, j)
int table[str.len][str.len] = {1}
int divide(string str)
    for i = 1 to str.len-1
        for j = 0 to (str.len-1)-i
            //strcmp(x, reverse(x))
            if(isPalindrome(str.substring(j, j+i)))
                table[j][j+i] = 1
            else
                min = i
                for k = j+1 to j+i-1
                    min = min(min, table[j][k]+table[k+1][j+i])
                table[j][j+i] = min
    return table[0][str.len-1]
```

   From the pseudo code we need to travel i and j with size str.len, and function isPalindrome takes O(str.len) to compute and the for loop of k takes str.len time, so the time complexity would be $O(\text{str.len}^2 * (\text{str.len}+\text{str.len})) = O(\text{str.len}^3) = O(n^3)$.

Problem 5. (20%)

The following pseudo code implements one solution to this problem.

```
1  int include[n]
2  int exclude[n]
3  traverse(int current)
4      include[current] = nodes[current].capacity
5      exclude[current] = 0
6      for child in nodes[current].children
7          traverse(child)
8          include[current] += exclude[child]
9          exclude[current] += max(include[child], exclude[child])
10
11 \\ Print the maximum allocation of birds.
12 backtrack(int current, bool occupied)
13     if occupied
14         print current
15     for child in nodes[current].children
16         backtrack(child, !occupied && include[child] > exclude[child])
17
18 find_max_number_of_birds()
19     traverse(root)
20     max_number = max(include[root], exclude[root])
21     print max_number \\ The maximum number of birds in the tree
22     backtrack(root, include[root] > exclude[root])
```

The basic idea of the solution is to compute include[i] and exclude[i] from leaves to the root. include[i] is the maximum number of birds in the subtree rooted at i with the constraint that node i must be occupied by birds; exclude[i] is the maximum number of birds in the subtree rooted at i with the constraint that there are no birds on node i. Once we have all include[child] and exclude[child] of node i, it is easy to compute include[i] and exclude[i]. That is, include[i] is the sum of the capacity of node i and all exclude[child], and exclude[i] is the sum of max(include[child], exclude[child]). After the traveral, we can get the answer, max(include[root], exclude[root]).

One way to give the maximum allocation of birds is backtracking. The procedure starts from root to leaves, and it uses the information of whether the node is occupied, include[child] and exclude[child] to determine whether the child is occupied.

In this pseudo code, we traverse all the tree nodes twice. It requires $2n$ visits. Therefore, the procedure can run in $O(n)$.