Q1:

Notation:

string starts from index 1.

"$x^{(1)}[n_1]$" means string 1 with characters indexing from $1\ to\ n_1$.

"$x^{(m)}[n_m]$" means string m with characters indexing from $1\ to\ n_m$.

"$x^{(m)}.at(n_m)$" means the $n_m th$ character of $x^{(m)}$.

$$LCS\left(x^{(1)}[n_1], \dots, x^{(m)}[n_m]\right) = \begin{cases} 1 + LCS\left(x^{(1)}[n_1 - 1], \dots, x^{(m)}[n_m - 1]\right) & if\ x^{(1)}.at(n_1) = \cdots = x^{(m)}.at(n_m) \\ maximum \begin{cases} LCS\left(x^{(1)}[0, \dots, n_1 - 1], x^{(2)}[0, \dots, n_2], \dots, x^{(m)}[0, \dots, n_m]\right) \\ LCS\left(x^{(1)}[0, \dots, n_1], \dots, x^{(i)}[0, \dots, n_i - 1], \dots, x^{(m)}[0, \dots, n_m]\right) \\ LCS\left(x^{(1)}[0, \dots, n_1 - 1], \dots, x^{(m-1)}[0, \dots, n_{m-1}], x^{(m)}[0, \dots, n_m - 1]\right) \end{cases} & otherwise \end{cases}$$

```
// for any 1 <= n <= m, string xi has length n1.
LCS(string x1, string x2, ... , string xm){
     int Longest[n1][n2]...[nm];
     for (int nn1 = 1; nn1 <= n1; ++nn1)
     ... // nested for loop from n2 to n(m-1)
     for (int nnm = 1; nnm <= n2; ++nnm){
          if (nn1 == 0 || nn2 == 0 || ... || nnm = 0){
               Longest[nn1][nn2]...[nnm] = 0;
          } else if (x1.at(nn1) == x2.at(nn2) == ... == xm.at(nnm)){
               Longest[nn1][nn2]...[nnm] = 1 + Longest[nn1 - 1][nn2 - 1]...[nnm - 1];
          } else {
               Longest[nn1][nn2]...[nnm] = max(Longest[nn1 - 1][nn2]...[nnm],
                                               Longest[nn1][nn2 - 1]...[nnm]
                                               ...
                                               Longest[nn1][nn2]...[nnm - 1]);
          }
     }
     ... // nested for loop
     }
     return Longest[n1][n2]...[nm];
}
```

The longest common subsequence(LCS) of m strings is either 1 + (LCS of strings dropping the last character if they are the same), or it is the LCS of m strings with one of them dropping the last character.

The algorithm goes though a “m dimensional” grid, it has $(n_1 * n_2 * \dots * n_m)$ cells. For each cell, we need at most m look ups in the table, each look up takes $\theta(1)$ time. In total, filling up the array and then returning the result takes $\theta(n_1 * n_2 * \dots * n_m * m)$ time.

Q2:

Since we need to put books on the shelf in order and minimize the number of bookshelves used, we should try to put as many books on each bookshelf as we can.

```
LibraryBookshelf(thickness[1..n], W){
    int remain = 0;
    int num_shelves = 0;
    int index = 1;
    while (index <= n){
        //not enough space
        if (remain < thickness[index]){
            remain = W;
            ++num_shelves;
        }
        // put book on the shelf
        remain -= thickness[index];
        ++index;
    }
    return num_shelves;
}
```

Since the books have to be put on the shelves in order, we cannot swap the position of books. Assume we have computed an arrangement of books by this algorithm. Assume it can be further optimized, then some book on some shelf must be put on the previous shelf to leave some space, so that the last shelf can be emptied. However, since this is a test case of this algorithm, no books can be put on the previous shelf. Hence the arrangement cannot be further optimized, the greedy algorithm gives the optimal solution.

The algorithm goes though each book once, so the algorithm has time complexity $\theta(n)$.

Q3:

For each employer we have three options: B_1 hires the person; B_2 hires the person; none the person.

$$HIRING(s[1 \dots n], r_1[1 \dots n], r_2[1 \dots n], n, B_1, B_2)$$
$$= max \begin{cases} r_1[n] - s[n] + HIRING(s[1 \dots n-1], r_1[1 \dots n-1], r_2[1 \dots n-1], n-1, B_1 - s[n], B_2) \\ r_2[n] - s[n] + HIRING(s[1 \dots n-1], r_1[1 \dots n-1], r_2[1 \dots n-1], n-1, B_1, B_2 - s[n]) \\ HIRING(s[1 \dots n-1], r_1[1 \dots n-1], r_2[1 \dots n-1], n-1, B_1, B_2) \end{cases}$$

Additional constraint is that the first option is available only if $B_1 - s[n] \geq 0$, the second option is available only if $B_2 - s[n] \geq 0$, the third option is always available.

We can store the results in an n by n grid. For each row, since the other company cannot hire the person, it is a simple dynamic programming problem that whether a company should hire the person.

$$HIRING2(s[1 \dots n], R[1 \dots n], n, Balance)$$
$$= max \begin{cases} R[n] - s[n] + HIRING2(s[1 \dots n-1], R[1 \dots n-1], n-1, Balance - s[n]) \\ HIRING2(s[1 \dots n-1], R[1 \dots n-1], n-1, Balance) \end{cases}$$

Similarly, the first option is available only if $Balance - s[n] \geq 0$.

The algorithm should be filling up a n by n grid, filling up each cell takes constant time, so the total run time should be in $\theta(n^2)$.

Q4:

We can define valid “scriptio continua” as follows:

1: an empty word is a “scriptio continua”;

2: if $ISWORD(x)$, then x is a “scriptio continua”;

3: if x and y are all “scriptio continua”, then $x + y$(concatenation) is a “scriptio continua”;

4: nothing else is a “scriptio continua”.

Then we have recurrence relationship as follows:

$$sc(1..n) = \big(sc(1) \wedge sc(2..n)\big) \vee \big(sc(1..2) \wedge sc(3..n)\big) \vee \ldots \vee \big(sc(1..n-1) \wedge sc(n)\big)$$

By analysing the recurrence, we know that to identify whether a string is a “scriptio continua”, we need to find a “space”, such that the string before and after this “space” are both “scriptio continua”. To avoid repeated test, we can save the result that whether string from 1 to n is a “scriptio continua”.

```
SriptioContinua(s[1..n]){
     if (n == 0){
          return true;
       }
     // SC[i] = true if s[1..i] is a sriptio continua, false other wise.
     bool array[n] = false;
     for (int end = 1; end <= n; ++end){
          SC[end] = ISWORD(s[1..end]);
          for (int i = 1; i < end; ++i){
               if (SC[end]){
                    break;
                      }
               SC[end] = SC[i] && ISWORD(s[i+1..end]);
          }
     }
     return SC[n];
}
```

Running time:

Inner loop has run time $\theta(n)$, loops for $\theta(n)$ times, in total the running time is $\theta(n^2)$.

Q5:

We compute the ratio of $\frac{v_i}{w_i}$ for all i, and sort them in increasing order. According to this ratio, we arrange the shipment from the factories. The smaller the value of $\frac{v_i}{w_i}$ is, it is cheaper that we choose to ship paper from V warehouse than average.

```
SHIPPING(rV, rW, d[1..n], v[1..n], w[1..n]){
    //ratio_list saves the ratio of v[i] / w[i] and its original index.
    map<double, int> ratio_list;
    for(int i = 1; i <= n; ++n){
        ratio_list.insert( pair(v[i]/w[i], i) );
    }

    //sort the ratio_list by its first field.
    //since we are using map, it is already sorted.

    double answer[n] = 0;
    double V_balance = rV;
    for(int i = 1; i <= n; ++n){
        int factory = ratio_list.at(i).second;
        if (d[factory] > V_balance){
            answer[i] = V_balance;
            break;
        } else {
            V_balance -= d[factory];
            answer[i] += d[factory];
        }
    }

    return answer;
}
```

Assume after this algorithm we obtained a result:

…	…	X	Y	…
V	…	d[x]	0	…
W	…	0	d[y]	…

Assume there is an different arrangement:

…	…	x	Y	…
V	…	0	d[x]	…
W	…	d[x]	d[y]- d[x]	…

Algorithm result requires cost

$$v[x]d[x] + w[y]d[y]$$

The other result requires cost

$$w[x]d[x] + w[y](d[y] - d[x]) + v[y]d[x]$$

the "additional cost" is:

$$\begin{aligned} & w[x]d[x] + w[y](d[y] - d[x]) + v[y]d[x] - v[x]d[x] - w[y]d[y] \\ &= w[x]d[x] + w[y]d[y] - w[y]d[x] + v[y]d[x] - v[x]d[x] - w[y]d[y] \\ &= d[x](w[x] - w[y] + v[y] - v[x]) \end{aligned}$$

Since by algorithm, $\frac{v[x]}{w[x]} < \frac{v[y]}{w[y]}$, for simplicity we assume $w[x] = w[y]$ and $v[x] < v[y]$

Then

$$\begin{aligned} & d[x](w[x] - w[y] + v[y] - v[x]) \\ &= d[x](v[y] - v[x]) \\ &> 0 \end{aligned}$$

Since the "additional cost" is greater than 0, the algorithm solution is the optimal.

Initialize the map "`ratio_list`" takes $\theta(n \log n)$ time, generating answer array takes $\theta(n)$ time, In total the algorithm takes $\theta(n \log n)$ time.