Q1

Here follows Java/pseudocode for a polynomial-time dynamic programming algorithm that produces an optimal shipping schedule.

```
/* Java/pseudocode for optimal shipping schedule */
ScheduleMaker(int[] supply, int r, int c){

   String[] schedule = new String[supply.length];
   int[] best = new int[supply.length];

   // iterates through the supply array, storing the optimal value in 'best' and
   // schedule type in 'schedule'
   for (int i = 0; i < supply.length; i++){
      if ( r*supply[i] + best[i-1] < 4c + best[i-4] ){
         best[i] = r*supply[i] + best[i-1];
         schedule[i] = A;
      }
      else{
         best[i] = 4c + best[i-4];
         schedule[i] = B;
      }
   }

   // This backtracks through to ensure that the proper schedule type is listed
   int i = supply.length;
   while ( i > 0 ){
      if ( schedule[i] == A){
         i--;
      }
      else{
         schedule[i-1] = B;
         schedule[i-2] = B;
         schedule[i-3] = B;
         i = i-4;
      }
   }

   return schedule;

}
```

Here is the result of running this algorithm on the sample set (11, 9, 9, 12, 12, 12, 12, 9, 9, 11) when r=1 and c=10.

n	choice 1	choice 2	schedule
1	**11**		A
2	**20**		A
3	**29**		A
4	29+12	**40**	B
5	40+12	**40+11**	B
6	51+12	**40+20**	B
7	60+12	**40+29**	B
8	**69+9**	40+40	A
9	**78+9**	40+51	A
10	**87+11**	40+60	A

Q2 CLRS 29.4-3

From page 860 of the text, and equations for a linear programming formulation of the Max Flow problem are as follows:

$$\text{maximize:} \tag{1}$$

$$\sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \tag{2}$$

$$\text{subject to:} \tag{3}$$

$$f_{uv} \leq c(u,v) \qquad \text{for each } u, v \in V \tag{4}$$

$$\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \qquad \text{for each } u \in V - \{s,t\} \tag{5}$$

$$f_{uv} \geq 0 \qquad \text{for each } u, v \in V \tag{6}$$

This can be converted to the standard form. This makes it easier to convert the primal to the dual.

$$\text{maximize:} \tag{7}$$

$$\sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \tag{8}$$

$$\text{subject to:} \tag{9}$$

$$f_{uv} \leq c(u,v) \qquad \text{for each } u, v \in V \tag{10}$$

$$\sum_{v \in V} f_{vu} - \sum_{v \in V} f_{uv} \leq 0 \qquad \text{for each } u \in V - \{s,t\} \tag{11}$$

$$\sum_{v \in V} -f_{vu} - \sum_{v \in V} -f_{uv} \leq 0 \qquad \text{for each } u \in V - \{s,t\} \tag{12}$$

$$f_{uv} \geq 0 \qquad \text{for each } u, v \in V \tag{13}$$

The conversion can be done utilizing the following rule:

If the primal is:

$$\text{maximize} \quad c^T x \quad \text{such that} \quad \begin{cases} Ax \leq b \\ x \geq 0 \end{cases} \tag{14}$$

Then the dual is:

$$\text{minimize} \quad b^T y \quad \text{such that} \quad \begin{cases} A^T y \geq c \\ y \geq 0 \end{cases} \tag{15}$$

From this, we can determine the dual of the Max Flow linear program to be:

$$\text{minimize:} \tag{16}$$

$$\sum_{(u,v)\in E} c(u,v)y_{uv} \tag{17}$$

$$\text{subject to:} \tag{18}$$

$$y_{sv} + y_v \geq 1 \qquad \text{for each } (s,v) \in E \tag{19}$$

$$y_{vt} - y_v \geq 0 \qquad \text{for each } (v,t) \in E \tag{20}$$

$$y_{uv} + y_v - y_u \geq 0 \qquad \text{for each } (u,v) \in E - \{s,t\} \tag{21}$$

$$y_{uv} \geq 0 \qquad \text{for each } (v,t) \in E \tag{22}$$

If we take

$$y_{uv} = \begin{cases} 1, & \text{if } u \in S \text{ and } v \in T \\ 0, & \text{otherwise} \end{cases} \tag{23}$$

and

$$y_u = \begin{cases} 1, & \text{if } u \in S \\ 0, & \text{otherwise} \end{cases} \tag{24}$$

then we can see that the dual of the Max Flow problem is the Min Cut problem; that is, the capacity across valid cuts is minimized.

Q3

This can be achieved using dynamic programming in a manner similar to Problem 1.

```
/* Java/pseudocode for optimal shipping schedule */
StringSegment(String A){

   int n = A.length;
   int[] spacing = new int[n];
   int[] best = new int[n];
   int highest;
   ArrayList<Integer> finalSpacing = new ArrayList<Integer>();

   // Iterates through the string array, storing the optimal value in 'best' and
   // ideal segments in 'spacing'
   for (int i = 1; i < n; i++){
      highest = 0;
      for (int j = 1; j <= i; j++){
         if ( best[j-1]+quality(j...i) > best ){
            highest = best[j-1]+quality(j...i);
            spacing[i] = j-1;
         }
      best[i] = highest;
   }

   // This backtracks through to ensure proper spacing. It starts at last element in
   // string and continually 'jumps' to that location in the spacing[] array to find
   // the optimal spacing for each substring
   int i = n;
   while ( i > 0 ){
      finalSpacing.append(spacing[i]);
      i = spacing[i];
   }

   return finalSpacing;

}
```

This algorithm runs in $O(n^2)$ time because it iterates through the length of the string using nested for loops. At each step, the nested loop size increases by one, starting at one and ending at n. Thus, the runtime is given by a summation; that is, $O(\frac{n(n-1)}{2})$ which is $O(n^2)$. The algorithm produces the correct outcome because the optimal value for the prefix string is used and a new optimum is calculated to be the highest sum of previous optimum plus the quality of the suffix string. This is guaranteed to produce the highest quality score at each iteration. From this, it is clear that the correctness that this point relies on the blackbox quality() method.

Q4

Given the following "loss" matrix A for Player 1 in a game of rock-paper-scissors.:

$$A = \begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix}$$

(a) We can construct the following assuming Player 1 and Player 2 both play a mixed strategy:

$$\begin{aligned} \text{Player 1 plays rock: } EV(x_1) &= y_1(0) + y_2(1) + y_3(-1) \\ \text{Player 1 plays paper: } EV(x_2) &= y_1(-1) + y_2(0) + y_3(1) \\ \text{Player 1 plays scissors: } EV(x_3) &= y_1(1) + y_2(-1) + y_3(0) \end{aligned}$$

The expected loss then is simply the sum of the expected values given above.

$$\text{expected loss } = \sum_{i=1}^{3} EV(x_i)$$

Thus, if both players play a mixed $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ strategy, the expected loss is zero.

(b) If Player 1 plays a mixed strategy but Player 2 plays exclusively rock, paper, or scissors, then the expected loss for both players will still be zero, as each player will win, lose, and draw a third of the time. If Player 1 plays any strategy in between $\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$ and $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, then the outcome is the same. For example, if Player 1 adopts a $\frac{1}{2}, \frac{1}{4}, \frac{1}{4}$ strategy, then we see the following:

$$EV = \frac{1}{6}(1) + \frac{1}{6}(-1) + \frac{1}{6}(0) + \frac{1}{12}(1) + \frac{1}{12}(-1) + \frac{1}{12}(0) + \frac{1}{12}(1) + \frac{1}{12}(-1) + \frac{1}{12}(0) = 0$$

If both players adopt a strategy other than $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, then there is the potential for loss (i.e. Player 1 always plays rock and Player 2 always plays scissors, then Player 1 will always win).

(c) As shown in parts (a) and (b), a player minimizes loss by adopting a $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ strategy. No matter what the opposing player's strategy is, the expected loss will be zero, and that is the best one can hope for, since any normal human will try to avoid being predictable in rock-paper-scissors.

If we solve the equations given in part (a), with the added requirement then $x_1 + x_2 + x_3 = 1$ and $y_1 + y_2 + y_3 = 1$, we find that $x_1 = x_2 = x_3 = y_1 = y_2 = y_3 = \frac{1}{3}$.

(d) No. As mentioned in part (c), a $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ strategy results in an expected loss of zero. However, if Player 2 changes their strategy to something other than $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, Player 1 could alter their own strategy to improve their gain. By definition, a Nash equilibirum is a case "in which each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only their own strategy." Clearly no scenario, except when Player 1 and Player 2 both adopt the $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ strategy, can be a Nash equilibrium.