Question I:

Part (a): To begin with, we set \( x_1 = x_2 = x_3 = x_4 = \text{False} \).

Step 1: Normal clause \( C_3 \) is not satisfied. So, we must set \( x_3 = \text{True} \).

Step 2: Now, normal clause \( C_2 \) is not satisfied. So, we must set \( x_4 = \text{True} \). At this point, all normal clauses are satisfied.

Step 3: Now, since both \( x_3 \) and \( x_4 \) are \( \text{True} \), the purely negative clause \( C_5 \) is not satisfied. So, by Step 3 of the algorithm, we must conclude that there is no satisfying assignment.

Part (b): The binary tree produced by Huffman’s Algorithm and the resulting prefix code are shown below. (The total cost of the prefix code is 226.)

```
Symbol  | Prefix Code
---------|------------
a        | 000        
b        | 01         
c        | 0010       
d        | 0011       
e        | 10         
f        | 11         
```

Question II:

Part (a): Consider two items whose weights, profits and profit per unit weight ratios are shown below.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Profit</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_1 )</td>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>( I_2 )</td>
<td>7</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Let the knapsack capacity be 7. The ordering of the items by the ratio is \( \langle I_1, I_2 \rangle \).

The heuristic chooses item \( I_1 \) with weight 6. Since the knapsack capacity is 7, it cannot add \( I_2 \). So, the profit produced by the heuristic = 6.

The optimal solution is to use item \( I_2 \) whose weight = 7. The optimal profit is 7. Thus, the heuristic does not produce an optimal solution in this case.
Part (b): The statement is false.

Counter-example: Suppose we have three symbols $x$, $y$ and $z$, each with a frequency of 10. An optimal tree for these symbols produced by Huffman’s Algorithm and the resulting prefix code are shown below.

![Huffman Tree](image)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prefix Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>00</td>
</tr>
<tr>
<td>$y$</td>
<td>01</td>
</tr>
<tr>
<td>$z$</td>
<td>1</td>
</tr>
</tbody>
</table>

The frequencies of $y$ and $z$ are equal. However, the above (optimal) prefix code uses 2 bits for $y$ and only one bit for $z$.

Question III: As stated in the problem, we assume that the size of the given array is at least 2.

Part (a):

Divide Step: Divide the array $X[1..n]$ into two parts, namely the left subarray $X[1..\lfloor n/2 \rfloor]$ and the right subarray $X[\lfloor n/2 \rfloor + 1..n]$. This step runs in $O(1)$ time.

Conquer Step: Recursively find the maximum lengths of the special subarrays of the left and right subarrays. Let $\text{max}_\text{left}$ and $\text{max}_\text{right}$ denote these lengths.

Recursion ends when the subarray size is 1; that is, the subarray is of the form $X[a..a]$. In that case, the return value is 1 if $X[a]$ is a vowel and 0 otherwise.

Combine Step: The length of a longest special subarray of $X$ is the maximum of the following three values: (i) $\text{max}_\text{left}$, (ii) $\text{max}_\text{right}$ and (iii) the maximum length of a special subarray that straddles the left and right subarrays constructed above; denote this value by $\text{max}_\text{straddling}$.

The value of $\text{max}_\text{straddling}$ can be computed as follows.

1. If $X[\lfloor n/2 \rfloor]$ is not a vowel or $X[\lfloor n/2 \rfloor + 1]$ is not a vowel, then $\text{max}_\text{straddling} = 0$. (This step can be done in $O(1)$ time.) Otherwise, carry out the following three steps.

2. Find the smallest index $i$ such that the subarray $X[i..\lfloor n/2 \rfloor]$ is special. This can be done by a simple loop that successively checks whether each character from $X[\lfloor n/2 \rfloor]$ down to $X[1]$ is a vowel. Thus, this step can be done in $O(n)$ time.

3. Find the largest index $j$ such that the subarray $X[\lfloor n/2 \rfloor +1..j]$ is special. This can be done in a manner similar to that in Step 2 except that we should successively check whether each character from $X[\lfloor n/2 \rfloor + 1]$ to $X[n]$ is a vowel. Thus, this step can also be done in $O(n)$ time.

4. Set $\text{max}_\text{straddling} = j - i + 1$. (This step can be done in $O(1)$ time.)

Thus, the time used for the Divide step is $O(1)$ and that for the Combine step is $O(n)$. In other words, the total time for the Divide and Combine steps is $O(n)$. 

2
Part (b): Pseudocode based on the above discussion is provided below.

```c
int max_length_special(X, p, q) { // It is assumed that p <= q.
    if (p == q) { // Base case (to stop recursion).
        if (X[p] is a vowel)
            return 1;
        else return 0;
    }
    else { // Array has at least two elements.
        temp = (p+q)/2;  // Divide step.
        max_left = max_length_special(X, p, temp);  // Conquer step.
        max_right = max_length_special(X, temp+1, q);  // Conquer step.

        // Code for the Combine step is shown below.
        if ((X[temp] is not a vowel) or (X[temp+1] is not a vowel))
            max_straddling = 0;
        else { // There is a special subarray that straddles the two halves.
            k = temp-1;  // Index to go through the left subarray.
            while ((k >= p) and (X[k] is a vowel))
                k = k-1;
            i = k+1;  // i is the smallest index such that X[i .. temp] is special.

            k = temp+1;  // To go through the right subarray.
            while ((k <= q) and (X[k] is a vowel))
                k = k+1;
            j = k-1;  // j is the largest index such that X[temp+1 .. j] is special.

            max_straddling = j-i+1;
        }
        return max{max_left, max_right, max_straddling};
    }
} // End of max_length_special.
```

Part (c): As mentioned above, the divide and combine steps run in $O(n)$ time and the conquer step uses two recursive calls on subarrays of size at most $\left\lceil n/2 \right\rceil$. Therefore, the recurrence for the running time $T(n)$ is

$$T(n) = 2T(\left\lceil n/2 \right\rceil) + cn$$

with $T(1) = c_1$, for some constants $c$ and $c_1$. (Using Master Theorem, it can be seen that the solution to the above recurrence is $T(n) = O(n \log n)$.)

Note: The problem can be solved in $O(n)$ time by dynamic programming. This is left as an exercise for the student.
**Question IV:**

**Part (a):** $B[0] = \text{True}$ since we can realize the value 0 (using an empty set of stamps) regardless of the given denominations. Thus, $B[0]$ can be computed in $O(1)$ time.

**Part (b):** In the following discussion, we will use the convention if the value of an index $i$ for the array $B$ is negative, then the value of $B[i]$ is $\text{False}$.

Consider the computation of $B[j]$ for any $j \geq 1$. Observe that the value $j$ can be realized if and only if at least one of the values $j - d_i$, $1 \leq i \leq n$, can be realized. Thus, the equation for computing $B[j]$ is given by

$$B[j] = B[j - d_1] \lor B[j - d_2] \lor \ldots \lor B[j - d_n]$$

$$= \bigvee_{i=1}^{n} B[j - d_i]$$

(1)

Note that each of the indices used on the right side of the above equation is less than $j$. If the index $j - d_i$ is non-negative, then we have already computed the value $B[j - d_i]$; if the index is negative, then by our assumption, the value $B[j - d_i]$ is $\text{False}$. Thus, for any $j$, $0 \leq j \leq Y$, the value of $B[j]$ can be computed by taking the logical-OR of $n$ Boolean values. In other words, $B[j]$ can be computed in $O(n)$ time.

**Part (c):** There are $Y + 1$ entries in the $B$ array. As explained above, each entry can be computed in $O(n)$ time. So, the time for computing all the entries of $B$ is $O(nY)$.

**Note:** Once all the entries of $B$ have been computed, we can determine whether or not $Y$ can be realized by examining the value of $B[Y]$. 

---

4