(1) DPV Problem 6.3. Your algorithm should run in time at most O(n2) (although a better running time is also possible).

Yuckdonalds is considering opening a series of restaurants along Quaint Valley Highway(QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, $m_1, m_2, ..., m_n$. The constraints are as follows:

At each location, Yuckdonalds may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, ..., n$.

Any two restaurants should be at least k miles apart, where k is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

If we start by saying that the maximum expected profit at a location i is represented by $P(i)$. Then for $P(1)$, the profit would be equal to p_1. However if $k \leq m_i$, then:

$P(i) = max_{j<i}\{P(j) + A(m_i, m_j) \cdot p_i\}$
for which:
$A(m_i, m_j) = 1$ if $m_i - m_k \geq k$ or 0 otherwise

This is signified by the presence of two possible cases.
In the first case, there will be no restaurant opened at $P(1)$ and the profit would therfore be p_i In the second case, there must be at least one restaurant located before i and we will call the location of the restaurant from before i to be in location j. The maximum total profit at i in this case will come from the sum of the maximum total profit at j and p_i (only if $m_i - m_k \geq k$ because this will represent a new restaurant may be opened at i, otherwise it will just be the profit at j).

To find the time complexity it helps to write a pseudo code implementation for the algorithm:

```
maximum_expected_profit(array P) {
    P[1] = p_1;
    for (i = 2; i ≤ n; i++) {
        P[i] = 0;
        for (j = 1; j ≤ i − 1; j++) {
            temp = P[j] + A(m_i, m_j) · p_i;
            if (P[i] < temp) {
                P[i] = temp;
            }
        }
        if (P[i] < p_i) {
            P[i] = p_i;
        }
    }
}
```

Because of the presence of a for loop within another for loop, this makes the time complexity of the algorithm $O(n^2)$.

(2) DPV Problem 6.4.

You are given a string of n characters $s[1...n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like itwasthebestoftimes...). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function dict$(\cdot)$: for any string w,

dict(w) = true if w is a valid word, false otherwise.

(a) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to dict take unit time.

This can be done by implementing an array of booleans we will call $word[1, ..., n]$ and in which $word[i]$ = true if $s[1, ..., i]$ is a valid series of words or false otherwise. It is possible to determine if $s[1, ..., i]$ is a valid series of words by checking if there is a j such that $s[1, ..., j]$ is a valid series of words while $s[j + 1, ..., i]$ is a valid word. Thus, $word[i]$ can be represented as false when $i \leq 0$ or:
$word[i] = max_{1 \leq j < i}(word[j]$ && $DICT(s[j + 1, ..., i])$

In the form of pseudocode, this can be shown by:

```
valid_words(String s, int n) {
    array of booleans word[1, ..., n];
    for (i = 2; i ≤ n; i++) {
        word[i] = false;
        for (j = 1; j ≤ i − 1; j++) {
            if (word[j] && DICT(s[j + 1, ..., i]) {
                word[i] = true;
            }
        }
    }
}
```

The algorithm shows that $s[\cdot]$ is a sequence of valid words if $word[n] = true$ and has a time complexity of $O(n^2)$.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

This can be done by adding another array that stores ints to store each valid word and initializing this array alongside the array $word$ and storing the value of j in the new array whenever $word[i] = true$ is called. This will give the starting positions of each word in the sentence so we will call this array $start$ and these words could be output by adding a code chunk to the end of the code such as:

```
for (i = 1; i ≤ n; i++) {
    print(s[i]);
    if (i in start) {
    print(" ");
    }
}
```

This algorithm works by adding a space at the end of every valid word, splitting the string into a sentence.

(3) DPV Problem 6.8.

Given two strings $x = x_1x_2 \cdots x_n$ and $y = y_1y_2 \cdots y_m$, we wish to find the length of their longest common substring, that is, the largest k for which there are indices i and j with $x_ix_{i+1} \cdots x_{i+k-1} = y_jy_{j+1} \cdots y_{j+k-1}$. Show how to do this in time $O(mn)$.

For values of i and j such that $1 \leq i \leq x$ and $1 \leq j \leq y$, we will define the length of the matching substring in the form $L(i, j)$ with the substring terminating at x_i and y_j. The recursion would then be:

$L(i, j) = L(i - 1, j - 1) + 1$ if x_i and y_j are equal or 0 otherwise.

Additionally for all $1 \leq i \leq x$ and $1 \leq j \leq y$, $L(0, 0)$, $L(i, 0)$ and $L(0, j)$ will all be initialized to 0. The recursion algorithm works by adding 1 to the value at $L(i, j)$ if i and j are equal and adds this one to any previous parts of the substring, eventually terminating the equality which is when the next position will show 0. This makes it so that the longest substring could be found by going through all the values and finding the maximum values for $L(i, j)$. The i and j would represent the final overlapping point in the substring and the value stored inside $L(i, j)$ would represent the length of the substring.

(4) DPV Problem 6.14.

Cutting cloth. You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1, n]$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i. Assume the a_i, b_i, and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired.

For values of i and j such that $1 \leq i \leq X$ and $1 \leq j \leq Y$, we will define the maximum selling price in the form $P(i, j)$. There will be a function in the form $R(i, j)$ such that:

$R(i, j) = max_k\{c_k\}$ if there is a product k such that $a_k = i$ and $b_k = j$
otherwise, if there is no product k, it returns 0

The recursion function will have two cases, either not cutting the piece of cloth to find the cost, or cutting it, leaving two pieces, from which we must select the one that gives the maximum cost. The recursion function will therefore be in the form:

$P(i, j) = max\{R(i, j), max_{1 \leq m < i}\{P(m, j) + P(i - m, j)\}, max_{1 \leq n < j}\{P(i, n) + P(i, j - n)\}\}$

Following this, for all $1 \leq i \leq X$ and $1 \leq j \leq Y$, values must be initialized such that:

$P(i, 1) = R(i, 1)$
$P(1, j) = R(1, j)$

Following this, the final solution can be represented by the value of $P(X, Y)$.