6.1 We define the subproblem \( S(i) \) to be the largest summation that terminates exactly at position \( i \). Let the input array be \( a[1..n] \).

The recurrence relation on \( S(i) \) is:

\[
S(i) = \max\{S(i - 1) + a[i], a[i]\}.
\]

To show this is true, we consider two cases: (1) the max sum contains at least one previous number and (2) the max sum does not contain previous numbers. Apparently for case (1), the largest sum you can have is \( S(i - 1) + a[i] \) and for case (2) the sum is \( a[i] \). We are all set.

We use dynamic programming tabular to solve the problem. The pseudocode is:

```python
function max_subsequence(a)
    create an array S[0..n]
    create an index array I[0..n]
    s[0] = 0
    I[0] = 0
    for i = 1 to n
        s[i] = max{ s[i-1] + a[i], a[i] }
        if s[i] == a[i]
            I[i] = i
        else
            I[i] = i - 1
    t = max{s[i], i = 1..n}
    j = k = argmax{s}
    while I[j] != j
        j = j - 1
    return the max sum t and the interval [j,k]
```

In the above code, we not only compute the max sum but also the interval that gives the result. The algorithm has a complexity \( O(n) \).

6.2 We define the sub-problem \( T(i) \) as the optimal cost of the trip that terminates at stop \( i \). Note that we have to stop at the “terminal” stop.

We have a recurrence relation for \( T \):

\[
T(i) = \max\{T(k) + (a_i - a_k - 200)^2 : k = 1..n - 1\}.
\]

The original problem corresponds to \( T(n) \). It is straightforward to use dynamic programming to solve this problem. The base problem is \( T(0) = 0 \). The complexity of this method is \( O(n^2) \).
6.3 We define the sub-problem as \( M(i) \) which is the optimal profit if we put the last restaurant at location \( i \), i.e., there will be no restaurants from location \( i + 1 \) to \( n \).

The recurrence relation on \( M \) is:

\[
M(i) = \max\{p(i) + M(j) : j = 1..i - 1 \text{ and } m_i - m_j > k\}
\]

The base problem \( M(1) = p(1) \). We can solve the problem using dynamic programming. The complexity is \( O(n^2) \).

6.4 Let \( T(i) \) be a sub-problem that detects whether the prefix string \( s[1...i] \) is composed of valid words. The recurrence relation on \( T(i) \) is

\[
T(i) = \lor\{\text{dict}(s[j+1...i]) \land T(j) : j = 1..i - 1\}
\]

which means \( T(i) \) is true if there is \( j \) so that the sequence \( s[j+1...i] \) is a word and \( T(j) \) is composed of valid words; otherwise it is false. The base problem \( T(0) = \text{true} \). A dynamic programming can be used to solve the problem with a complexity \( O(n^2) \).

To recover the words in the sequence, we need to keep index \( j \) that makes \( T(i) \) true. We can then do back tracking and stripe of word by word from the sequence \( s[1..n] \) based on the stored indexes.

6.5 (a) Each column can have 8 patterns: one no-tile pattern, 4 one-tile patterns and 3 two-tile patterns.

(b) We define the sub-problem as \( S(i, j) \), which is the max-summation from column 1 to \( i \) and the last column pattern is \( j \). The recurrence relation is:

\[
S(i, j) = \max\{S(i - 1, m) + v(i, j) : m \text{ is compatible with } j\}
\]

where \( v(i, j) \) is the covered total value on the column \( i \) by pattern \( j \). This can be solved by dynamic programming with a complexity \( O(n) \).

6.17 The subprogram is \( C(v) \), which is true if \( v \) can be represented as the sum of values from \( x_1 \), \( x_2 \), ..., \( x_n \), and otherwise is false.

We have recurrence relation on \( C \):

\[
C(v) = \lor\{C(v - x_i) : i = 1..n\}
\]

We can write a table and use dynamic programming to solve the problem. The base problem \( C(0) = \text{true} \) and \( C(v) = \text{false} \), if \( v < 0 \). The complexity is \( O(nv) \).

6.20 Given a sequence of words that are sorted by their frequencies, the tree can be constructed by selecting a word as the root, its left subsequence to construct the left sub-tree, and its right sequence to construct the right sub-tree. Apparently, this will result in valid binary search trees. Note that the optimal binary search tree has the sub-optimal structure: for an optimal tree, the left and right sub-trees are also optimal. This can be proved by contradiction: if the sub-trees are not optimal, we can find better sub-trees so that the whole tree has lower average search times, which contradicts to the optimal assumption.
We define the sub-problem as $T(i, j)$, which gives the optimal binary tree and the weighted search times for the input sub-sequence from word $i$ to $j$. The recurrent relation is

$$T(i, j) = \min\{P(i, k - 1) + Q(k + 1, j) + p_k\}$$

$$P(i, k - 1) = T(i, k - 1) + \sum_{l=i}^{k-1} p_l, \text{ if } k - 1 \ge i, \text{ else } 0$$

$$Q(k + 1, j) = T(k + 1, j) + \sum_{l=k+1}^{j} p_l, \text{ if } k + 1 \le j, \text{ else } 0$$

The base problems $T(i, i) = p_i$. We can write a 2D table and use bottom-up dynamic programming to fill it up. The sequence to fill up the table is similar to optimizing matrix multiplications, i.e., line by line parallel to the diagonal lines until you hit $T(1, n)$ at the corner, which is the target. How can you build the binary search tree based on the DP result? Think about it.

6.21 Let $C(u)$ be the optimal vertex cover of the sub-tree rooted at vertex $u$. The recurrence relation on $C$ is:

$$C(u) = \max\{1 + \sum_{\text{Grand-children } v \text{ of } u} C(v), \sum_{\text{Children } w \text{ of } u} C(w)\}$$

For empty trees, $C = 0$. The DP starts from the leave nodes and propagates to the root. The complexity is $O(V)$. 