1. DPV Problem 6.4

(a) We first create an array a[boolean] that has a size of the input array s.length. The values of the array are defined as:

```
a[i]: True if a[0…i] is in the dictionary      // dict(a[0…i]) == true
      False otherwise
```

<pseudocode>

```
for i = 0; i < s.length; i++
        if dict(s[0…i]) is true, then set a[i] = true
        if a[i] == true, for j = i + 1; j < s.length; j++
                if dict(s[i+1…j] == true)
                        a[j] == true

if a[s.length-1] == true, then string s is a sequence of valid words
```

From 0 to end of the array, i is the first (leading) index of the outer loop. If s[0…i] is present in the dictionary, then a[] stores the Boolean values. If the "prefix" from 0 to i is a valid word, then index j is introduced in the inner loop. From i+1 to s.length-1, check from i+1 to j is a valid word. This introduces the dynamic aspect of the algorithm. If another word exists in a[i+1…j], then we iterate the process until the end of the string. Once the for-loops are done running, check if the last element of array a is true. If true, then it means that up to the last word examined, it was a valid word in the dictionary.

Recurring subproblem: $\text{dict}[j] = \text{false}$ if $j <= 0$ (base case)

or

$\max_{1<=i<=j} (\textit{dict}[i] \ \&\& \ \textit{dict}(s[i+1\ldots j]))$

Time complexity will be O(n^2) and Ω(nlogn). The solution above incurs a O(n^2) complexity because two for-loops run through the array twice with i and j (given that dict() incurs unit time).

(b) So, I can make use of a tree of strings. For every prefix that is found, we store it in a tree, and the tree will have multiple branches as the recurrent execution expands out. For example, in the string s[0 … i … j … n-1], if dict(0…i) was true and dict(i+1…j) was also true, then we store the two strings s[0…i] and s[i+1…j] into the tree. Once the two for-loops are done running, then whichever branch of execution that returned true can output the proper sentences from the words stored in the tree.

```
                              String s[0]
                                /     \
                               /       \
                 s[0…1] is in dict       s[0…3] is in dict
                   /       \                  /       \
                  /         \                /         \
S[1…4] is in dict   s[1…6] is in dict   s[3…6] is in dict   s[3…8] is in dict
                                    .
                                    .
```

2. DPV Problem 6.8

Instead of using confusing indices, I will provide an intuitive explanation. Let two strings m & n be represented as (mmmmm) and (nnnn). Each m and n are arbitrary characters.
Variable int longest_substring is initially at 0.

```
        (m   m   m   m   m)
(n   n   n   n)
```

You first compare the first character m and the last character n. If they are the same character, then longest_substring == 1.

```
        (m   m   m   m   m)
   (n    n    n    n)
```

Then, we compare two characters of m and two characters of n. If both pairs of m and n match, then longest_substring == 2. Otherwise, longest_substring remains as the previous value 1.

```
        (m   m   m   m   m)
             (n   n   n   n)
```

These recurrent steps repeat until the last character of m is compared with the last character of n. Finally, we returned what longest_substring is, which will give us the length of the longest possible substring.

Each time the comparison takes place between two strings, there are at maximum n number of character comparisons to do because the smaller string has only n characters. The smaller string n shifts underneath m for m times, starting from left to right until the end of string m. Therefore, the runtime is O(m * n).

3. DPV Problem 6.13

For this problem, assume that the cards have values in the range [1, 13], with an Ace counting as 1, a Jack counting as 11, a Queen counting as 12, and a King counting as 13.

(a) Counterexample: [1, 2, 5, 3]
If both players play in a greedy fashion, player A grabs 3. Then B takes 5, followed by A taking 2. A will have 5, and B will have 6. So, grabbing 3 as the first move was a wrong one. This shows how the greedy approach is only sub-optimal.

(b) Let a function mv(i, j) represent the Maximum Value we can collect from the cards if we had cards i through j left. Also, let an array A[1…n] hold the values of each card dealt. Then we can define a recurrence relation as

$$\mathrm{mv}(i, j) = \begin{cases} 1.\ 0 & \text{if } i>j \\ 2.\ A[i] & \text{if } i=j \\ 3.\ \max \begin{cases} A[i] + \mathbf{mv}(i + 1 + [A[i+1] > A[j]],\ j - [A[j] >= A[i+1]]) \\ A[j] + \mathbf{mv}(i + [A[i] > A[j-1]],\ j - 1 - [A[j] >= A[i]]) \end{cases} \end{cases}$$

If i>j, then there are no valid cards between them. If i=j, then both are pointing at one card, so take that last card and the game is terminated. As either i increments or j decrements, meaning that the other player is taking cards from the either end, we return a new mv() on (j- i + 1) by induction.

Throughout the iterative steps, the range of i and j are: 1 <= i <= n+1 && 0 <= j <= n. Then, we can store the values from mv() into a 2D-array where the results from the subproblems are stored in appropriate indices. One loop will increment i for every iteration, and j will decrement for every iteration, so the entire runtime is n * n * (constant look-up time), which is O(n^2).

4. DPV Problem 6.22

As a hint, translate this problem into the Knapsack problem without repetition from Lab 5.

t is the desired sum we are trying to get. We use t has a variable to keep track of the sum we are trying to get at the moment as the program runs dynamically. A is the array of numbers that we are using. Let's define a function SubsetOf(A, n, t). Then the base case and the recurrent relation are the following:

base case: SubsetOf(A, n, t) == **false** if n == 0 and t > 0

true if t == 0

recurrent case: SubsetOf(A, n, t) = SubsetOf(A, n-1, t) **||** SubsetOf(A, n-1, t – A[n-1])

When the case is recurrent, the SubsetOf() function calls back itself with two different cases.

SubsetOf(A, n-1, t) is the case where A[n] cannot be an element in the subset either because subtracting A[n] from t makes t negative, or because along down the computational path, it is found that A[n] cannot be in the subset whose sum is t.

SubsetOf(A, n-1, t – A[n-1]) is the more common case. The function is called with such parameters when t-A[n-1] is positive. This means that there is a chance that A[n] is in the subset that makes up t and therefore we access whether t-A[n] can be given from some arbitrary elements in A[1…n-1].

Finally, if there is any branch of execution where all the SubsetOf() function-calls returns **true** down the line of computation, then the initial SubsetOf() call also returns true, and the program terminates.

In terms of runtime, we run SubsetOf(A, n, t) for n times, and there is at max t row-wise operations for every i incrementing. This yields O(n * t) = O(nt).