Lab assignment solution

Problem 1:

We say subproblems of Binary search do not overlap because no two subproblems are the same or have a common element. For example, checking if array = [1,3,6,9,23,76,99] contains 5. The sub problems in this case are finding the target integer in [1,3,6,9] and in [23,76,99]. The two subproblems or their sub-subproblems will not have a common sub array. Hence, they don't overlap.

On the other hand, for Fibonacci series, their two sub-subproblems may come out to be the same. For example: f(4) = f(3)+f(2) = [f(2)+f(1)]+f[2], we see that the sub problem f(3) computes f(2), and f(2) is also computed separately. Which is overlap.

Problem 2:

D	""	"k"	"ka"	"kal"	"kale"
""	0	1	2	3	4
"m"	1	1	2	3	4
"ma"	2	2	**1**	2	3
"map"	3	3	2	2	3
"mapl"	4	4	3	**2**	3
"maple"	5	5	4	3	**2**

Problem 3:

Here are pseudocodes for longestCommonSubsequence. The first one a recursive and the second one with dynamic programming.

```
// longestCommonSubsequence recursive
Algorithm lss(s,t)
    Input: String s and t
    Output: the longest subsequenence of letters common to s and t
        // base cases
    if(s.length()*t.length() = 0 )
        return 0
    // if the last letter in t and in s is the same
    sp ← s.withoutLastCharacter()
    tp ← t.withoutLastCharacter()
    if(s.getLast() != t.getLast())
        return max( lss(s, tp), lss(sp, t) )
    return 1 + lss(sp, tp)
```

```
// longestCommonSubsequence dynamic programming
table ← new HashTable()
Algorithm lss(s,t)
    Input: String s and t
    Output: the longest subsequence of letters common to s and t
    key ← [s,t]
    value ← table.get(key)
    if(value != NULL)
        return value
    // base cases
    if(s.length()*t.length() = 0)
        table.put(key, 0)
        return 0
    sp ← s.withoutLastCharacter()
    tp ← t.withoutLastCharacter()
    // if the last letter in t and in s is the same
    if(s.getLast() != t.getLast())
        value ← max( lss(s, tp), lss(sp, t) )
        table.put(key, value))
        return value
    // otherwise
    value ← 1+lss(sp,tp)
    table.put(key, value)
    return value
```

Problem 4:

Pseudocode with and without storage.

```
// LeastPerfectSquareSumCount recursive without cacheing
Algorithm lpssc(n)
    Input: integer n
    Output: the least number of perfect squares whose sum is n
    if(n<=0)
        return n
    i ← 0
    min ← n     // in case the only perfect squares that add up to n is 1+1+...+1
    while(i*i <= n)
        c ← lpssc(n-i*i)+1
        if(c < min)
            min ← c
        i++
    return min
```

```
// LeastPerfectSquareSumCount dynamic programming
table = new HashTable()
Algorithm lpssc(n)
    Input: integer n
    Output: the least number of perfect squares whose sum is n
    // check if value was already computed
    if(table.containsKey(n))
        return table.get(n)
    // base case
    if(n<=0)
        table.put(n,n)
        return n
    i ← 0
    min ← n     // in case the only perfect squares that add up to n is 1+1+...+1
    while(i*i <= n)
        c ← lpssc(n-i*i)+1
        if(c < min)
            min ← c
        i++
    table.put(n,min)
    return min
```