# Answer of 1

In binary search, the sub-problem is searching for X in the search space of [ lower. upper]. The problem is non-overlapping because we make the decision to search in either [lower, mid - 1] search space or [mid+1, upper] search space which will never repeat.

In fib(n), the sub-problem is calculating the fib(n-1) and fib(n-2) as fib(n) = fib(n-1) + fib(n-2). This results in multiple overlapping calls as for fib(n) we need to find fib(n-1) and fib(n-2). Similarly, for fib(n-1), we need to calculate fib(n-2) and fib(n-3). Already we are calculating fib(n-2) twice. This is the overlapping subproblem.

# Answer of 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 |

# Answer of 3

class Solution {

public int climbStairs(int n) {

if (n == 0 || n == 1) {

return 1;

}

int prev = 1;

int current = 1;

for (int i = 2; i <= n; i++) {

int next = prev + current;

prev = current;

current = next;

}

return current;

}

}

# Answer of 4

class Solution {

public int longestCommonSubsequence(String text1, String text2) {

int[][] cache = new int[text1.length() + 1][text2.length() + 1];

for (int i = 0; i < cache.length; i++) {

cache[i][0] = 0;

}

for (int j = 0; j < cache[0].length; j++) {

cache[0][j] = 0;

}

for (int i = 1; i < cache.length; i++) {

for (int j = 1; j < cache[i].length; j++) {

if (text1.charAt(i - 1) == text2.charAt(j - 1)) {

cache[i][j] = cache[i - 1][j - 1] + 1;

} else {

cache[i][j] = Math.max(cache[i - 1][j], cache[i][j - 1]);

}

}

}

return cache[text1.length()][text2.length()];

}

}