

# 02635 Fall 2016 — Module 11 (solutions)

## Exercises

---

1. Let  $f_n$  denote the number of function calls required to compute  $F_n$  recursively, i.e.,

$$f_n = f_{n-1} + f_{n-2} + 1$$

and  $f_0 = f_1 = f_2 = 1$ . The number of function calls required to compute the 5th Fibonacci number  $F_5$  is shown in the following table:

$n$	$F_n$ ( $n$ th Fibonacci number)	$f_n$ (no. function calls)	$2^n$	$2^{n/2}$
0	0	1	1	1
1	1	1	2	1.414
2	1	3	4	2
3	2	5	8	2.828
4	3	9	16	4
5	5	15	32	5.657
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
k	$F_{k-1} + F_{k-2}$	$f_{k-1} + f_{k-2} + 1$	$2^k$	$2^{k/2}$

The number of function calls can be verified with the following program:

```

#include <stdlib.h>
#include <stdio.h>

int ncalls = 0;

unsigned long fibonacci(unsigned long n) {
    ncalls+=1;
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}

int main(int argc, const char * argv[]) {

    unsigned long n=0;
    if (argc != 2) {
        fprintf(stdout, "Usage: %s n\n", argv[0]);
        exit(EXIT_SUCCESS);
    }
    n = atoi(argv[1]);
    fprintf(stdout, "fibonacci(%lu) = %lu\n", n, fibonacci(n));
    fprintf(stdout, "ncalls = %d\n", ncalls);

    return EXIT_SUCCESS;
}

```

## 2. What is the time complexity of this recursive implementation?

The time complexity can be expressed in terms of function evaluations since the time required for each function evaluation (excluding the time spent on other function calls) can be upper bounded by a constant. It follows from the table and the recursive expression  $f_n = f_{n-1} + f_{n-2} + 1$  that  $f_n > f_{n-1}$  for  $n \geq 2$ , and hence

$$f_n = f_{n-1} + f_{n-2} + 1 \leq 2f_{n-1}, \quad n \geq 3.$$

This implies that the number of function calls is upper bounded by  $2^n$  (since  $f_n$  is at most twice as large as  $f_{n-1}$ ), or equivalently,  $f_n = O(2^n)$  which implies that **the cost is (at most) exponential.**

**Remark:** It is also possible to obtain an exponential lower bound on the number of function evaluations. Indeed, the inequality  $f_n > f_{n-1}$  also implies that

$$f_n = f_{n-1} + f_{n-2} + 1 \geq 2f_{n-2}, \quad n \geq 3,$$

which means that the number of function calls *at least* doubles when increasing  $n$  by two. Thus, as can be verified from the table, the exponential sequence  $2^{n/2}$  is a lower-bound for the sequence  $f_n$ .

### 3. What is the space complexity of the recursive implementation of the Fibonacci function?

Although the time complexity is exponential, the space complexity is only *linear*. This is because the *depth* of the recursion is linear. This can be verified with the following program:

```
#include <stdlib.h>
#include <stdio.h>

int size = 0;    /* number of fib. calls on stack */
int max_size = 0;

unsigned long fibonacci(unsigned long n) {
    unsigned long res;
    size += 1;
    max_size = (max_size >= size ? max_size : size);
    if ( n == 0 ) { res = 0; }
    else if ( n == 1 ) { res = 1; }
    else { res = fibonacci(n-1) + fibonacci(n-2); }
    size -= 1;
    return res;
}

int main(int argc, char * argv[]) {

    unsigned long n;
    if ( argc != 2 ) {
        fprintf(stderr, "Usage: %s n\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    n = atoi(argv[1]);
    printf("fibonacci(%lu) = %lu\n", n, fibonacci(n));
    printf("Max. calls on stack: %d\n", max_size);

    return 0;
}
```

Compile and run the program for different values of  $n$ . It is easy to verify that the maximum number of calls on the stack is equal to  $n$ .

4. Rewrite the Fibonacci function so that you avoid recursive function calls. The return type should be an unsigned long. What is the time complexity of your implementation? What is the space complexity?

The recursive function calls can be avoided by computing the Fibonacci numbers sequentially:

```
unsigned long fibonacci(unsigned long n) {  
  
    int i;  
    unsigned long f, fm = 1, fmm = 0;  
    if ( n == 0 ) return 0;  
    else if ( n == 1 || n == 2 ) return 1;  
    for (i=2; i<=n; i++) {  
        f = fm + fmm;  
        fmm = fm;  
        fm = f;  
    }  
    return f;  
}
```

This implementation has time complexity  $O(n)$  (i.e., linear time complexity; this is because of the loop) and the space complexity is  $O(1)$  (it does not depend on  $n$ ).

It is of course also possible to use the definition

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

to compute the  $n$ th Fibonacci number without the recursion. The complexity of this approach is  $O(\log(n))$  since evaluating the term  $\varphi^n$  has complexity  $O(\log(n))$ .

5. Find the largest Fibonacci number that can be represented as an unsigned long?

The largest unsigned integer with  $b$  bits is  $2^b - 1$ . If we approximate the  $n$ th Fibonacci number by

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}} \approx \frac{\varphi^n}{\sqrt{5}},$$

we obtain the following (approximate) equation

$$\frac{\varphi^n}{\sqrt{5}} = 2^b.$$

Solving for  $n$  and rounding down yields

$$n \approx \left\lfloor \frac{b \log(2) + \log(5)/2}{\log(\varphi)} \right\rfloor.$$

Thus, on systems where an `unsigned long` consists of 4 bytes (32 bits), we obtain  $n \approx 47$ , and on systems where an `unsigned long` is 8 bytes, we get  $n \approx 93$ . Indeed, the 93rd Fibonacci number is 12,200,160,415,121,876,738 whereas the 94th Fibonacci number is 19,740,274,219,868,223,167 which is larger than  $2^{64} - 1 = 18,446,744,073,709,551,615$ . This can be verified with the program in exercise 4.