Tutorial Algorithms and complexity: solutions

Solutions

1. See figure with Solution 1, File Flowchart.

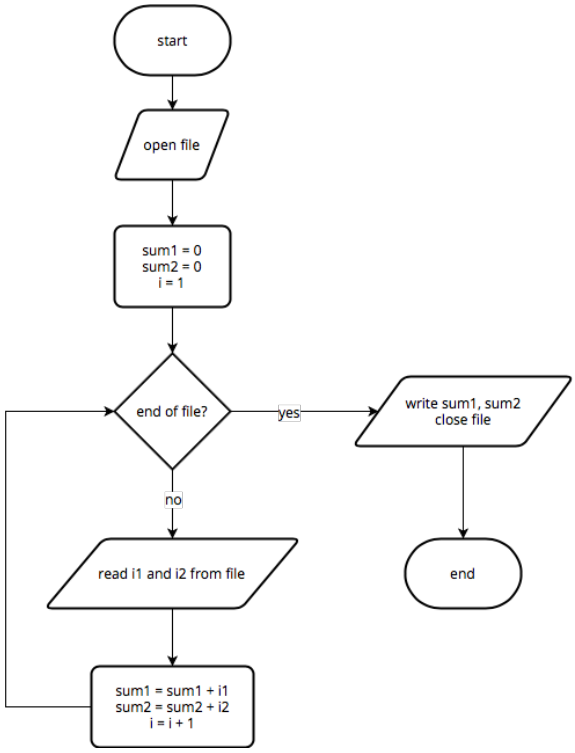

Figure 1: Solution 1, File Flowchart

2.

```
1  start
2      read n
3      read list x1, x2, x3 .... xn.
4      min = x1
5      for i = 2 to n by 1 do
6          if xi < min
7              min = xi
8          end if
9      end for
10     write min
11 end
```

If the condition less than $<$ inside of the enumeration loop (line 6) is replaced by greater than

$>$, the algorithm will find the largest number (min becomes max). This can also be solved using pre-test and post-test loops.

3. This is a well-known unsolved problem in mathematics, known as the $3n + 1$ conjecture or the Collatz conjecture. At this point in time it has not yet been proven that it will halt for all positive numbers, but a counterexample has not been found either. As of April 2019 no numbers under 10^{17} have been found that do not end up at 1.

   You might like to write this up in a programming language of your choice. Project Euler #14 (https://projecteuler.net/problem=14) is about this problem.

4.

a) $O(N)$ as the runtime will depend on the value of m; as m increases, the runtime will increase linearly.

b) $O(N^2)$ as the snippet contains two nested loops of size m. As m increases, the runtime will increase quadratically.

c) $O(N^3)$ as the snippet contains three nested loops. As m increases, the runtime will increase cubically.

d) $O(N)$ as the runtime will depend on the value of m again; note the loops are not nested and will be run 2m times. This could be refactored to move the calculation of product and sum together.

5. Take the file open and the assignment to variables as one step each = 4 steps.

   Within the loop, there is checking for the end of the file (1), reading two values (2), and doing three additions and assignments (6). This loop repeats for each line in the file, so with n lines = 6n steps.

   After the loop ends, we have two writes and the closing of the file = 3 steps.

   In total, we have 7 + 6n steps as an approximation. If the file has 10 rows, then around 67 steps will be taken; with 20 rows, around 127 steps, which is roughly twice as long compared to 10 rows. With 100 rows, it will take around 607 steps, which is roughly 10 times as long as 10 rows. So the performance of this code can be described as linearly increasing with the number of rows in the file.

6. For binary search, the best case is when the item sought is exactly in the middle of the list (or at position $\left\lceil \frac{n}{2} \right\rceil$ where n is the length of the list). The worst case is when it is in a position that requires $\log_2 n$ attempts to find.

   Given the list 1 3 5 8 13 17 21, searching for 8 is the best case, while 13 is an example of worst case.

7.

a) For 10 items, if an algorithm with the upper bound $O(n)$ takes 10^{-5}s to execute, the $O(n^2)$ algorithm will take 10^{-4}s, the $O(n^3)$ algorithm will take 10^{-3}s, the $O(\log n)$ algorithm 4×10^{-6}s and the $O(2^n)$ algorithm 0.001s. For such small input there is not much difference.

b) For 100 items, the $O(n^2)$ algorithm will take 0.01s, the $O(n^3)$ algorithm will take 1s, the $O(\log n)$ algorithm 7×10^{-6}s and the $O(2^n)$ algorithm 4×10^{16} years. That escalated quickly.

c) For 1000 items, the $O(n^2)$ algorithm will take 1s, the $O(n^3)$ algorithm will take 16.7 minutes, the $O(\log n)$ algorithm 10^{-5}s and the $O(2^n)$ algorithm 3×10^{287} years.

8.

a)

```
function times2(n):
    if n == 1 then
        return 2
    else
        return 2 + times2(n-1)
    end if
end
```

b) If this is tricky, you can start with adding all values to n and then refactor (this means to adapt).

```
function sum_all(n):
   if n == 0 then
      return 0
   else
      return n + sum_all(n-1)
   end if
end
```

Adapt this to sum even numbers only:

```
function sum_even(n):
   if n == 0 then
      return 0
   else
      if n is even then
         return n + sum_even(n-1)
      else
         return sum_even(n-1)
   end if
end
```

This could be simplified further to jump two steps:

```
function sum_even(n):
   if n == 0 then
      return 0
   else
      if n is even then
         return n + sum_even(n-2)
      else
         return sum_even(n-1)
   end if
end
```

c) In this case the base case is C(k,0) or C(k,k), the edges of Pascal's triangle. We know these two coefficients are always 1.

```
function choose(n,r):
    if r == 0 or r == n then
        return 1
    else
        return choose(n-1, r-1) + choose(n-1, r)
    end if
end
```

Extension hints

9. For this, think about the calls that are being made in the recursive version – write them down like the stack we saw in the lecture. You should see that there is a lot of repetition (that is, the same fibonacci number being calculated over and over again); can you think of a way around this?