Instructions: For each problem description, show, in outline form, how the problem can be fully factored and give an algorithm to solve the problem based on your factoring. There may be multiple solutions, and you may not need all of the variables provided.

0. Example problem description for "Function Maximum over Bounded Integers": `lower_bound` and `upper_bound` each hold an integer input. The algorithm should compute the maximum value of a certain function f over the integers between `lower_bound`, inclusive, and `upper_bound`, exclusive, and store it in `maximum`. `current` is available as temporary storage.

Example solution:

- Finding the maximum of f over the integers in [`lower_bound`, `upper_bound`)
  Splits into quotient subproblems differentiated by `current`*:*
  - Setting `current` to `lower_bound` (initiation)
  - Setting `current` to `current` + 1 (consecution)
  - Checking that `current` is less than `upper_bound` (termination)
  - Finding the maximum of zero numbers (base case)
  - Finding the maximum of f(`lower_bound`) through f(`current`) (action)
    by finding the maximum of f(`current`) and the previous result
    Splits into alternative subproblems:
    * Setting `maximum` to f(`current`) when f(`current`) is greater than `maximum`
    * Leaving `maximum` unchanged otherwise

```
let maximum ← −∞
let current ← lower_bound
while current < upper_bound do
|  if f(current) > maximum then
|  |  let maximum ← f(current)
|  end
|  let current ← current + 1
end
```

1. Problem description for "Odds": `limit` is an input; `current` is available for internal computations. The algorithm should print all positive odd numbers that are less than `limit`.

> *When displaying tables with long rows, a common technique for improving readability is to have rows alternate colors. If the user applies such formatting to a table, the computer must change the color of every odd row but leave the even rows untouched (or vice versa).*

2. Problem description for "Bounded Function Caching": `limit` is an input; `input` and `output` are available for internal computations. The algorithm should, for each integer from zero to `limit` − 1, print out the value of a certain function f applied to that integer.

> *When speed is an issue, a common algorithmic technique is to precompute important functions over likely inputs (table lookup is fast on most computers, often faster than consulting the function directly). For instance, precomputed radiance transfer, a 3D graphics algorithm that simulates realistic diffuse lighting, involves light transfer functions, whose inputs are points on an object surface. Only by pretabulating the functions' outputs on key points can the algorithm keep up with real-time rendering.*

3. Problem description for "Greedy Bin Packing": `limit` is an input; `count` and `total` are available for internal computations. A certain positive-valued function f is known ahead-of-time. The algorithm should compute a value for `count` such that $f(0) + f(1) + \cdots + f(\texttt{count} - 1)$ is less than `limit`, but $f(0) + f(1) + \cdots + f(\texttt{count} - 1) + f(\texttt{count})$ is greater than or equal to `limit`.

> *Packing algorithms are used extensively in logistics. For instance, if* `limit` *in the above problem is the weight capacity of a truck, and $f(i)$ gives the weight of the i^{th} item to be loaded, the algorithm will compute how many of the upcoming items can be loaded without exceeding the weight limit.*

4. Problem description for "Factorial": `number` is an input; `factorial` should, after the algorithm runs, hold `number` factorial. `current` is available for internal computations.

> *Factorials and related functions are used extensively in probability and statistics. Perhaps most importantly, much of modern scientific research relies on factorials to quantify the significance of its findings.*

5. Problem description for "Integer Factors": `number` is an input; the algorithm should print out all factors of `number`. `candidate` is available for internal computations.

(Note: the notation $x \,\%\, y$ means the remainder after dividing x by y. To test whether some number y is a factor of some other number x, check that the remainder is zero.)

> *Almost all of the algorithms that compress or decompress sound, images, or video depend on a family of algorithms call fast Fourier transforms. The fastest fast Fourier transforms, in turn, factor the length of their data and then solve parallel subproblems, one for each of the prime integer factors. Chances are that any computer presenting web-based multimedia is solving an integer factoring problem.*

6. Problem description for "Collatz Stopping Time": `number` is a positive input. The algorithm should set `count` to the minimum number of times that the hailstone function must be applied to `number` to reach a value of one. `current` is available as temporary storage.

For reference, the hailstone function:

$$h(x) = \begin{cases} x/2 & \text{if } x \equiv 0 \pmod 2 \\ 3x + 1 & \text{if } x \equiv 1 \pmod 2 \end{cases}$$

For example, if `number` is 3, `count` should end up being 7, as there are seven steps to get from 3 to 1:

$$h(3) = 10;\ h(10) = 5;\ h(5) = 16;\ h(16) = 8;\ h(8) = 4;\ h(4) = 2;\ h(2) = 1$$

> *In business and physics simulations (including those in games), a more complicated function is substituted for h, each application of which advances the simulation by one unit of time. Likewise, the termination predicate is altered to detect some event of interest, a profit goal or a collision perhaps. The result, then, is the in-simulation time elapsed until the event.*

