# Week 4: Algorithmic efficiency

By now, we have a good idea of what it means for a solution to be an algorithm—it solves the problem, and it actually finishes if we start it on a problem. We haven't worried about the efficiency of algorithms, though. There isn't much difference between an algorithm that never finishes and one that finishes billions of years in the future, so perhaps that was an oversight. We should amend that now.

## Exercises

### Counting primitive operations

**Exercise:** Consider this way of computing the mean of a sequence of numbers:

```
accumulator = 0
for n in numbers:
 accumulator += n
mean = accumulator / len(numbers)
```

Count how many primitive operations it takes. To do it correctly you need to distinguish between updating a variable and assigning a value to a new one. Updating the accumulator `accumulator += n` usually maps to a single operation on a CPU because it involves changing the value of a number that is most likely in a register. Assigning to a new variable, as in

```
mean = accumulator / len(numbers)
```

doesn't update `accumulator` with a new value, rather it needs to compute a division, which is one operation (and it needs `len(numbers)` before it can do this, which is another operation), and then write the result in a new variable, which is an additional operation.

**Exercise:** Consider this alternative algorithm for computing the mean of a sequence of numbers:

```
accumulator = 0
length = 0
for n in numbers:
 accumulator += n
 length += 1
mean = accumulator / length
```

How many operations are needed here? Is it more or less efficient than the previous algorithm?

### Guessing-game complexity

Recall the guessing game from the previous chapter, where one player thinks a number between 1 and 20 and the other has to guess it. We had three strategies

for the guesser:

1. Start with one. If it isn't the right number, it has to be too low—there are no smaller numbers the right one could be. So if it isn't one, you guess it is two. If it isn't, you have once again guessed too low, so now you try three. You continue by incrementing your guess by one until you get the right answer.
2. Alternatively, you start at 20. If the right number is 20, great, you got it in one guess, but if it is not, your guess must be too high—it cannot possibly be too small. So you try 19 instead, and this time you work your way down until you get the right answer.
3. The third strategy is this: you start by guessing ten. If this is correct, you are done, if it is too high, you know the real number must be in the range [1, 9], and if the guess is too low, you know the right answer must be in the range [11, 20]—so for your next guess, you pick the middle of the range it must be. With each new guess, you update the interval where the real number can be hidden and pick the middle of the new range.

**Exercise:** Identify the best- and worst-case scenarios for each strategy and derive the best-case and worst-case time usage.

### Function growth

Consider the classes

1. $O(\log n)$, $o(\log n)$, $\Omega(\log n)$ and $\omega(\log n)$
2. $O(n)$, $o(n)$, $\Omega(n)$, and $\omega(n)$
3. $O(n^2)$, $o(n^2)$, $\Omega(n^2)$, and $\omega(n^2)$
4. $O(2^n)$, $o(2^n)$, $\Omega(2^n)$, $\omega(2^n)$

**Exercise:** For each of the functions below, determine which of the 16 classes it belongs in. Remember that the complexity classes overlap, so for example, if $f \in o(g)$ then $f \in O(g)$ as well, and if $f \in \Theta(g)$ then $f \in O(g)$ as well as $\Omega(g)$ (but $f \notin o(g)$ and $f \notin \omega(g)$).

1. $f(n) = 23 \times n$
2. $f(n) = 42 \times n^2 - 100 \times n$
3. $f(n) = n/\log(n)$
4. $f(n) = \log(n)/n$
5. $f(n) = n^2/\log(n)$
6. $f(n) = \log(n) + \log(n)/n$
7. $f(n) = 5^n - n^3$
8. $f(n) = n!$
9. $f(n) = 2^n/n$
10. $f(n) = \log(\log(n))$

### Sieve of Eratosthenes

Recall the Sieve of Eratosthenes from the previous chapter.

**Exercise:** Derive an upper bound for its running time.

**Exercise:** Is there a difference between its best-case and worst-case running time?

### Merging

Recall the *merging* algorithm from the previous chapter.

**Exercise:** Show that you can merge two sorted lists of size n and m, respectively, into one sorted list containing the elements from the two, in time $O(n + m)$.