Week 3: Introduction to algorithms

This week we learn what the building blocks of algorithms are, we learn how to reason about algorithms and their correctness, and we get started on the first mandatory project (of which you can choose between two).

Exercises

In the exercises this week there are both algorithmic questions and programming questions. You do not need to implement any of the algorithms—it is more important that you manage to design the algorithms themselves—but I strongly encourage you to implement what you can, since you will learn a lot more from trying to tell the computer what your ideas are, than just telling yourself of a study-mate.

Merging

Assume you have two sorted lists, `x` and `y`, and you want to combine them into a new sequence, `z`, that contains all the elements from `x` and all the elements from `y`, in sorted order. You can create `z` by *merging* `x` and `y` as follows: have an index, `i`, into `x` and another index, `j`, into `y`—both initially zero—and compare `x[i]` with `y[j]`. If `x[i] < y[j]`, then append `x[i]` to `z` and increment `i` by one. Otherwise, append `y[j]` to `z` and increment `j`. If either `i` reaches the length of `x` or `j` reaches the end of `y`, simply copy the remainder of the other list to `z`.

Exercise: Argue why this approach creates the correct `z` list and why it terminates.

Now you get to implement a merge (at least if you go to GitHub and get the exercise).

Below or above

Here's a game you can play with a friend: one of you think of a number between 1 and 20, both 1 and 20 included. The other has to figure out what that number is. He or she can guess at the number, and after guessing will be told if the guess is correct, too high, or is too low. Unless the guess is correct, the guesser must try again until the guess *is* correct.

The game can be implemented like this (you can get your own version to play with on GitHub):

```
guess = input_integer("Make a guess> ")
while guess != true_val:
    if guess > true_val:
        print("Your guess is too high!")
    else:
        print("Your guess is too low!")
    guess = input_integer("Make a guess> ")
print("You got it!")
```

In this game, the computer gets to pick the number and you need to guess it.

Here are three different strategies you could use to guess the number:

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 correct 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. Tired of trying all numbers from one end to the other, you can pick this strategy: you start by guessing 10. If this is correct, you are done, if it is too high, you know the real number must be in the interval [1,9], and if the guess is too low, you know the right answer must be in the interval [11,20]—so for your next guess, you pick the middle of the interval it must be. With each new guess, you update the range where the real number can hide and choose the middle of the previous range.

Exercise: Prove that all three strategies terminate and with the correct answer, i.e. they are algorithms for solving this problem.

Exercise: Would you judge all three approaches to be equally efficient in finding the right number? If not, how would you order the three strategies such that the method most likely to get the correct number first is ranked highest, and the algorithm most likely to get the right number last is rated lowest. Justify your answer.

In the GitHub code there is also a version of the program where you get to pick the number and the computer guesses.

If you do not lie to the computer when it asks you about its guess compared to the number you are thinking of, the program implements the first strategy (going in sequence from 1 to 20).

Exercise: Implement the other two strategies and test them.

When iterating from 20 and down, for the second strategy, you should always get the result `"high"` when you ask about your guess, so you can use a `for` loop and not worry about the actual result form `input_selection`. When you implement strategy number three, however, you need to keep track of a candidate interval with a lower bound, initially 1, and an upper bound, initially 20. If you guess too high, you should lower your upper bound to the value you just guessed minus one (no need to include the guess we know is too high). If you guess too low, you must increase your lower bound to the number you just guessed plus one.

In both cases, after updating the interval, you should guess for the middle point in the new range. When you compute the middle value in your interval, you can use:

```
guess = (upper_bound + lower_bound) // 2
```

Sieve of Eratosthenes

The Sieve of Eratosthenes is an early algorithm for computing all prime numbers less than some upper bound n. It works as follows: we start with a set of candidates for numbers that could be primes, and since we do not a priori know which numbers will be primes we start with all the natural numbers from two and up to n.

```
candidates = list(range(2, n + 1))
```

We are going to figure out which are primes by elimination and put the primes in another list that is initially empty.

```
primes = []
```

The trick now is to remove from the candidates the numbers we know are not primes. We will require the following loop invariants:

1. All numbers in `primes` are prime.
2. No number in `candidates` can be divided by a number in `primes`.
3. The smallest number in `candidates` is a prime.

Exercise: Prove that the invariants are true with the initial lists defined as above.

We will now loop as long as there are candidates left. In the loop, we take the smallest number in the `candidates` list, which by the invariant must be a prime. Call it p. We then remove all candidates that are divisible by p and then add p to `primes`.

Exercise: Prove that the invariants are satisfied after these steps whenever they are satisfied before the steps.

Exercise: Prove that this algorithm terminates and is correct, i.e., that `primes` once the algorithm terminates contain all primes less than or equal to n. Correctness does not follow directly from the invariants so you might have to extend them.

Exercise: Implement and test this algorithm.

Longest increasing substrings

Assume you have a list of numbers, for example

```
x = [12, 45, 32, 65, 78, 23, 35, 45, 57]
```

Exercise: Design an algorithm that finds the longest sub-sequence `x[i:j]` such that consecutive numbers are increasing, i.e. `x[k] < x[k+1]` for all `k` in `range(i,j)` (or the left-most, if there are more than one with the same length).

A brute-force approach is to explore all `0 <= i < j < n` and check if `x[i:j]` is increasing, and then pick the longest of them. It is not particularly efficient, though, and there are smarter ways.

Hint: One way to approach this is to consider the longest sequence seen so far and the longest sequence up to a given index into `x`. From this, you can formalise invariants that should get you through.

If you want to try implementing your algorithm you can start from here.

Computing power-sets

The *power-set* P(S) of a set S is the set that contains all possible subsets of S. For example, if `S={a,b,c}`, then

$P(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$

Exercise: Assume that S is represented as a list. Design an algorithm that prints out all possible subsets of S. Prove that it terminates and is correct.

Hint: You can solve this problem by combining the numerical base algorithm with an observation about the binary representation of a number and a subset of S. We can represent any subset of S by the indices into the list representation of S. Given the indices, just pick out the elements at those indices. One way to represent a list of indices is as a binary sequence. The indices of the bits that are 1 should be included, the indices where the bits are 0 should not. If you can generate all the binary vectors of length `n = len(S)`, then you have implicitly generated all subsets of S. You can get all these bit vectors by getting all the numbers from zero to 2^n and extracting the binary representation.

If you want to implement a solution based on this approach, you don't need to find your old code. You can already compute a string that works as a binary representation of a number.

```
bits = format(i, "b").zfill(len(x))
```

The `format(i, "b")` converts `i` into binary and the `zfill` then makes sure that the string is `len(x)` long by padding with zeros.

Longest increasing subsequences

Notice that this problem has a different name than "longest increasing *substring*"; it is a slightly different problem. Assume, again, that you have a list of numbers. We want to find the longest sub-sequence of increasing numbers, but this time we are not looking for consecutive indices `i:j`, but a sequence of indices `i[0], i[1], ..., i[m]` such that `i[k] < i[k + 1]` and `x[i[k]] < x[i[k+1]]`.

Exercise Design an *algorithm* for computing the longest (or a longest) such sequence of indices `i[0], i[1], ..., i[m]`.

Hint: This problem is harder than the previous one, but you can brute force it by generating *all* subsequences and checking if the invariant is satisfied. This is a *very* inefficient approach, but you need to learn a little more about algorithms before we will see a more efficient solution (and it won't be in this class, sorry).

If you want to implement your solution, here's a template for you.

Changing base

When we write a number such as 123 we usually mean this to be in base 10, that is, we implicitly understand this to be the number $3 \times 10^0 + 2 \times 10^1 + 1 \times 10^2$. Starting from the right and moving towards the left, each digit represents an increasing power of tens. The number *could* also be in octal, although then we would usually write it like 123_8. If the number were in octal, each digit would represent a power of eight, and the number should be understood as $3 \times 8^0 + 2 \times 8^1 + 3 \times 8^2$.

Binary, octal and hexadecimal numbers—notation where the bases are 2, 8, and 16, respectively—are frequently used in computer science as they capture the numbers you can put in one, three and four bits. The computer works with bits, so it naturally speaks binary. For us humans, binary is a pain because it requires long sequences of ones and zeros for even relatively small numbers, and it is hard for us to readily see if we have five or six or so zeroes or ones in a row.

Using octal and hexadecimal is more comfortable for humans than binary, and you can map the digits in octal and hexadecimal to three and four-bit numbers, respectively. Modern computers are based on bytes (and multiples of bytes) where a byte is eight bits. Since a hexadecimal number is four bits, you can write any number that fits into a byte using two hexadecimal digits rather than eight binary digits. Octal numbers are less useful on modern computers, since two octal digits, six bits, are too small for a byte while three octal digits, nine bits, are too larger. Some older systems, however, were based on 12-bits numbers, and there you had four octal numbers. Now, octal numbers are merely used for historical reasons; on modern computers, hexadecimal numbers are better.

Leaving computer science, base 12, called duodecimal, has been proposed as a better choice than base 10 for doing arithmetic because 12 has more factors than 10 and this system would be simpler to do multiplication and division in. It is probably unlikely that this idea gets traction, but if it did, we would have to get used to converting old decimal numbers into duodecimal.

In this exercise, we do not want to do arithmetic in different bases but want to write a function that prints an integer in different bases.

When the base is higher than 10, we need a way to represent the digits from 10 and up. There are proposed special symbols for these, and these can be found in Unicode, but we will use letters, as is typically done for hexadecimal. We won't go above base 16, so we can use this table to map a number to a digit up to that base:

```
digits = {}

for i in range(0,10):
    digits[i] = str(i)

digits[10] = 'A'
digits[11] = 'B'
digits[12] = 'C'
digits[13] = 'D'
digits[14] = 'E'
digits[15] = 'F'
```

To get the last digit in a number, in base `b`, we can take the division rest, the modulus, and map that using the `digits` table:

```
digits[i % b]
```

Try it out.

You can extract the base-b representation of a number by building a list of digits starting with the smallest. You can use `digits[i % b]` to get the last digit and remember that in a list. Then we need to move on to the next digit. Now, if the number we are processing is `n = b**0 * a[0] + b**1 * a[1] + b**2 * a[2] + ... + b**n a[m]`, then `a[0]` is the remainder in a division by `b` and the digit we just extracted. Additionally, if // denotes integer division, `n // b = b**0 * a[1] + b**1 * a[2] + ... b**(m-1) * a[m]`. So, we can get the next digit by first dividing n by b and then extract the smallest digit.

If you iteratively extract the lowest digit and put it in a list and then reduce the number by dividing it by b, you should eventually have a list with all the digits, although in reverse order. If your list is named `lst`, you can reverse it using this expression `lst[::-1]`. The expression says that we want `lst` from the beginning to the end—the default values of a range when we do not provide anything—in steps of minus one.

Exercise: Flesh out an algorithm, based on the observations above, that can print any integer in any base $b \leq 16$. Show that your method terminates and outputs the correct string of digits.

If you feel up to it, you can try implementing the algorithm (here's a link to a GitHub exercise).

Be warned, though. While the algorithm is sound, there are special cases we didn't consider. I know many people who use this problem as an interview question exactly because the number of special cases (that depend on programming language, number representations, phases of the moon and much more) makes it so interesting. But give it a go!