CS1110 Summer 2021 — Assignment 6

Due Tuesday 8/3 at 11:00**am**

Read this before you begin

This assignment, which will also double as your last test, will consist of several independent exercises related to recursion and `while`-loops. There is no overarching theme here; on the other hand, it also means that you will be able to complete individual parts without having to complete the others.

Learning Goals The first two parts of this assignment are about recursion (Lectures 21 and 22). The last part is about `while`-loops (Lecture 23). You will be writing your own test cases again (Unit 7). We assume you are now comfortable using `if`-statements (Units 9 and 10). If you run into any trouble along the way, please be sure to use your debugging skills (Unit 11), particularly reading error messages, to help solve issues you run into.

Good style We want you to keep adhering to a clean coding style. That is why you can again earn **10 points** for adhering to the style guide as posted on Canvas. It may be worth it to consult this guide again, because we have discussed some new topics since the last time you looked at it.

Restrictions For the first two parts, there are several restrictions in place, because we want you to solve these problems using recursion specifically. Other than that, there are no hard restrictions on which parts of Python you are allowed to use. Keep in mind, however, that we expect you to fully understand what the code is doing. If you find some undiscussed feature of Python that you want use, that is fine, but please make sure you are precisely aware of how it works. In particular, we will still deduct points for exotic code that does not function correctly.

Submission For this assignment, you will submit two files: `a6.py` and `a6test.py`.

Submission within 24 hours after the deadline will be accepted with a 10% late penalty. Even if only one file (of several) is submitted late, the assignment as a whole will be marked late. Assignment submission on Canvas closes 24 hours after the deadline and no further submission will be allowed.

Do not put your name in the files that you will submit, but **do** include your NetID. The reason is to help promote fairness in grading. Thank you for heeding our request, even if it sounds a little strange.

Extensions Because this is the last assignment for this course, and because the School of Continued Education sets a hard deadline for entering the final grade, we have *very little* room for giving out extensions on this assignment. If you are in a situation where your mental or physical health (or some other circumstance beyond your control) prevents you from turning in this assignment on time, please get in touch *as soon as possible*.

Group work and academic integrity You must work either on your own or with one partner.

If you work with a partner, you must first register as a group in Canvas before you submit your work.

For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is *certainly* not OK to copy code from another person or from other sources.

If you feel that you cannot complete the assignment on you own, please seek help from the course staff.

Part A: Binary Search [20 points]

Background Let's say you are given a list of integers `numbers`, and you want to find out whether a certain integer `n` appears in the list. The straightforward (and usually most efficient) method is to check whether `numbers[0]` is equal to `n`, then check whether `numbers[1]` is equal to `n`, and so on, until you reach the end of the list.

When `numbers` is *sorted*, however, more efficient techniques become possible. For instance, you could still look at each element, but once you start seeing numbers that are higher than the number you are looking for, there is no hope of ever finding the right number, and so you can stop before you reach the end of the list.

This part of the assignment is about an *even more efficient* method of searching for a given integer `n` inside a sorted list `numbers`, called *binary search*. The binary search algorithm first looks at the number `m` at the *middle* of a sorted array. If `m` is equal to `n` (the number you are searching for), the algorithm has found the number, and can stop. Otherwise, if the number you are looking for is *smaller* than the number in the middle, you know that, if `n` appears in `numbers` at all, it must appear *before* the `m`. Similarly, if `n` is *greater* than the number in the middle, you know that you need to look in the part of `numbers` *after* `m` to see if `n` appears in there.

For example, let's say we have the list `[0, 2, 5, 8, 10, 42, 101]`, and we are trying to work out whether the number `42` appears inside of this list. First, we look at the middle of the list, where we see the number `8`. Since `8` is smaller than `42`, this means we must now search in the part of the list following `8`, i.e., in `[10, 42, 101]`. The middle of this list is 42, and so the algorithm can stop here, because we found what we were looking for.

Your task You need to implement the function `binary_search` according to its docstring in `a6.py`.

This function **must** be implemented recursively. In particular, this means that you are not allowed to use any loop primitives (i.e., `for`-loops and `while`-loops, including list comprehensions). It also means that you are not allowed to use any of the built-in search methods for lists like `index`, or the built-in function `filter`.

Hints Some ideas to send you on your way:

- Think of a specific input when your function *always* returns `False`. This is the base case. You can also take a look at some of the recursive search functions we have implemented before — they have the same base case.
- Your function does not need to raise any specific errors, as can be seen in the docstring. If you want, you can still put in assertions or some checks that make your life easier, but we will not grade for these.
- Don't overthink it. Our reference answer is just 9 lines of code (not counting blank lines). Your code may be longer, and that is OK too. You can look at the implementation of `merge` from Lecture 23 for inspiration.

Part B: Gray Code [40 points]

Background For Assignment 4, we looked at binary numbers, which are used to store almost all integers that your computer uses. In certain situations, however, it is a good idea to encode your numbers differently.

For instance, suppose you are digitally telling your friend that they should bring 42 bottles of beer to the party you are organizing together. In standard binary, the number 42 is written down as **`00101010`**. If the medium you are using to transmit these numbers is not very reliable, it may happen that one of the **`0`**s arrives as a **`1`**. When this happens to the first **`1`**, your friend will receive the binary number **`00001010`**, which means they'll bring just 10 beers. Similarly, if the first **`0`** is flipped to a **`1`**, your friend will receive the number **`10101010`**, and end up bringing 170 bottles. This may be good for your party, but it won't be good for your wallet.

This part of the assignment is about *Gray code*, which is a method of representing whole numbers using zeroes and ones in such a way that codes that correspond to close numbers are themselves also close in terms of number of changed bits. Let's see an example first. The Gray code for three bits is given by the following table.

Number	Gray code
0	**`000`**
1	**`001`**
2	**`011`**
3	**`010`**
4	**`110`**
5	**`111`**
6	**`101`**
7	**`100`**

Notice how, if you go from one row to the next in the table above, the code in the second column changes at exactly one position. For example, in the third row we see the code **011**, while the fourth row has the code **010**. To change the first code into the next, all you have to do is flip the last bit from **1** to **0**.

When you use 3-bit Gray code to transmit your numbers, there is a good chance they will not change very much if your Gray code changes from one row to the next. For example, if your friend should bring three bottles of wine, you would send the Gray code **010**. If your friend receives the adjacent code **110** instead, they will bring only four bottles of wine, which is not so much of a change. Your party (and your wallet) will be relatively safe.

Now, you might wonder: *how do you generate such a list of binary numbers?* Fortunately, the answer can be worked out by using recursion. Let's assume you have already figured out the Gray codes for `n-1` bits. For instance, the translation table for the Gray code for 2 bits is given by the following table:

Number	Gray code
0	**00**
1	**01**
2	**11**
3	**10**

To derive the Gray code for `n` bits from the Gray code for `n-1` bits (*which you compute recursively!*), you need to:

1. Create the reverse of the *old* list; this becomes your *reverted* list. For the table above, this works out to be

   00, **01**, **11**, **10** gives **10**, **11**, **01**, **00**

2. Put **0** in front of each number in your *old* list. So, for the table above, this works out as follows

   00, **01**, **11**, **10** gives **000**, **001**, **011**, **010**

3. Put **1** in front of each number in your *reverted* list (from the first step). So, for the example we find

   10, **11**, **01**, **00** gives **110**, **111**, **101**, **100**

4. Add the lists computed in steps 2 and 3 together. So, for the example, we get

   000, **001**, **011**, **010**, **110**, **111**, **101**, **100**

   which is exactly the list from the table for 3-bit Gray code that we saw before.

The only question that remains is what the "simple" cases are for this process — i.e., when do we *not* rely on the Gray code for fewer bits? This may not be obvious, but when you have *zero* bits, there is actually exactly *one* Gray code: the one with no bits at all. In Python, this is represented as the empty list (with no booleans inside).

Your task You need to implement the function `gray_code` in `a6.py`, so that it returns the list of `n`-bit Gray codes for any nonnegative input integer `n`. To do this, you have to implement the steps detailed above. Like the last part, your implementation **must** be recursive (along the lines of the process above). However, for this part, you *can* use loops to implement the second and third step above, where you put **0** or **1** in front of each number.

To revert the list in the first step above, you are *not* allowed to use the built-in `reverse` method, nor are you allowed to use the slicing trick `x[::-1]` to get the reverted version of a list `x`. Instead, you **must** implement the helper function `revert`, which computes the reverse of the input list, recursively. Your implementation of `gray_code` **must** then use this helper function in the first step.

Hints Some things to keep in mind as you implement this part:

- Don't panic. If you calmly implement the steps above, you will get there.
- Don't overthink it. Our implementation is 11 lines long (excluding blank lines). Yours can be longer.
- Think recursive. To compute `gray_code(n)`, you need `gray_code(n-1)` in almost all cases.
- The easy "base" case is spelled out in one of the examples in the docstring.
- Keep your tests simple. If your (recursive) code works on the given examples, it is probably OK.

Part C: Approximating Square Roots [30 points]

Background Say you have a number S, and you want to find the square root $\sqrt{S}$. Another way of phrasing this problem is as follows: you are given the function $f(x) = x^2 - S$, and you are looking for an x such that $f(x) = 0$. Newton's method, which you may have encountered in a calculus class, is a way to find a very good approximation for x. It starts with an initial guess of x, and continuously improves the guess until it is "good enough".

The exact details of Newton's method in general do not matter for this part. We will tell you how to apply it to this particular problem now. First, you start out with an initial guess x_0. Next, you update your guess to find new guesses x_1, x_2, and so on. Concretely, you calculate your new guess x_{n+1} from the old guess x_n as follows:

$$x_{n+1} = x_n - \frac{x_n^2 - S}{2 \cdot x_n}$$

The next question becomes: when do you know you can stop? Well, since we are approximating the square root, one way of checking the quality of our guess is to square it, and to see how far it is from the input S. In other words, we would like $|x_n^2 - S|$ to be small. When this value is small enough, we stop refining our approximation.

As an example, let's say you want to approximate $\sqrt{2}$. You remember that it should be one-point-something, so you start off your initial guess at $x_0 = 1.9$. You then calculate x_1 and x_2, as follows:

$$x_1 = x_0 - \frac{x_0^2 - 2}{2 \cdot x_0} = 1.49750 \qquad\qquad x_2 = x_1 - \frac{x_1^2 - 2}{2 \cdot x_1} = 1.41653$$

The square of x_2 is 2.00656, which is "close enough" to 2, and so we stop. Notice how the true value of $\sqrt{2}$ is about 1.41421, so our approximation x_2 is somewhat decent at this point — two decimal places of accuracy!

Your task You need to implement the above procedure to approximate square roots, in the `approx_sqrt` function in `a6.py`. To this end, you should come up with an initial guess for the square root. Your guess can be very bad (that is OK!), but it should depend on the input S. Then, you need to keep improving that guess using the method described above, until finally the difference between the square of your guess and the input value is small enough. You may *not* use a `for`-loop specifically to do this, and you are not allowed to use the `math` module to implement this function, or anything that computes square roots directly for you.

Hints Here are a few pointers for your implementation:

- Your implementation should not need to use a list. Simple variables suffice.
- In particular, you do not need to store all of your earlier guesses. Simply update your guess in each step.
- The absolute value of a variable `x` (denoted `|x|` above) can be calculated using the built-in function `abs`.
- The restrictions mentioned in the first two parts do not apply here. You should use a `while` loop.
- Don't overthink it. Our reference answer is just four lines of code (not counting blank lines).
- Don't choose a `tolerance` value that is too small when testing; in cases like these, it is possible for your function to get "stuck" because of numerical imprecision inherent to floating point numbers.
- Even though you are not allowed to use `math` to implement the function, you *can* use it in your tests.
- Remember that comparing floats directly is a bad idea; instead, you should use `assert_floats_equal` function provided by the `testcase` module to do this.