Assignment 1

Submission instructions: Upload to canvas a zip archive containing `part*.py` (i.e. `part1.py`, `part2.py`, ...), and a text file called `output.txt` showing the output of running `autograder.py`.

Collaboration policy: You are encouraged to discuss these problems with other students. Include the names of anyone you collaborated with in this capacity. But do not copy other's answers or code. See *Academic Integrity* on the syllabus.

ChatGPT policy: You welcome to use ChatGPT, Copilot, etc. BUT if you do that, you *need* to include an extra paragraph in `output.txt` where you detail what you used these models for, what they succeeded at, what they failed at, and what you learned about their capabilities from using them. See *ChatGPT* on the syllabus.

Part 1: Implement an interpreter for a simple programming language (5 points)

You will be writing two program synthesizers for a domain specific language (DSL) allowing basic arithmetic. This DSL is generated by the following context-free grammar:

```
IntExpression ::=
    |  Plus(IntExpression, IntExpression) // arithmetic: addition
    |  Times(IntExpression, IntExpression) // arithmetic: multiplication
    |  Number(Z) // Z is an integer, such as 0, 1, -5
    |  NumberVariable(N) // N is the name of a variable, such as 'x' or 'y'
    |  If(BoolExpression, IntExpression, IntExpression) // conditionals

BoolExpression ::=
    |  LessThan(IntExpression, IntExpression) // check if one number is smaller than another
    |  And(BoolExpression, BoolExpression) // logical conjunction
    |  Not(BoolExpression) // logical negation
    |  False // logical constant: false
```

But first, as a warm-up, you will be writing an interpreter for this language.

Inside of `part1.py`, you will see the definition of a class called `Expression`, which is the parent class of all abstract syntax tree (AST) nodes. You will also see other classes which inherit from `Expression`, such as `NumberVariable` and which correspond to the different rules in the above context free grammar. (What non-terminal symbol a class corresponds to is indicated by the `return_type` field, *not* its parent class, which is always `Expression`.)

There are a few other methods which you will have to override in other parts of this problem set, such as `minimum_cost_member_of_extension` and `version_space_size`. Ignore them for now.

Write a method for each of the classes in `part1.py` called `evaluate`. The `evaluate` method should take as input an `environment`, which is a dictionary mapping variable names to values. After you do this you should be able to run `part1.py` and see that all of the test cases pass. To get you started, we have provided implementations of `evaluate` for the classes `NumberVariable`, `Number`, and `Plus`.

Part 2: Implement a bottom-up enumeration synthesizer (30 points)

Now that you have fleshed out this simple programming language, you are going to write a program synthesizer for it which enumerates expressions in increasing order of size. This synthesizer will take as input a specification via input-outputs. You must also implement the standard trick of only keeping around the smallest observationally equivalent program on the given input-outputs.

Inside `part2.py` you will see a wrapper routine called `bottom_up`. This routine calls a function that you need to write, `bottom_up_generator`, which should `yield` programs (i.e. `Expressions`) in increasing order of AST size.

Make sure that you implement your `bottom_up_generator` so that it is agnostic as to the operators and terminals in the DSL: you will reuse it for different DSLs in this problem set. It should also be agnostic as to the typing system, and should only enumerate well typed expressions. You can assume that the DSL is defined by classes which are subtypes of `Expression`, and that each such class has a class attribute called `return_type` (which is the type of values returned from `.evaluate` calls on instances of that class) and also another attribute called `argument_types` (which is a list of types that need to be input into the constructor of the class).

We provide some example starting code for `bottom_up_generator` but you are free to not use it if you wish. If you want to make your code truly generic, you will need to handle DSL operators which take an arbitrary number of inputs. The provided `integer_partitions` helper routine may prove useful.

After you do this you should be able to run `part2.py` and see that all of the test cases pass. The last test case might take 10-30 minutes.

Part 3: Implement divide-and-conquer synthesis with enumeration backend (19 points)

Part 3.1: Programming (16 points)

Now you are going to reuse your code from Part 2 to build a sophisticated synthesizer, eusolve. Inside of `part3.py` you will see a function that you have to write, `dcsolve`, and you are free to follow the pseudocode in the original paper. You should call your `bottom_up_generator` as a subroutine. As a reminder, this should return a generator, and you can get the next element from a Python generator using `next(a_generator)`.

Because this is a synthesis class and not a machine learning class, you are welcome to use the reference implementation we give for learning decision trees (`learn_decision_tree`). Or you are welcome to code your own.

Part 3.2: Answer a question (3 points)

When you have succeeded, all of the test cases in `part3.py` should pass. Pay attention to the last test case, which is the same in parts two and three. How much faster was eusolve compared to bottom-up enumeration? Why was it so much faster?

Put your response in `question_3_2` in `part3.py`.

Part 4: Create a simple FlashFill implementation (38 points + 0 point bonus problem)

We are now going to work with a different DSL inspired by FlashFill - notice that unlike actual FlashFill, there are no loops, conditionals, or regular expressions:

```
Program ::=
   |  Concatenate(Program, Program) // concatenate strings
   |  ConstantString(str) // constant string, e.g. str="hello world"
   |  Substring(Var, Int, Int) // extract substring given a pair of integer indices

Int ::=
   |  Number(Z) // Z is an integer, such as 0, 1, -5


Var ::=
   |  StringVariable(N) // N is the name of a variable, such as 'input1' or 'input2'
```

In `part4.py` you will see implementations of these classes. We are going to reuse the implementation of `Number` from `part1.py`.

Part 4.1 (5 points)

As before, start out by defining `evaluate` methods on these classes. `Substring` semantics are tricky, and to ensure consistency across homeworks, we have provided its `evaluate` method.

If you run `part4.py` you should see `[+] 4.1, text editing evaluation, +5/5 points` printed out when you have successfully implemented the `evaluate` methods.

Part 4.2 (5 points)

Implement version space routines. We have provided a class, `Union`, which should be used in order to build AST-like structures representing sets of programs. Recall that FlashFill-style version spaces represent these sets of possible programs. Define simple recursive methods on the `Expression` classes used in this DSL (including the `Number` class in `part1.py`) which will compute the size of the set of programs contained in the version space (`version_space_size`), and also return a list of such programs (`extension`), and also return the smallest program in the extension of the version space (`minimum_cost_member_of_extension` - be sure to write this recursively, and if you memoize it the test cases will run faster but this is not necessary for full credit). We have provided example implementations for `Union` and `StringVariable` to get you started.

If you run `part4.py` you should see `[+] 4.2, version space evaluation, +5/5 points` printed out when you have successfully implemented these methods.

Part 4.3 (10 points)

Implement FlashFill for a single input-output example. The function you need to fill in is `flashfill_one_example`, which takes an `environment` and a `target_output`. It should return a new version space containing all programs generated by the above grammar which map the `environment` to the `target_output`. You should implement this function recursively, but you absolutely *must* use dynamic programming. We have provided a strong hint of an optional `dynamic_programming_table` argument, which should be threaded through recursive calls. We recommend implementing subroutines `generate_substring` and `generate_position`, which correspond to witness functions for `Substring` and its integer arguments, respectively. Feel free to consult the FlashFill paper, which sketches `generate_substring` and `generate_position` in Figure 7.

Tip: When building `Unions` you should try to use the `Union.make` static method, which automatically flattens nested unions and prevents the creation of singleton unions. This will make your version spaces look nicer when printing them, so it's helpful for debugging.

When you are done, you should be able to run `part4.py` and see the text printed out `[+] 4.3, flashfill_one_example, +10/10 points`.

Part 4.4 (10 points)

Implement FlashFill for multiple input-output example. FlashFill for multiple input outputs is simple: just run the synthesizer on each such example individually to get a version space, and then intersect those version spaces to find what programs will work for every example. Implement `flashfill`, which will also require implementing version space intersections. You should implement the intersection routine in `intersect`. You should definitely do dynamic programming for intersections, and we have provided a hint of setting up such dynamic programming at the beginning of `intersect`. As an aside, dynamic programming is pervasive in version space synthesis methods.

When you are done, you should be able to run `part4.py` and see the text printed out `[+] 4.4, flashfill, +10/10 points`.

Part 4.5 (3 points)

Question: Inspect the outputs from your implementation. How large do the version spaces get, in terms of how many programs they represent? How does this size change as more examples are added, and why does that happen? Write a short response in the function `question_4_5` in `part4.py`, what you should see printed out when you run `part4.py`

Part 4.6 (5 points)

If you look at the synthesis test cases and the corresponding expressions that your code synthesizes, you should observe that the synthesize program isn't always the most natural code to write. In the test cases, after getting the version space of all programs, we output the "minimum cost program" in that version space, where cost is defined using the `.cost`/`.minimum_cost_member_of_extension` methods. For example, you should see something like:

```
[+] passed synthesis test case:
         {'input1': 'June', 'input2': '14', 'input3': '1997'}  --> '1997, June 14'
        with the program:
         "1997, June 14"
```

meaning that, when given the single input-output example `('June','14','1997') --> '1997, June 14'`, the system decided to synthesize a program which just always prints out the constant string `"1997, June 14"` rather than build that string from the inputs. Although technically correct, this is probably not what a user of a programming-by-examples system wants! Even given two input-outputs, the minimum cost program is subtly wrong:

```
    [+] passed synthesis test case:
         {'input1': 'June', 'input2': '14', 'input3': '1997'}  --> '1997, June 14'
         {'input1': 'October', 'input2': '2', 'input3': '2012'}  --> '2012, October 2'
        with the program:
         Substring(input3, 0, 3) + ", " + Substring(input1, 0, -1) + " " + Substring(input2, 0, -1)
```

Here, the system has learned to take only the first four characters of `input3` rather than take the whole string.

Modify the definitions of the `cost` method for the `Number` and `ConstantString` classes so that the minimum cost program printed out during the test cases looks correct. It should be biased toward using the entire input string (it should assign lower cost to using the entire input string). Also, it should be biased against using constant strings (it should assign higher cost to constant strings, with longer strings also having higher cost). Rerun `part4.py` to verify that you get the following outputs:

```
[+] passed synthesis test case:
         {'input1': 'June', 'input2': '14', 'input3': '1997'}  --> '1997, June 14'
        with the program:
         Substring(input3, 0, -1) + ", " + Substring(input1, 0, -1) + " " + Substring(input2, 0, -1)
```

and

```
[+] passed synthesis test case:
         {'input1': 'June', 'input2': '14', 'input3': '1997'}  --> '1997, June 14'
         {'input1': 'October', 'input2': '2', 'input3': '2012'}  --> '2012, October 2'
        with the program:
         Substring(input3, 0, -1) + ", " + Substring(input1, 0, -1) + " " + Substring(input2, 0, -1)
```

If everything is working correctly you should see `[+] 4.6, cost definitions, +5/5 points`

Part 4.7: Optional challenge (+0 points)

Optionally, rewrite your FlashFill program synthesizer to more closely match the modern FlashMeta framework as follows:

- For each of the DSL components (`Number`, `Concatenate`, `StringInput`, `Substring`) create a static class method called `witness`. This method should take a single input-output specification, and return as output specification(s) on the arguments to that DSL component which must be satisfied in order to satisfy the original single input-output specification. You can do this in several ways, but at a high level the idea is that if we want to satisfy a specification ϕ using DSL component κ of arity A (so κ is called with A arguments, i.e. $\kappa(e_1, \cdots, e_A)$), then we want to output a collection of A-tuples of specs where, for each such tuple $\phi_1 \cdots \phi_A$:

$$\left(\bigwedge_{j=1}^{A} 1\left[e_j \text{ satisfies spec } \phi_j\right]\right) \implies 1\left[\kappa(e_1, \cdots, e_A) \text{ satisfies spec } \phi\right]$$

  The above the equation amounts to *soundness* (any specs returned by κ's witness function run on ϕ will give you an expression built via κ which satisfies ϕ). Make sure that your witness functions also satisfy *completeness* (that any expression built via κ which satisfies ϕ also has arguments which satisfy specs returned by the witness function). The lengthy docstring on `extra_credit_witness_synthesize` suggests one way of structuring the outputs of `witness`.

- Implement `extra_credit_witness_synthesize` by recursively calling the witness functions on every DSL component to break the specification into smaller specifications on the input of each component, and then recursing as appropriate (remember to do dynamic programming!). Aggregate the returned programs into version spaces. It should run just as fast as `flashfill_one_example`. Test it out adding the following to the top of `flashfill_one_example` and ensuring that all the test cases still pass:

```
return extra_credit_witness_synthesize( (frozendict.frozendict(environment), target_output),
                                        [ConstantString,Number,Substring,Concatenate])
```

Question to think about: Write down an equation which describes what it means for a witness function to be *complete*, in the notation used above when defining what it means for a witness function to be *sound*.

Part 5: Bottom-up enumeration on the FlashFill DSL (8 points)

Part 5.1: Implementation (5 points)

In `part5.py` you should reuse your implementation of `bottom_up` with your implementation of the FlashFill DSL in order to see what happens when you attempt to use enumeration to solve these problems. Concretely, implement `bottom_up_flashfill` using components that you wrote in earlier parts. You will have to make a decision about what string constants to include, and the provided skeleton suggests using all printable characters. You will also have to make an a priori decision about what `int` constants to use. You can either use a fixed range of possible integers or you can decide to set the range depending on the input-outputs. By just calling the code you have already written, you should be able to do `bottom_up_flashfill` in about five lines of code.

When you are done, you should be able to run `part5.py` and see the text `[+] 5.1, bottom-up "flashfill", +5/5 points`.

Part 5.2: Question (3 points)

Answer the following question: The last test case in 5.1 should take a long time. Why does it take a long time? What does this tell you about the ability of enumeration to synthesize arbitrary constants? Put your response in `question_5_2`.

Submission instructions (reminder!)

Upload to canvas a zip archive containing `part*.py` (i.e. `part1.py`, `part2.py`, ...), and a text file called `output.txt` showing the output of running `autograder.py`. Note that it might take up to 30 minutes in order to produce the final output, so work on each piece individually and run `partN.py` to check your intermediate work instead of running the full autograder.