# Foundations of Algorithms, Fall 2021

Liam McCann

Due Monday, October 18th

## 1 Included Code

This project includes the following files:

1. **algo.py** – This file contains the algorithm definitions.
2. **function.py** – This file defines a class for working with polynomial functions and converting string representations of recursive functions into function objects
3. **tree.py** – This file defines a recursion tree object and a node object for creating and printing recursion tree results
4. **rational.py** – This file contains a class for working with Rational numbers in the context of this project
5. **logger.py** – This file defines the root logger for the application.
6. **settings.py** – This file defines all settings for the application.
7. **Pipfile** – This file is used by *pipenv* to manage the packages installed in the python virtual environment and defines scripts for easily working with the environment.
8. **Pipfile.lock** – This file is used by *pipenv* to install required packages.
9. **conftest.py**
10. **test_function.py** – This file defines tests for the functions in the function.py file.
11. **test_algo.py** – This file defines tests for running the different recursive functions defined within the problem statement.

In addition to the included Python files, the attached .zip file contains selected output files. Application coverage files (.cover) files are generated by *trace* and are found in *output/trace*. A sample of the debug logs, which help to trace the execution of the algorithm, can be found at *output/debug.log*.

## 2 Development Setup

### 2.1 Required Dependencies

This project requires dependencies to install and run the necessary Python packages to test and run the algorithm:

1. *Python 3.9*
2. *Pipenv*

*Pipenv* can be installed using Pip or another command line install tool depending on the operating system. The full installation details can be found at `https://pipenv.pypa.io/en/latest/`. Once both *Python 3.9* and *pipenv* are installed, *pipenv* can be used to install the necessary Python packages. The following command can be used to complete the installation:

*pipenv install -d*

# 3 Algorithm

## 3.1 Pseudocode

This section details the pseudocode for the necessary algorithms for creating a recursion tree for a recursive function $T(n)$.

**Pre-Process-Function($func$):**
$reg$ = regular expression matching polynomial functions
$reg_\Theta$ = regular expression matching Big-$\Theta$ functions
$reg_O$ = regular expression matching Big-O functions
$reg_\Omega$ = regular expression matching Big-$\Omega$ functions
$reg_c$ = regular expression matching constant values
$match$ =NIL
**if** $func$ matches $reg$:
    $match$ = portion of $func$ matching regular expression
**else if** $func$ matches $reg_\Theta$:
    $match$ = portion of $func$ matching regular expression without $\Theta$
    $match = c + match$
**else if** $func$ matches $reg_O$:
    $match$ = portion of $func$ matching regular expression without $O$
    $match = c + match$
**else if** $func$ matches $reg_\Omega$:
    $match$ = portion of $func$ matching regular expression without $\Omega$
    $match = c + match$
**else if** $func$ matches $reg_c$:
    $match$ = portion of $func$ matching regular expression
**else**
    **return**
$c$ = parse the first term in the match
$d$ = parse the second term in the match
// Create a polynomial function with the cost function variables
// The initializer is defined in **Init-Polynomial-Function**
**return** PolynomialFunction($func$, $c$, $d$)

## 3.2 Worst-case Running Time

The worst-case running time for the **Create-Tree** algorithm is defined by three distinct process. The pre-processing step, the recursive step, and the computation step (which computes the total cost of each level of the recursion) that is run during each recursive call. The pre-processing step involves ingesting a string function and parsing this string into an object (PolynomialFunction defined in *function.py*) that is more accessible for further computations. The cost of this operation directly correlates to the type of function that is given and the size of the input string. For instance if we were to provide a division-based recursion algorithm with a polynomial cost function, the algorithm has to visit all characters in the string, giving this algorithm a $O(n)$ running time in terms

```
    Init-Polynomial-Function(func, c, d):
1   reg_sub = regex expression for matching T(n) type functions
2   reg_div = regex expression for matching T(n-b) type functions
3   if func matches reg_sub:
4       a = parse a from aT(n/b)
5       b = parse b from aT(n/b)
5   else if func matches reg_div
4       a = parse a from aT(n − b)
5       b = parse b from aT(n − b)
7   else:
8       error Couldn't parse function.

    Create-Tree(func, tree, layer):
1   if layer ≥ 0:
2       Create-Tree(func, tree, layer − 1)
3   if layer == 0:
        // Add a new layer, represented by an array of nodes, to the tree object
4       return Add-Layer-To-Tree(func, layer, layer)
5   if layer ≥ 0:
        // Add a new layer, represented by an array of nodes, to the tree object
6       return Add-Layer-To-Tree(func, func.a ∗ layer, layer)
7   return tree
```

of the length of the input string, $n$.

The recursive step for defining each layer of the recursion tree is defined by the recursive function that is used to generate the recursion tree and is capped at 4 recursive calls to generate 4 layers of the tree. Using either the Divide and Conquer ($T(n) = aT(n/b) + f(n)$) or the Chip and be Conquered ($T(n) = aT(n-b) + f(n)$) function format, the value $a$ determines the additional number of nodes generated on each layer. For instance, if $a = 7$ then the second layer of the tree will contain 7 nodes and then layer 3 would contain 7 * 7 nodes since each node will subdivide by a further 7 nodes. It is difficult to assign a measurement which relates the length of the input function string to the number of nodes generated during the recurrence, so we will instead let the number of nodes in the in final recurrence layer be $m$, since this is the worst case running time.

The last piece to analyze is the running time of computing the total cost for a layer within the recurrence tree. A naive approach might simply add the cost of each node, which would result in a running time of $O(m)$, however, we know that each node in a layer has the same cost, so we can simplify. Instead of adding all nodes we can simply multiple the cost by the number of nodes in the layer. Given that determining the length of the array containing the nodes can be done in constant time, the operation to determine the cost of the entire layer can also be computed in O(1) time.

Therefore, the overall worst-case running time of the **Create-Tree** algorithm $O(m + n)$.

# 4 Trace Runs

## 4.1 trace Module

The trace module allows records both the count of function calls as well as the stack trace of function calls. Since we are using debug logging within the application, the *trace* module primarily provides a count of the function calls for this application. The count of function calls helps to reveal the points in the code that are

**Add-Layer-To-Tree(**$func$**,** $data$**,** $layer$**):**
// total cost of the layer by multiplying the cost of one node by num nodes in the layer
1 $totalCost$ = (cost of $data$[0]) * $data.length$
2 Tree[layer] = ($data$, $total_cost$)

most heavily executed and contribute heavily to the overall running time. This algorithm does not require any particularly heavy load on a single function call. However, if we look at the *tree.cover* file across any of the traced recursive functions, we can see the cost of the node is frequently accessed. This is because we need to compute the overall cost at each step. These traces aid our understanding of the overall running time of the algorithms and point towards potential optimizations.

The *trace* package can be run against the entire project or against a single file using a *pipenv* script, which is defined as follows:

*pipenv run trace <file>* (where *algo.py* is *<file>* for the entire application)

Example *.cover* coverage files are included in the .zip file at */output/trace*. Several functions have been traced to showcase the differences in running time for different recursive functions. There is a text file in each directory showing the command that was run to generate the coverage traces.

## 4.2 Debug Logging

The second way that the application stack trace can be monitored is by using debug logging. For many applications, debug logging offers engineers and developers a way to better understand where errors might be occurring and how their code is executing over time. Similarly, in the context of this programming project, debug logs offer insight into the execution of the **Create-Tree** algorithm. Debug logs are not enabled by default, however, they can be enabled using an environment variable. Setting *LOG_LEVEL* to *DEBUG* will tell the application to display *INFO*, *DEBUG*, and *ERROR* level logs. For instance, running the following command will run the **Create-Tree** algorithm for a given recursive function with debug logs enabled:

*LOG_LEVEL=DEBUG pipenv run generate <recursive function>*

Debug logs are also enabled by default when using *pytest*. So running *pipenv run test* will also reveal the debug logs. There are quite a few logs generated for the set of included tests.

An example log file is included in the .zip file for this project and can be found at */output/debug.log*. These logs are generated from the test files to include the set of functions required by the assignment. These logs are generated using the following command: *pipenv run test -k test_algo*.

# 5 Tests

The application uses the *pytest* module to run unit-tests of the various functions. The tests can be run using a *pipenv* script.

*pipenv run test*

These tests run the full set of recursive functions required by this assignment. There are also tests which verify that the function parser, which converts string representations of the input functions into object representations, are working as anticipated. Using these tests, we can verify that the input functions are being correctly interpreted and that the total cost at each layer of the tree is correctly computed.

# 6 Pseudocode versus Implementation

To evaluate difference between the perceived running time using the pseudocode and the actual running time given by the implementation, we varied the values of certain parameters within the recursive function $T(n)$. As noted in Section 3.2 Worst-case Running Time, the running time of both recursive functions of format $T(n) = aT(n/b)+f(n)$ and $T(n) = aT(n-b)+f(n)$ are heavily predicated on the $a$ term of the expression. This is because the coefficient $a$ directly determines the number of nodes which will be generated on each recursive call. In fact, with the number of nodes can be computed as $a$ to the power of the layer of interest (for example $numNodes = a^4$ gives the number of nodes in layer 4 not inclusive of the root of the tree).

We use the Python trace packages coverage functionality to determine how many function calls are made to a specific line in our code. These results of our testing can be found in the *output/trace/func** directories. We tested functions with the $a$ coefficient equal to 7, 1, and 3. As anticipated, we saw that the number of nodes generated is higher when the coefficient is higher. This also means that the number of computations for each layer also increases since we need to create an object to represent each node.

The implementation, therefore, appropriately matches the worst-case running time we saw in the pseudocode $O(n+m)$.