COMPSCI 311: Introduction to Algorithms **Fall 2018**

Extra Credit Assignment

Released 10/30/2018 **Authors: Amir Ghafari and Rik Sengupta** Due 11:59pm 12/06/2018

1. This assignment is extra credit. It is entirely optional.
2. This assignment is worth up to 5 grade percentage points.
3. Every member of a group working together will get the same number of points.

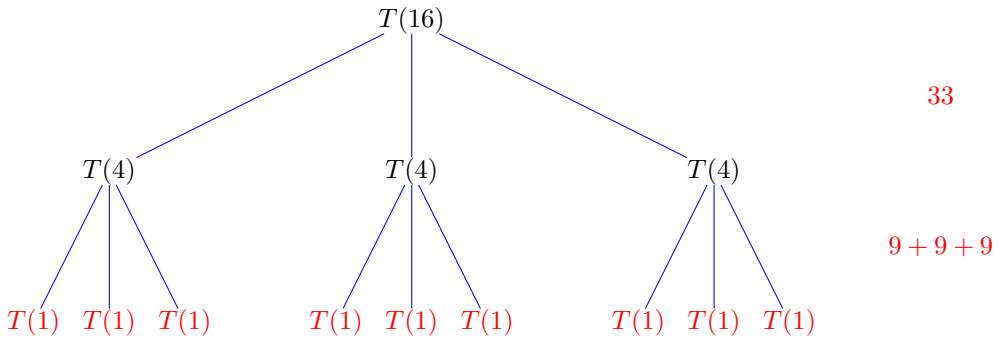
Problem. The problem is in four parts. You can do as many or as few as you like.

1. **(10 points)** Design and implement an efficient algorithm that, given an input (r, a, b, c, d, e), checks that

   - r, a, b, c, d, e are nonnegative integers, $1 \leq a \leq 4$, $b \geq 2$, $r \leq 200$.
   - $r = b^k$ for some $k \in \{1, 2, 3\}$.

   If either of these checks fail, the algorithm outputs `INVALID INPUT`; otherwise it outputs the recursion tree for the recurrence $T(n) = aT(n/b) + cn + d$ (base cases $T(n) = e$ for $n < b$) **rooted at** r (see **Implementation** for details). The nodes should be labeled, and each node (including the root r) should store the recursively computed value of T at that node.

   For instance, if the input given is $(16, 3, 4, 2, 1, 1)$, then the recursion tree looks like this:



$$T(16) = 69$$

   Your algorithm should output this tree, with two stored attributes for each node: its label, and the value of T at that node. You should convince yourself that given the recurrence

$$T(n) = 3T(n/4) + 2n + 1 \text{ with } T(n) = 1 \text{ for } n < 4,$$

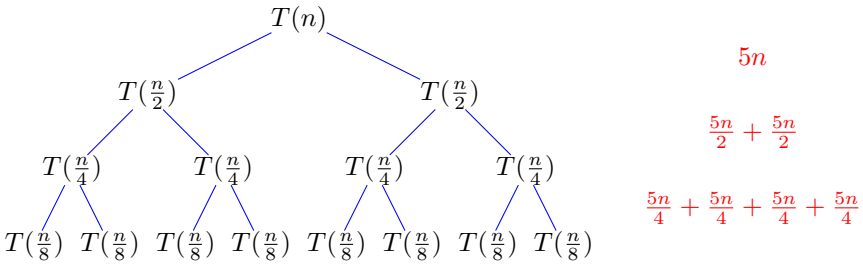
   the value of $T(16)$ is, indeed, 69: the recursion adds up everything in red in the picture above.

2. **(10 points)** Design and implement an efficient algorithm that, given an input (a, b, c, d), checks that

   - a, b, c, d are nonnegative integers, $1 \leq a \leq 4$, $b \geq 2$.

   If the check fails, it outputs `INVALID INPUT`; otherwise it outputs a tree corresponding to the **first four levels** of the recursion tree for the recurrence $T(n) = aT(n/b) + cn + d$. The nodes again should have two attributes: their label (as a function of n), and their value (as a function of n).

   For instance, if the input given is $(2, 2, 5, 0)$, the first four levels of the recursion tree look like this:



Your algorithm should output this tree, with two stored **string** attributes for each node: its label, and the value of T at that node. For instance, the label of the left child of the root in the tree above should be the string `T(n/2)`, and its value should be the string `2T(n/4) + 5n/2`.

3. **(15 points)** As an extension to part 2, given (a, b, c, d) with the same constraints, output a closed form for the asymptotic behavior of the function $T(n) = aT(n/b) + cn + d$. Make any reasonable assumptions about $T(n)$ being constant for "low enough" values of n (e.g. $n < b$), and explain this in your README file. So in the previous example, your algorithm should output `Theta(n*log n)`. Please see the implementation details below for how you should output logarithms.

4. **(15 points)** Same as the previous part, except for the recurrence $T(n) = aT(n/b) + \Theta(n^c(log(2, n)^d)$. Use the Master Theorem, if possible, to output the asymptotic behavior of $T(n)$. So for instance, if your input is once again $(2, 2, 5, 0)$, you should output `Theta(n^5)`. If it's not possible you should output `NO ANSWER`.

Other points to remember:

1. You might be able to get some partial credit if your code runs only for some specific cases (e.g. only for $b = 2$). However, we encourage you to remember that if you have it for $b = 2$, then other "small" values for b should not be much harder.

2. You do **NOT** have to draw the recursion trees. See **Implementation** below for more details: we just need you to encode the trees in a specific way.

3. You are encouraged above everything else to think about these functions first, before starting to code. We gave you very specific kinds of recursions, of the form $T(n) = aT(n/b) + f(n)$ for an extremely restricted class of functions $f(n)$. What are you supposed to get as the asymptotic behavior for this class of functions?

4. Be careful about the Master Theorem, and remember when the "gaps" in the theorem arise.

5. Please post to Piazza for any further questions or clarifications, we'll do our best to answer.

Implementation. You may use whatever programming language you like, but we strongly recommend that you use `Python 3.6` and use the Python file next to this document (`extracredit.py`). You will find the `anytree` library useful for storing and working with trees. Regardless of what language you use, your program should read the input as specified.

An input to Part 1 would be of the form: `extracredit.py 1 r a b c d e`.

The first argument is the problem part number (here, 1). The other arguments encapsulate (r, a, b, c, d, e). Note the space separators. An input to Part 2 would be of the form: `extracredit.py 2 a b c d`, similar for parts 3 and 4.

For the output, if you are asked to output a tree and it's not an invalid input, we are looking for a single file that encodes the required tree. The easiest way to encode these trees is to output a file called `tree.json`, which would store the tree structure by means of nodes along with pointers to its children. Each node of

your tree should have two attributes, `label` and `value`. A sample `tree.json` is provided. You can use the method in file `extracredit.py` for exporting your tree as a JSON object. If your output contains a logarithm function you should print it a as a function with two arguments (though you will get generous partial credit for a reasonably good decimal approximation). The first argument is the base of the logarithm and the second one is the exponent. For instance `log(2,n)` is just $\log_2(n)$, and `log(3, 4)` is $\log_3(4)$. We would prefer that you simplify things like `log(2, 8)` to `3` whenever possible. If you get a logarithmic factor in your final Theta output, then you do not have to specify the base. For instance, since $\log_2(n) = \Theta(\log_3(n))$, you can just output `Theta(log n)` instead of `Theta(log(3, n))`.

Collaboration. You must work in groups of 2-4. You may not discuss your solution with any other group. You may use the internet and external resources for programming help *only*; you may not use other resources for algorithmic ideas.

Note that the internet has at least three or four purported "Master Theorem solvers". The ones that we found are all **incorrect**; they give you solutions in cases where you are not supposed to be able to use the theorem. You have been warned.

Submission instructions. This extra credit assignment is due by **11:59pm** on **Thursday, December 6**. We will **NOT** accept any late submissions.

Your group's submission must consist of a single file (such as `extracredit.py`) for the program, together with a `README` file in ASCII plain text with the following information:

1. Your group members
2. The intuition for your algorithm(s).
3. Instructions, if any, on how to run the code, if it differs from our given format.
4. High-level pseudocode.
5. Run-time analysis.

Once you are done, submit a zip file (extension `.zip`) to Gradescope.

We will run your algorithm on various test inputs (for all four parts), and check if your outputs are correct. Beware of edge cases!

We will also time your program; we reserve the right to terminate it if it seems to be unreasonably slow.