CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Problem Set 0 (preview)

Harvard SEAS - Fall 2021 *Due: Wed Sep. 8, 2020 (12 noon)*

The purpose of this problem set is to reactivate your skills in proofs and programming from CS20 and CS50. For those of you who haven't taken one or both those courses, the problem set can also help you assess whether you have acquired sufficient skills to enter CS120 in other ways and can fill in any missing gaps through self-study. To quickly assess your preparation for CS120 before entering the lottery, you should at least read through the problem set, and see if you understand what is being asked and the problems feel approachable. Actually completing the problem set (which is due September 8) may require several hours of effort, consulting background readings or the staff to refresh your memory, and/or attending review sessions or office hours during the first week of the semester. To receive a grade of "R" (ready to move on), you should demonstrate a solid understanding of the major concepts and skills listed in the non-starred problems and thereby construct solutions that are correct except possibly for minor errors. Doing so

For those of you who are wondering whether you should wait and take CS20 before taking CS120, we strongly encourage you to also complete the the CS20 Placement Self-Assessment. Some problems there that are of particular relevance to CS120 and are complementary to what is covered below are Problems 1 (discrete probability), 2 (counting), 4 (comparing growth rates), 9 (quantificational logic), and 11 (graph theory).

Motivation for the problems below: As we will see in the beginning of CS120, binary trees are a very useful data structure for storing and searching large datasets. These data structures are most efficient when the trees are roughly balanced, i.e. have depth $O(\log n)$ where n is the size of tree. In this problem, you'll see and analyze an example of a procedure for making binary trees more balanced. For simplicity, in this problem we won't be worrying about the functionality of the binary tree (e.g. how it used for searching), but when we perform balancing in the context of algorithmic applications, we will have to ensure that the functionality is maintained even during the balancing procedure.

1. (recursive data structures in Python, programming recursive functions, graph-theoretic understanding of trees) Below is a recursive Python data structure to represent a (rooted) binary tree, in which each node can potentially have a left child, a right child, a key, and a temporary integer value. (All of these attributes can be `Null`.) The keys will not play any role in this problem other to distinguish nodes from each other and illustrate what the algorithms are doing.

```
class BinaryTree:
  left: BinaryTree
  right: BinaryTree
  key: string
  temp: int
```

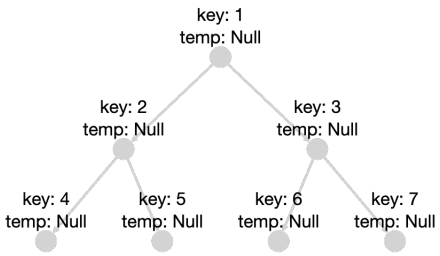
   In CS50, the concept of a Python `class` was not covered. Here we are using them in the same way as a `struct` in C, which was used extensively in CS50. An instance `T` of the

`BinaryTree` class is a *pointer* to a structure containing the four attributes above. These attributes can be accessed as `T.left`, `T.right`, `T.key`, and `T.temp`. Thus `T.left.key` is the key associated with `T`'s left child. Classes are more general than structs, in that they can also have private attributes and methods that operate on the attributes, allowing for object-oriented programming. However, you won't need that generality in this problem set.

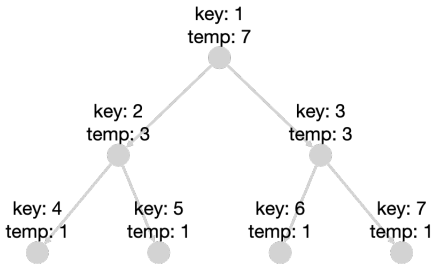
Recall that the *size* of a tree is the number of nodes in the tree, and the *depth* of a tree is the longest distance from the root to a leaf.

Write a recursive function `calculate_sizes(T: Tree)` that modifies `T` so that for every subtree `U` of `T`, `U.temp` equals the *size* of `U`.

For example, if `T` is the following tree:



then the result of `calculate_sizes(T)` should be as follows:



Your implementation of `calculate_sizes(T)` should run in time $O(n)$ on trees of size n; confirm for yourself that this is the case.

2. (proofs by induction, graph-theoretic understanding of trees) Prove by induction on depth that for every binary tree T, and natural numbers U, L such that $\text{size}(T) > U \geq 2L$, T has a proper subtree of size in the interval $[L, U]$.[1]

[1] A *proper* subtree of T is a subtree that is rooted at a node other than the root of T, i.e. a subtree whose nodes form a strict subset of the nodes of T.

3. (recursive data structures in Python, programming recursive functions, asymptotic growth rates) Implement your proof of Part 2 into an $O(n)$-time Python program `FindSubTree(T: BinaryTree, L: int, U: int)` such that whenever `size(T)`$>$ `U`$\geq$ `2L`,

   - `FindSubTree(T,L,U)` outputs a subtree `subT` of `T` of size in the interval [`L`,`U`], and
   - `FindSubTree(T,L,U)` removes `subT` from `T` (by replacing the pointer to `subT` in its parent with `Null`).

   `FindSubTree` is allowed to freely modify the `temp` fields in the nodes of `T` (so can safely run `calculate_sizes`), but should not modify the `key` fields.

4. (understanding Python code) Now consider the following function for "balancing" a binary tree by moving nodes around and adding new nodes.

```
def Balance(T: BinaryTree):
   calculate_sizes(T)
   if T.temp<=2:
     return T
   else:
     L = floor(T.temp/3)
     U = ceiling(2*T.temp/3)
     subT = FindSubtree(T,L,U)
     newT = BinaryTree()
     newT.left = Balance(T)
     newT.right = Balance(subT)
     return newT
```

   Note that when $n \geq 3$, we have $n > \lceil 2n/3 \rceil \geq 2 \cdot \lfloor n/3 \rfloor$, so the inputs to `FindSubtree` satisfy the precondition `size(T)`$>$ `U`$\geq$ `2L`.

   Draw the output of `Balance(T)` on the following tree `T`: (include a picture of a very imbalanced tree)

5. (asymptotic growth rates, finding and solving recurrences) Prove that `Balance` always returns a tree of depth $O(\log n)$ when given a tree of size of n. (Hint: write a recurrence for the maximum depth tree that `Balance` can return on a tree of size at most n.)

6. (*challenge) Show that the runtime of `Balance` is $O(n \log n)$.