COMP4019.G54AAD - Lab Session 3 – Complexity of Recursive Algorithms; Binary Search Trees

Xavier Carpent & Ian Knight

October 21, 2021

Complexity of Recursive Algorithms

Use the Master Theorem to determine the asymptotic complexity of the following recurrence relations:

- $T(n) = 3T(n/5) + 7$
- $T(n) = 4T(n/2) + n$
- $T(n) = 4T(n/2) + n^2$
- $T(n) = 4T(n/2) + n^3$
- $T(n) = 3T(n/5) + n$
- $T(n) = 2T(n/4) + \sqrt{n}$

Implementation of Binary Search Trees

In the programming language of your choice implement ("from scratch", not using libraries that already provide BST functionality) the BST data type and the algorithms for `search`, `in-order-traverse`, `insert`, and `delete`.

Complexity of BST algorithms

For each of the four operations you implemented, determine their worst-case running complexity by writing down a recurrence relation expressing the number of steps of computation, then solve it (using the Master Theorem if possible) to find their O complexity class.

Further algorithms on BSTs

Write down in pseudo-code algorithms to perform the followingtasks on BSTs (you can start by an initial simplistic direct implementation and then try to improve it):

- Merging of trees: given two BSTs, `t1` and `t2`, compute a tree `t = merge(t1, t2)` containing exactly the union of elements in `t1` and `t2`;

- Finding the middle element of the tree: if a tree `t` contains n elements, compute the value `m = middle(t)` that, in an ordered sequence of elements of `t`, has index $\lfloor \frac{n}{2} \rfloor$.

Hint: One idea to keep in mind in these and following tasks, is to add extra information to the nodes to make the specific task easier (as discussed in class).

Balancing

We have seen in class that `insert` and `delete` operations may modify the structure of a BST such that we cannot guarantee that the height is a logarithmic function of its number of elements (in other words, $h \notin O(\log n)$). This is bad news, because it means that the complexity of the operations on the BST are worse than the could be if we could keep it "*balanced*".

A binary search tree is called *balanced* if the lengths of any two paths from the root to a leaf differ by at most one. A slightly stronger notion is to require that the difference in the number of elements of the left and right children of every node is at most one. Can you see why these two definitions are not exactly equivalent? Find a tree that satisfies one but not the other.

One simple idea to keep a tree balanced is to *rebalance* it whenever the number of elements in the left and right subtrees differ by more than one, using the following procedure: suppose that we start with a balanced tree and either `insert` or `delete` an element causing the tree (or a subtree) to become unbalanced, for example because the left child now has two elements more than the right child. Consider this re-balancing algorithm:

1. Find and delete the maximum element y from the left subtree;
2. Replace the root element x with y;
3. Insert x in the right subtree.

What is the complexity of this algorithm? Is it sufficient to apply it (or its left-right symmetric) after every `insert`/`delete` operation to keep the tree balanced? If not, how can you modify it for that to be the case? What is the complexity of the new version?

Challenge

These questions are for an extra challenge regarding the complexity of recursive algorithms. They are more difficult than what you can expect of the exam. Only tackle them once you have finished the rest of the exercises.

1. Consider the following recurrence relation: $T(n) = T(n/2) + 2T(n/4)$, with $T(1) = 1$ and $T(2) = 2$. Can you use the master theorem to solve it? Try solving it using the substitution method. *Hint: explore it using specific values of n.* What about $T(n) = T(n/2) + T(n/4)$, with $T(1) = T(2) = 1$?
2. Consider the following recurrence relation: $T(n) = nT(n/2)$, with $T(1) = 1$. Can you use the master theorem to solve it? Can you manage to solve it using a different method? How large is the overal complexity?