Fundamental Algorithms, Sample Midterm, Fall 2022, Solutions

1.a. (**5 pts.**) For each pair of functions, circle the one that grows to be the asymptotically larger as n tends to infinity. If they remain within constant factors of each other, circle both.

i. $\boxed{10n^5}$ $\qquad 1,000,000n^4 + 7n^2 \log n$

ii. $\boxed{n^{1/\log n}}$ $\qquad \boxed{1}$

iii. $\boxed{3^n}$ $\qquad 2^n$

iv. $n^{\sqrt{n}}$ $\qquad \boxed{\sqrt{n}^{\,n}}$

v. $\boxed{\log n}$ $\qquad \log \log n$

b. (**5 pts.**) Suppose that f, g and h are positive functions on the positive integers. Suppose further that $f = o(g)$ and $h = \Omega(g)$. Show that $f = o(h)$.

Solution:

As $f = o(g)$, for each $c > 0$ there is an integer $n_c \geq 1$ such that for all $n \geq n_c$, $f(n) \leq c \cdot g(n)$. As $h = \Omega(g)$, there is a constant $d > 0$ and an integer $n_d \geq 1$ such that for all $n \geq n_d$, $g(n) \leq d \cdot h(n)$. Therefore, for all $n \geq \max\{n_c, n_d\}$, $f(n) \leq cd \cdot h(n)$. For each $b > 0$, define $c = b/d > 0$. Setting $n_e = \max\{n_c, n_d\}$, implies that for all $n \geq n_e$, $f(n) \leq cd \cdot h(n) = b \cdot h(n)$. As this holds for every $b > 0$, this is the statement that $f = o(h)$.

2. Use the recursion tree method to solve, as best you can, the following recurrence equations. It is OK to write $T(n) =$ some specific sum of n or $\log n$ terms or whatever if you do not recall how to add up the sequence you obtain (this will result in a small reduction in score).

a. (**5 pts.**)

$$\begin{aligned} R(1) &= 1 \\ R(n) &= 2 + R(n/4) \qquad n > 1 \text{ and } n \text{ is an integer power of 4.} \end{aligned}$$

Solution:

For $n > 1$:

There is one size $n = n/4^0$ problem with non-recursive cost 2;

there is one size $n/4 = n/4^1$ problem with non-recursive cost 2;

$\cdots$

there is one size $4 = n/4^{\log_4 n - 1}$ problem with non-recursive cost 2;

there is one size $1 = n/4^{\log_4 n}$ problem with non-recursive cost 1;

Thus $R(n)$, the total cost, is equal to $1 + 2 \cdot \log_4 n = 1 + \log_2 n$.
Check: This gives $R(1) = 1$ and $R(4) = 1 + 2 = 3$ both of which are correct.

b. (**5 pts.**)

$$S(0) = 1$$
$$S(n) = 3^n + 9S(n-1) \qquad n > 0$$

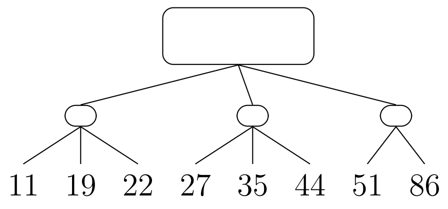
Solution:
There is one size n problem with non-recursive cost 3^n;
there are 9 size $n-1$ problems with combined non-recursive cost $9 \cdot 3^{n-1} = 3^{n+1}$;
$\ldots$
there are 9^{n-1} size 1 problems with combined non-recursive cost $9^{n-1} \cdot 3^1 = 3^{2n-1}$;
there are 9^n size 0 problems with combined non-recursive cost $9^n \cdot 3^0 = 3^{2n}$;

Thus, for $n \geq 0$, $S(n)$, the total cost, is equal to $3^n + 3^{n+1} + \ldots + 3^{2n} = (3^{2n+1} - 3^n)/2$.
Check: This gives $S(0) = 1$ and $S(1) = (3^3 - 3)/2 = 12$ both of which are correct.

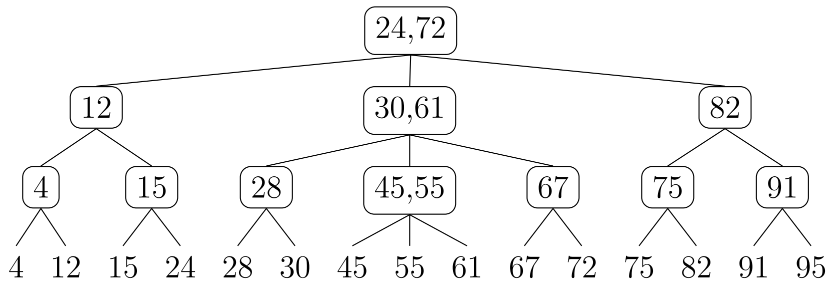
3. This question concerns 2–3 trees.

a. (**2 pts.**) What are the guides at the root for the 2–3 tree shown below?

Solution: 22, 44.

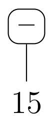


b. (**6 pts.**) Show the effect of the operations Delete(24) on the 2–3 tree shown below. Show the sequence of steps performed in carrying out this operation.

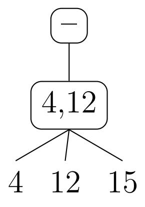


Solution:

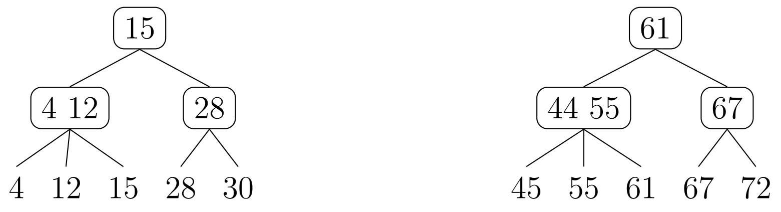
We start by finding and removing the leaf storing the item 24, yielding a 1-child node as shown below:



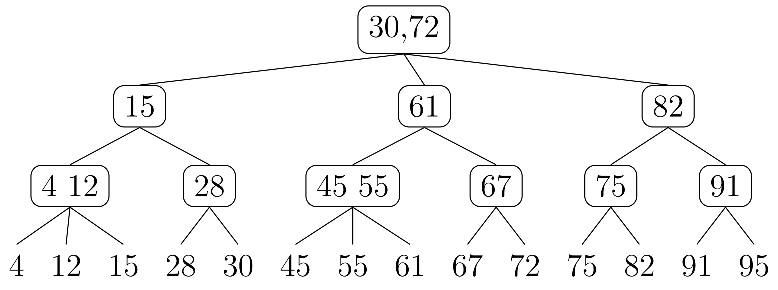
We then combine it with its 2-child sibling, yielding:



We now take a subtree from the sibling of the node with one child, thus:



The complete resulting tree is:



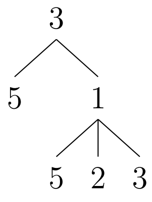
c. (**2pts.**) What are the exact minimum and exact maximum heights of a 2–3 tree storing 100 items (where the height of a single node tree is defined to be zero)?

Solution: Maximum: 6 ($= \lfloor \log_2 100 \rfloor$); minimum: 5 ($= \lceil \log_3 100 \rceil$).

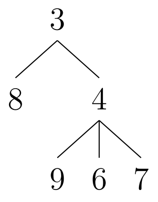
4.a. (**5 pts.**) Let T be an arbitrary tree so each vertex can have any number of children.

Suppose that there is a field $v.\mathrm{val}$ which already holds a non-negative integer value for each vertex. You are to write an algorithm which will use an additional field, $v.\mathrm{PthTo}$. For each node v your task is to compute the sum of the values on the path from the tree root to v including the value at node v; this value should be stored in the field $v.\mathrm{PthTo}$.

The following example tree shows the values to be computed.



Input tree
showing $.\mathrm{val}$ field



output tree showing
$.\mathrm{PthTo}$ field

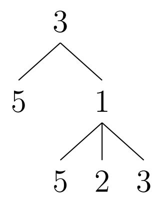
Please complete the following procedure. Remember to make an initial call.

```
PathCalc($v$, PthSoFar);

    $v.\mathrm{PthTo} \leftarrow v.\mathrm{val} + \mathrm{PthSoFar}$;

        **for** each child $w$ of $v$ **do**

            PathCalc($w$, $v.\mathrm{PthTo}$);

        **end** (* **for** *)

**end**
```

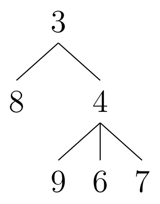
Initial call or a Driver procedure (**answer here too**): PathCalc$(T, 0)$

b. (**5 pts.**) Now suppose that we want to also compute in field $v.\,\mathrm{PthToLf}$ the maximum $x.\,\mathrm{PthTo}$ value over all the leaves in v's subtree. Augment your program for part (a) so as to compute this value too.

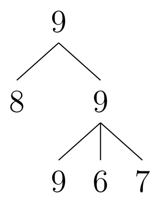
The following example tree shows the values to be computed.



Input tree showing $.\,\mathrm{val}$ field



output tree showing $.\,\mathrm{PthTo}$ field



output tree showing $.\,\mathrm{PthToLf}$ field

```
PathCalc($v$, PthSoFar);

    $v.\,\mathrm{PthTo} \leftarrow v.\,\mathrm{val} + \mathrm{PthSoFar}$;
    **if** $v$ is a leaf **then** $v.\,\mathrm{PthToLf} \leftarrow v.\,\mathrm{PthTo}$
                    **else**  $v.\,\mathrm{PthToLf} \leftarrow 0$;

    **for** each child $w$ of $v$ **do**

        PathCalc($w$, $v.\,\mathrm{PthTo}$);
        $v.\,\mathrm{PthToLf} \leftarrow \max\{v.\,\mathrm{PthToLf}, w.\,\mathrm{PthToLf}\}$

    **end** (* **for** *)

**end**
```

Initial call or a Driver procedure (**answer here too**): PathCalc($T, 0$)

5. (**10 pts.**) Let $S = \{e_1, e_2, \ldots, , e_n\}$ be a set of n distinct integers in a range large enough to mean that radix sort does not run in linear time. Give an expected $O(n)$ time algorithm to report all pairs (e, f) of integers in S such that $e = 2f$.

Solution:

We use a hash function with a table H of size n chosen uniformly at random from a universal family of hash functions.

All the integers in S are inserted into H. This takes (worst-case) $O(n)$ time. Then for integer $f \in S$, we compute $2f$ and check whether $2f = e$ for any value e already hashed to $H[h(2f)]$; if so, we report the pair (e, f). Note that there are an expected $O(1)$ items hashed to this location so this takes expected $O(1)$ time. Item f is not inserted at this location.

Overall, this algorithm takes expected $O(n)$ time.

6.a. (**4 pts.**) Suppose that you are maintaining some statistics regarding tennis players, identified by name. For each player, you need to store the number of games won. Suppose that there are n players. There are three basic operations: add a player, delete a player, and for a given player, increase its score (of games won). These all need to run in $O(\log n)$ time. What data structure would you use? Explain how it supports these three operations in $O(\log n)$ time.

Solution:

Use a 2-3 tree with each player being an item and its name being the key. An extra field is used to store the player's score. Adding and deleting a player are the standard insert and delete 2–3 tree operations, which run in $O(\log n)$ time. Changing a player's score is implemented by searching for the player using the 2–3 tree search operation, and then updating its score. This too takes $O(\log n)$ time.

b. (**6 pts.**) Suppose that you want to be able to determine in $O(\log n)$ time how many players won at least x games, given a query x. Explain how to modify your solution to part (a) so as to support this operation in addition to the three operations specified in part (a).

Now we use a second 2–3 tree over the set of scores. At each leaf we keep a count of the number of players with that score. A player addition requires an insertion to both trees; in the second tree, we are adding one to the number of players with score 0; if the new player is the only player with this score, then a leaf has to be inserted. A deletion is performed by finding the player in the first tree (with names as the key) and once the leaf storing the player is located, noting its score and deleting it in the first tree. Then, this score is searched for in the second tree, and the count of players with this score is reduced by 1. If this reduces the count to zero, then this item is deleted. An update of the score is performed as before on the first tree. On the second tree, this involves searching for the score, decrementing the count for that score (and if need be deleting that leaf), searching for the new score, incrementing the count for the latter score, or if not present inserting a leaf with a count of 1.

Finally the new query is carried out on the second 2–3 tree. To support this query, we need, at each internal node, to maintain the sum of the counts at the leaves in each subtree (this equals the number of players with scores stored in this subtree). These are readily maintained when an insertion or deletion is performed, while maintaining the $O(\log n)$ time performance; this is because the count at a node is just the sum of the counts for its children, and the only counts that might change are those for nodes on the search path and their siblings; there are $O(\log n)$ of these nodes in total.

To determine the number of players with a score of x or more, perform a search for the item with key x, and sum the counts for all the subtrees hanging to the right of this search path, and if there is an item with value x, then include the count for the item with value x. Note that there are $O(\log n)$ such subtrees, thus computing this sum takes an additional $O(\log n)$ time over and above the $O(\log n)$ time needed for the search.