University of Waterloo
CS240E, Winter 2025
Assignment 2

Due Date: Tuesday, February 4, 2024 at 5pm

Be sure to read the assignment guidelines (`https://student.cs.uwaterloo.ca/~cs240e/w25/assignments.phtml#guidelines`). Submit your solutions electronically to Crowdmark.

Grace period: Submissions made before 11:59PM on Feb. 4 will be accepted without penalty. Please note that submissions made after 11:59PM **will not be graded** and may only be reviewed for feedback.

Remark: For all questions on this assignment you **cannot compute logarithms or exponentiations in constant time**. If you need these, then your algorithm must compute them from scratch, and the time to do so must be counted when analyzing run-time.

Question 1 [1+1+7=9 marks]

Let $T(w)$ be the number of times that we print a '*' when calling Algorithm 1 with a bitstring w of length n. Now let $T^{\text{best}}(n)$, $T^{\text{worst}}(n)$, and $T^{\text{avg}}(n)$ be the best, worst, and average of this (over all bitstrings of length n).

a) Prove an O-bound on $T^{\text{best}}(n)$.

b) Prove an Ω-bound on $T^{\text{worst}}(n)$.

c) Show that $T^{\text{avg}}(n) \in O(T^{\text{best}}(n))$.

Algorithm 1: *silly-algo*(w, n)

```
Input: bitstring $w$ of length $n$
1 $i \leftarrow 0$, while $w[i] == 0$ do $i \leftarrow i + 1$
  // $w[i]$ is leftmost non-zero, possibly $
2 for $k = 1$ to $(i+1)^2$ do print "*"
```

Hint: If you find part (c) difficult, then for partial credit state a summation-formula for $T^{\text{avg}}(n)$, simplify it as much as possible, and give as tight an upper bound for it as you can manage. You may use without proof that $x \geq 4 \log x$ for $x \geq 16$. The course-notes have bounds for some summations that you may use without proof (but state the page-number).

Question 2 [2+5=7 marks]

Consider Algorithm 2, which is a variant of *quick-sort*.

Here, subroutine *use-one-key-comparisons* is unknown but uses exactly one key-comparison, and *partition* is as in class (and uses exactly n key-comparisons). Now let $T^{\text{best}}(n)$ and $T^{\text{worst}}(n)$ be the best and worst number of key-comparisons for an array of size n.

Algorithm 2: $mysteryQS(A, n \leftarrow A.size)$

```
if $n > 1$ then
    $i \leftarrow partition(A, n-1)$
    $mysteryQS(A[0, 1, \ldots, i-1])$
    $mysteryQS(A[i+1, \ldots, n-1])$
    for $j = 1$ to $i$ do
        for $k = i + 1$ to $n - 1$ do
            use-one-key-comparison()
```

a) Show that $T^{\text{worst}}(n) \in \Omega(n^2)$.

b) Show that $T^{\text{best}}(n) \in O(n^2 \log n)$.

Remark: Obviously (a) and (b) cannot both be tight. To keep the assignment shorter we did *not* ask you to get the tight bounds here.

Question 3 [6+3=9 marks]

a) Assume that you are given a non-empty list L that contains n key-value pairs in sorted order. Design an algorithm to build a binary search tree that contains exactly the items of L and that is *perfectly size-balanced*, i.e., for every vertex z we have $|size(z.left) - size(z.right)| \leq 1$. The run-time must be $O(n)$ and the algorithm should use $O(\log n)$ auxiliary space (i.e., space other than the input-list L and the output-tree T that you are building).

b) Prof. Quirky thinks that he can do the above with a comparison-based algorithm even if L is in unsorted order. Show that this is not possible.

Question 4 [3+3+3+6(+5)=15(+5) marks]

A *list-of-stacks* data structure consists of a (doubly-linked) list L of stacks $S_0, S_1, S_2, \ldots, S_\ell$ where stack S_i is stored as an array of capacity exactly 3^i. Initially L contains just S_0. The user always pushes and pops elements from the leftmost stack S_0. However, before any element can be pushed onto a stack S_i, we first check whether S_i is *full* (its size equals its capacity), and if so, move all its elements to S_{i+1}. (If S_{i+1} is full then we move its elements in turn, and so on.) Similarly, before any element can be popped from a stack S_i, we first check whether S_i is empty, and if so, fill it with 3^i items from S_{i+1}. (If S_{i+1} is empty, then we recursively fill it from S_{i+2}, and so on.) See Algorithms 3 and 4 for the pseudocodes.

Algorithm 3: *Mpush*(x)

```
S ← leftmost stack on L
i ← 0, c ← 1   //idx/capacity of S
while S is full do
    i ← i+1, c ← 3*c
    if S ≠ rightmost stack on L then
        S ← stack after S on L
    else S ← L.append(new stack)
// S is leftmost non-full stack
while i > 0 do
    S_next = S, i ← i−1, c ← c/3
    S ← stack before S on L
    for j = 1 to c do
        S_next.push(S.pop())
S.push(x)
```

Algorithm 4: *Mpop*()

```
S ← leftmost stack on L
i ← 0, c ← 1   //idx/capacity of S
while S is empty do
    i ← i+1, c ← 3*c
    if S ≠ rightmost stack on L then
        S ← stack after S on L
    else return "error: No items"
// S is leftmost non-empty stack
while i > 0 do
    S_next = S, i ← i−1, c ← c/3
    S ← stack before S on L
    for j = 1 to c do
        S.push(S_next.pop())
return S.pop
```

a) Argue that these algorithms do not crash in lines 11-12. In particular, why do we have always room to push $c = 3^i$ items onto S_{next} for *Mpush*? Why are there always at least 3^i items in S_{next} for *Mpop*? (Hint: What can you say about the size of stack S_i at any given time?)

b) Assume that you only push items onto the stack, you never pop. Show that the list-of-stacks uses $\Theta(n)$ space, where n is the current number of items that are stored.

c) Briefly say why the space is *not* $\Theta(n)$ in the worst case if we also pop items. Then describe how you would modify the algorithm so that the space is always $\Theta(n)$ even when we pop items. Only state the idea and analyze the space-requirement.

d) For a list-of-stacks $\{S_0, \ldots, S_\ell\}$, define $\Phi = \sum_{i=0}^{\ell} size(S_i) \cdot (\ell - i)$. Assume that at least one of the stacks is not full. Argue that using Φ as potential function (and with a suitable choice of time units) the amortized run-time for *Mpush* is $O(\log n)$.

e) **(Bonus)** Argue that the amortized run-time of *Mpush* is $O(\log n)$ even when all stacks are full. (Hint: You will need to define a different potential function—finding it is part of the problem.)

While a correct answer to e) likely contains a correct answer to d), for ease of grading please write a separate answer to d) and reuse it here as needed.

Question 5 [2+3+2+3=10 marks]

Motivation: Scapegoat trees as defined in class store at each node z the size of the subtree rooted at z. This is not actually required; this assignment will guide you towards a variation that operates without storing the size of the subtree.

Define a *light scapegoat tree* to be a binary search tree that is *height-balanced*, by which we mean that the height is at most $\lfloor \log_{4/3} n \rfloor$ (or equivalently, every node has depth at most $\lfloor \log_{4/3} n \rfloor$). You may assume that a light scapegoat tree has a field *size* with its number of items. However, a light scapegoat tree does not store its height, and a node knows *nothing* except its key and left and right subtree. (In particular it knows neither its depth, nor its parent, nor the size of its subtree, nor the height of its subtree.)

Algorithm 5 gives (incomplete) pseudo-code to insert in such a tree:

Algorithm 5: *LightScapegoatTree::insert*(k, v)

```
  // Current tree is height-balanced
1 $z \leftarrow$ BST::insert$(k, v)$
2 if height-unbalanced$(z)$ then
3     $p \leftarrow$ lowest-small-ancestor$(z)$
4     completely rebuild the subtree at $p$ as a perfectly size-balanced tree
```

a) Show that (after inserting the new leaf z) the tree can be height-unbalanced only if the depth of z is too big.

b) Design algorithm *height-unbalanced*(z) to test whether the tree is now no longer height-balanced. The run-time must be $O(\log n)$, where n is the current size of the tree.

c) Show that if the tree is not height-balanced after *BST::insert*, then there exists an ancestor x of leaf z such that

$$size(x) < \left(\tfrac{4}{3}\right)^{d(x,z)}.$$

Here $size(x)$ denotes the number of items in the subtree rooted at x, and $d(x, z)$ denotes the distance from z to x, i.e., the number of levels that x is above z. (So $d(z, z) = 0$, $d(parent(z), z) = 1$, etc.).

d) Sub-routine *lowest-small-ancestor*(z) does the following: Find an ancestor x of z with $size(x) < \left(\frac{4}{3}\right)^{d(x,z)}$, and among all those, return the one that minimizes $d(x, z)$. You need **not** say how to implement this (it is not easy to do efficiently).

Argue that the rest of the insertion-routine is correct. Thus, show that after rebuilding the subtree (at the node p returned by *lowest-small-ancestor*) to be perfectly size-balanced, the resulting binary search tree is height-balanced.

For all parts, you may use results of previous parts even if you did not prove them.

Continuing the "motivation": We are *not* asking you to show that *LightScapegoatTree::insert* has $\Theta(\log n)$ amortized time, but with the above results it would be very easy to do so using the potential function from class.