Homework #2

(Due: Apr 20)

Task 1. [70 Points] Yet Another Quicksort Variant[1]

Consider the quicksort variant given in Figure 1.

QUICKSORT($A[1:n]$, n)

Input: An array $A[1:n]$ of n distinct numbers.

Output: $A[1:n]$ with its numbers sorted in increasing order of value.

1. ***if*** $n < 2$ ***then return***
2. ***else***
3. $x \leftarrow A[1]$
4. rearrange the numbers of $A[1:n]$ such that
   - $A[k] = x$ for some $k \in [1, n]$,
   - $A[i] < x$ for each $i \in [1, k-1]$,
   - $A[i] > x$ for each $i \in [k+1, n]$,
   - $A[1] < A[i]$ for each $i \in [2, n]$,
   - $A[n] > A[i]$ for each $i \in [1, n-1]$,
5. QUICKSORT($A[2:k-1]$, $k-2$)
6. QUICKSORT($A[k+1:n-1]$, $n-k-1$)
7. ***return***

Figure 1: [Task 1] A variant of standard quicksort algorithm that excludes the smallest and the largest numbers (in addition to the pivot) in the input array from recursive calls.

Given an input of size n, in this task we will analyze the average number of element comparisons (i.e., comparisons between two numbers of the input array) performed by this algorithm over all $n!$ possible permutations of the input numbers. We will assume that the partitioning algorithm is *stable*, i.e., if two numbers p and q end up in the same partition and p appears before q in the input, then p must also appear before q in the resulting partition.

(*a*) [**5 Points**] Show how to implement step 4 of Figure 1 to get a stable partitioning of the numbers in $A[1:n]$ using only $2\left(n + \frac{1}{n} - 2\right)$ element comparisons on average, where the average is taken over all $n!$ possible permutations of the input numbers.

(*b*) [**10 Points**] Let t_n be the average number of element comparisons performed by the algorithm given in Figure 1 to sort $A[1:n]$, where $n \geq 0$ and the average is taken over

[1] Taras Kolomatski helped in setting this problem.

all $n!$ possible permutations of the numbers in A. Show that

$$t_n = \begin{cases} 0 & \text{if } n < 2, \\ 1 & \text{if } n = 2, \\ 2\left(n + \frac{1}{n} - 2\right) + \frac{2}{n}\sum_{k=0}^{n-2} t_k & \text{otherwise.} \end{cases}$$

(*c*) [**15 Points**] Let $T(z)$ be a generating function for t_n:

$$T(z) = t_0 + t_1 z + t_2 z^2 + \ldots + t_n z^n + \ldots \quad \ldots$$

Show that $T'(z) = \frac{2z}{1-z}\, T(z) + \frac{2z(1+z)}{(1-z)^3}$.

(*d*) [**20 Points**] Solve the differential equation from part (*c*) to show that

$$T(z) = F(z)G(z),$$

where, $F(z) = \sum_{k=0}^{\infty}\left(\sum_{j=0}^{k}\frac{(-2)^{k-j}(j+1)}{(k-j)!}\right)z^k$, and $G(z) = \sum_{k=2}^{\infty}\frac{2}{k}\left(\frac{2^{k-2}}{(k-2)!} + 2\sum_{j=0}^{k-3}\frac{2^j}{j!}\right)z^k$.

(*e*) [**15 Points**] Use your solution from part (*d*) to show that for $n \geq 0$,

$$t_n = \sum_{k=0}^{n-2}\left(\frac{2}{n-k}\right)\left(\sum_{j=0}^{k}\frac{(-2)^{k-j}(j+1)}{(k-j)!}\right)\left(\frac{2^{n-k-2}}{(n-k-2)!} + \sum_{j=0}^{n-k-3}\frac{2^{j+1}}{j!}\right).$$

Compute the numerical value of t_n for $0 \leq n \leq 10$.

(*f*) [**5 Points**] Use your solution from part (*e*) to show that $t_n = \Theta(n \log n)$.

Task 2. [**50 Points**] **Tiling an** $n \times 3$ **Grid**

Given an $n \times 3$ grid $G[n]$ for some integer $n \geq 0$, your task is to write a program that counts the number of different ways one can entirely cover the grid using only the four tile types (a, b, c, d) shown in the top row of Figure 2 but always avoiding the forbidden pattern shown in the middle row. You are not allowed to rotate or flip the tiles. Also tiles covering G must not overlap.

Now answer the following questions.

(*a*) [**10 Points**] Give a recursive algorithm for printing each of the different ways one can cover $G[n]$. Assuming that printing each tiling requires $\Theta(n)$ time, make sure and prove that your algorithm's running time, $T(n) = \Theta(n \times c_n)$, where c_n is the number of different ways one can tile $G[n]$.

(*b*) [**5 Points**] We need to know c_n to predict $T(n)$. Argue that c_n can be described using the following recurrence.

$$c_n = \begin{cases} 1 & \text{if } n < 2, \\ 2 & \text{if } n = 2, \\ 5 & \text{if } n = 3, \\ c_{n-1} + c_{n-2} + 2c_{n-3} & \text{otherwise.} \end{cases}$$

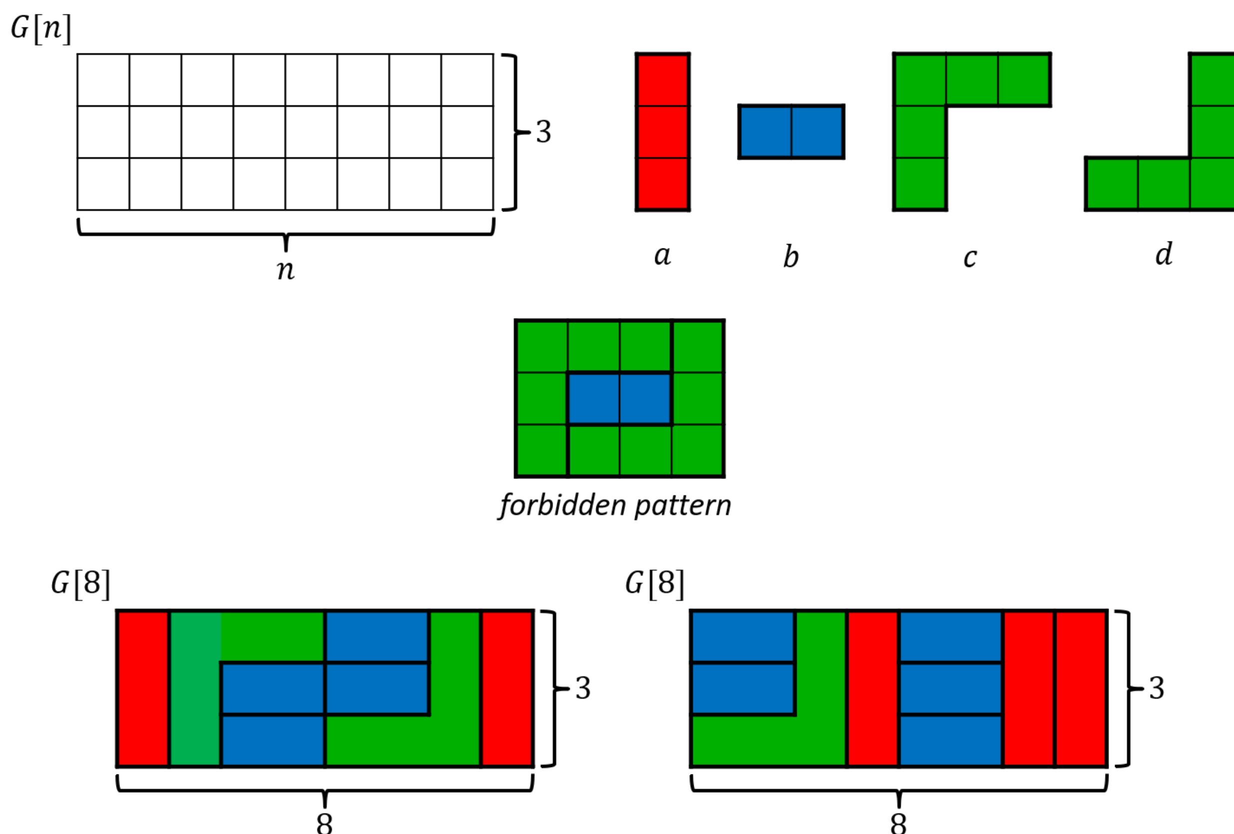

Figure 2: [Task 2] Given an $n \times 3$ grid $G[n]$ in how many ways can one entirely cover the grid (without any overlaps) using only the four tile types (a, b, c, d) shown in the top row but always avoiding the pattern shown in the middle row? The bottom row shows two different ways of covering a $G[8]$.

(c) [**20 Points**] Use a generating function to show that

$$c_n = \frac{1}{21}\left(3 \times 2^{n+2} - \left(\frac{\sqrt{-3}}{2} - \frac{9}{2}\right)\left(-\frac{1}{2} + \frac{\sqrt{-3}}{2}\right)^n - \left(-\frac{\sqrt{-3}}{2} - \frac{9}{2}\right)\left(-\frac{1}{2} - \frac{\sqrt{-3}}{2}\right)^n\right).$$

(d) [**10 Points**] Simplify the expression for c_n from part (c) to show that

$$c_n = \frac{1}{21}\left(3 \times 2^{n+2} - \sqrt{3} \times \sin\left(\frac{4n\pi}{3}\right) + 9 \times \cos\left(\frac{4n\pi}{3}\right)\right).$$

(e) [**5 Points**] Simplify the expression for c_n from part (d) even further to show that

$$7c_n = \begin{cases} 2^{n+2} + 3 & \text{if } n \bmod 3 = 0, \\ 2^{n+2} - 1 & \text{if } n \bmod 3 = 1, \\ 2^{n+2} - 2 & \text{if } n \bmod 3 = 2. \end{cases}$$

Task 3. [60 Points] A Priority Queue with Decrease-Keys

Consider the following heap data structure $\mathcal{H}$ that supports INSERT, MINIMUM, EXTRACT-MIN, DECREASE-KEY, and UNION operations.

Heap $\mathcal{H}$ is a collection of trees stored in two separate doubly linked lists $\mathcal{H}.\mathcal{D}_{new}$ and $\mathcal{H}.\mathcal{D}_{old}$. The linked list $\mathcal{H}.\mathcal{D}_{new}$ stores only singleton trees each containing a node newly inserted into $\mathcal{H}$ which is waiting for further processing. The other linked list $\mathcal{H}.\mathcal{D}_{old}$ stores all other trees, e.g., subtrees cut during DECREASE-KEY operations and trees generated by UNION operations.

A pointer, called a *min pointer*, is maintained that always points to the root of a tree with the smallest key in $\mathcal{H}.\mathcal{D}_{new} \cup \mathcal{H}.\mathcal{D}_{old}$.

Similar to the heap data structures we saw in the class, LINK(x, y) links two trees rooted at nodes x and y to form a single tree by making the node with the larger key the leftmost child of the node with the smaller key (ties are broken arbitrarily). However, unlike in those data structures x and y do not need to have the same rank.

By n we will denote the total number of items (i.e., nodes) currently in $\mathcal{H}$.

While $\mathcal{H}.\mathcal{D}_{new}$ has no restriction on how many trees it can store, $\mathcal{H}.\mathcal{D}_{old}$ is restricted to store fewer than $\lceil \log n \rceil$ trees.

The following procedure links all trees in $\mathcal{H}.\mathcal{D}_{new} \cup \mathcal{H}.\mathcal{D}_{old}$ to form a single tree.

CONSOLIDATE($\mathcal{H}$). Extract and put every tree from $\mathcal{H}.\mathcal{D}_{new}$ into an initially empty FIFO queue Q. Set $\mathcal{H}.\mathcal{D}_{new} \leftarrow \emptyset$. Then perform the following steps until the number of trees in Q reduces to one: dequeue two trees from Q, link them, and enqueue the new tree into Q. Let $\mathcal{T}_{new}$ be the final tree in Q. Now extract all trees from $\mathcal{H}.\mathcal{D}_{old}$, sort them in non-decreasing order of the keys stored at their roots, and push them in that order into an initially empty stack S. Set $\mathcal{H}.\mathcal{D}_{old} \leftarrow \emptyset$. Now perform the following steps until the number of trees in S reduces to one: pop two trees from S, link them, and push the new tree into S. Let $\mathcal{T}_{old}$ be the final tree in S. Now link $\mathcal{T}_{new}$ and $\mathcal{T}_{old}$ to form a new tree $\mathcal{T}$. Put $\mathcal{T}$ into $\mathcal{H}.\mathcal{D}_{old}$.

The various heap operations are performed as follows.

INSERT($\mathcal{H}$, x). Create a singleton tree containing only node x and insert that tree into $\mathcal{H}.\mathcal{D}_{new}$. Update the *min pointer*.

MINIMUM($\mathcal{H}$). Return the key stored at the root of the tree pointed to by the *min pointer*.

EXTRACT-MIN($\mathcal{H}$). First remove the root of the tree pointed to by the *min pointer*. Let that root be x and let r be the number of children of x. Let S be an empty stack. Now scan the children of x from left to right, and let $y_1, y_2, y_3, \ldots, y_r$ be those children in that order. Now if $r > 1$ then for each i from 1 to $\lfloor \frac{r}{2} \rfloor$ link y_{2i-1} and y_{2i} and push the resulting tree into S. If r is odd then push y_r into S at the end. Now perform the following steps until the number of trees in S reduces to one: pop the top two trees from S, link them, and push the new tree into S. Let $\mathcal{T}$ be the final tree in S. Insert $\mathcal{T}$ into $\mathcal{H}.\mathcal{D}_{old}$. Then call CONSOLIDATE($\mathcal{H}$). Return the key stored at node x.

DECREASE-KEY($\mathcal{H}$, x, k). Decrease the key of x to k. If x is not a root then (i) cut x's subtree from its parent y, (ii) cut the subtree rooted at x's leftmost child z from x, (iii) let z's subtree take x's subtree's position as a child of y, and (iv) put rest of x's subtree (i.e., without the subtree

rooted at z) as a separate tree in $\mathcal{H}.\mathcal{D}_{old}$. Now if $\mathcal{H}.\mathcal{D}_{old}$ holds $\lceil \log n \rceil$ trees, call CONSOLIDATE($\mathcal{H}$). Update the *min pointer* as necessary.

UNION($\mathcal{H}_1$, $\mathcal{H}_2$). Suppose $\mathcal{H}_1$ contains no more keys than $\mathcal{H}_2$ (if not, swap the roles of $\mathcal{H}_1$ and $\mathcal{H}_2$). Call CONSOLIDATE($\mathcal{H}_1$) to link all trees of $\mathcal{H}_1$ into a single tree $\mathcal{T}_1$. Add $\mathcal{T}_1$ to $\mathcal{H}_2.\mathcal{D}_{old}$ and destroy $\mathcal{H}_1$. If now $\mathcal{H}_2.\mathcal{D}_{old}$ holds $\lceil \log n \rceil$ trees, call CONSOLIDATE($\mathcal{H}_2$). Update the *min pointer* as necessary.

Clearly, the worst-case cost of MINIMUM($\mathcal{H}$) is $\mathcal{O}(1)$. We will analyze the amortized costs of all other operations above. We will use the following credit assignments, where c, c_{new} and c_{old} are positive constants.

- (i) We store c_{new} credits at the root of each tree in $\mathcal{H}.\mathcal{D}_{new}$.
- (ii) We store $c_{old} \log\log n$ credits at the root of each tree in $\mathcal{H}.\mathcal{D}_{old}$.
- (iii) When we link two subtrees rooted at nodes x and y making y the leftmost child of x, we store $\log(n_x + n_y) - \log n_y$ credits with the link (i.e., edge) connecting x and y, where n_x and n_y are the number of nodes in the subtrees rooted at nodes x and y, respectively, at the time they were linked.
- (iv) We store $c \log n$ credits with $\mathcal{H}$ itself.

Observe that the credits under items (ii) and (iv) above depend on n which is the number of nodes currently in $\mathcal{H}$, but those under items (i) and (iii) do not.

Now answer the following questions.

- (a) [**10 Points**] Show that CONSOLIDATE($\mathcal{H}$) is free provided the number of trees in $\mathcal{H}.\mathcal{D}_{old}$ is $\Omega\left(\frac{\log n}{\log\log n}\right)$ at the time of consolidation, otherwise its amortized cost is $\mathcal{O}(\log n)$.
- (b) [**5 Points**] Observe that the credit stored at the root of each tree in $\mathcal{H}.\mathcal{D}_{old}$ depends on n which is the number of nodes currently in $\mathcal{H}$. That means even if you do not change anything in $\mathcal{H}.\mathcal{D}_{old}$ but the number of nodes in $\mathcal{H}$ increases from n to some $n' > n$, the credit stored at each root of $\mathcal{H}.\mathcal{D}_{old}$ must also change from $c_{old} \log\log n$ to $c_{old} \log\log n'$. Prove that for any $n' \in (n, 2n]$, you can make this change for all tree roots in $\mathcal{H}.\mathcal{D}_{old}$ adding only a total of $\mathcal{O}(1)$ credits, i.e., $r \cdot c_{old} \cdot (\log\log n' - \log\log n) = \mathcal{O}(1)$, where r is the number of trees in $\mathcal{H}.\mathcal{D}_{old}$.
- (c) [**7 Points**] Show that the amortized cost of INSERT($\mathcal{H}$, x) is $\mathcal{O}(1)$. Remember that you will have to account for the credit changes elsewhere in the data structure that depend on n.
- (d) [**15 Points**] Show that the amortized cost of EXTRACT-MIN($\mathcal{H}$) is $\mathcal{O}(\log n)$.
- (e) [**15 Points**] Show that the amortized cost of DECREASE-KEY($\mathcal{H}$, x, k) is $\mathcal{O}(\log\log n)$.
- (f) [**8 Points**] Show that the amortized cost of UNION($\mathcal{H}_1$, $\mathcal{H}_2$) is $\mathcal{O}(\log\log n)$, where n is the total number of nodes in the combined heap.