Homework #2  
( Due: Nov 1 )

Task 1. [ 80 Points ] Average Case Analysis of 3-way Quicksort

Consider the 3-way quicksort algorithm given in Figure 1.


Perform the following steps if $n > 1$, return otherwise:


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) [ 8 Points ] Show how to implement steps 1 and 2 of Figure 1 to get a stable partitioning of $A[1:n]$ using only $2n − \text{rank}(\min(x,y)) − 2$ element comparisons, where rank($u$) gives you the number of entries in $A[1:n]$ with value not larger than $u$.

(b) [ 12 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 $n \geq 1$ and the average is taken over all $n!$ possible permutations of the numbers in $A$. Show that

$$t_n = \begin{cases} 0 & \text{if } n < 2, \\ \frac{5n-7}{3} + \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (t_{i-1} + t_{j-i-1} + t_{n-j}) & \text{otherwise}. \end{cases}$$

(c) [ 20 Points ] Give an algorithm that can compute $t_n$ in $O(n)$ time and $O(1)$ space, where $n \geq 1$ is an integer.
(d) [20 Points] Let $T(z)$ be a generating function for $t_n$:

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

Show that $T''(z) = \frac{6}{(1-z)^2} T(z) + \frac{2}{(1-z)^3} + \frac{10z}{(1-z)^4}$.

(e) [20 Points] Solve the differential equation from part (d) to express the precise value of $t_n$ in terms of $n$.

Task 2. [100 Points] Amortized Analysis of a Priority Queue

In this task we will consider a priority queue $Q$ that supports $\text{Insert}$ and $\text{Extract-Min}$ operations. An $\text{Insert}(Q, x)$ operation inserts an item $x$ into $Q$, and $\text{Extract-Min}(Q)$ deletes and returns the item with the smallest key from $Q$. We will assume for simplicity that all values stored in $Q$ are distinct.

The structure of $Q$ at any given time is determined by three constants: $\mu_{\text{max}} > 1$, $\mu \in (1, \mu_{\text{max}})$ and $\alpha \in (1, 2]$, where $\mu_{\text{max}}$ and $\alpha$ remain fixed throughout the lifetime of $Q$, but $\mu$ may change when the data structure is reconstructed periodically.

The data structure consists of $L + 1$ levels, where $L = \log_{\mu_{\text{max}}} (N_0)$ and $\frac{N_0}{2}$ is the number of items in $Q$ at the time of its last reconstruction. The data structure is reconstructed periodically in order to ensure that $\frac{1}{4} N_0 \leq N \leq \frac{3}{4} N_0$ always holds, where $N$ is the number of items currently in $Q$.

Let $n_i = \left\lfloor \mu^\alpha \right\rfloor$ for $0 \leq i \leq L$. Level 0 consists of two buffers $F_0$ and $S_0$, where $|F_0|, |S_0| \leq n_0$.

![Figure 2: Structure of the priority queue in task 2. Intuitively, float sets store items that are on their way up (i.e., floating) and sink sets store items that are on their way down (i.e., sinking). The arrow inside each box shows this general direction of movements.](image-url)
Figure 3: Pictorial depiction of Invariants 1–4. An arrow directed from box $B_1$ to box $B_2$ means that the key of every item in box $B_1$ is smaller than the key of every item in box $B_2$.

$1 \leq i \leq L$, level $i$ consists of a sink set $S_i$ with $|S_i| \leq n_i$, and at most $k_i = \lceil \frac{n_i}{n_{i-1}} \rceil + 1$ float sets $F_{i,1}, F_{i,2}, \ldots, F_{i,k_i}$, where $|F_{i,1}| < 2n_{i-1}$ and $n_{i-1} \leq |F_{i,j}| < 2n_{i-1}$ for $2 \leq j \leq k_i$. See Figure 2.

The entire priority queue is stored in a linear array $A$ as follows. The first $n_0$ locations of $A$ are reserved for $S_0$ and the next $n_0$ locations for $F_0$. Then space is reserved for the sink set and float sets of level 1, followed by those of level 2, then level 3, and so on. For any level $i \in [1, L + 1]$, first $n_i$ locations are reserved for $S_i$ followed by $2n_{i-1}$ locations for each of the $k_i$ possible float sets of level $i$. The float sets are not necessarily stored in sorted order\(^1\), but they are always linked together to form an ordered linked list.

The following four invariants are always maintained (see Figure 3 for a pictorial depiction).

**Invariant 1** For $1 \leq i \leq L$ and $1 \leq j < k_i$, keys in $F_{i,j}$ are smaller than those in $F_{i,j+1}$.

**Invariant 2** For $1 \leq i \leq L$ and $1 \leq j \leq k_i$, keys in $F_{i,j}$ are smaller than those in $S_i$.

\(^1\) e.g., $F_{i,j+1}$ may be necessarily be stored next to $F_{i,j}$ and it may appear before or after $F_{i,j}$
Invariant 3 For $1 \leq i < L$ and $1 \leq j \leq k_i$, keys in $F_{i,j}$ are smaller than those in $F_{i+1,1}$.

Invariant 4 Keys in $F_0$ are smaller than those in $S_0$ and $F_{1,1}$.

The item with the largest key in $F_{i,j}$ will be called a splitter, and will be denoted by $f_{i,j}$.

The Insert and Extract-Min operations internally use the following two recursive operations.

- **Lift**$(i, V)$: Extracts the $n_i$ items with the smallest keys from level $i + 1$ (and below), and returns them in $V$.

- **Sink**$(i, V)$: Inserts the $n_i$ items stored in $V$ (each with a key larger than all keys in the float sets of level $i$) into level $i + 1$.

The Lift and Sink operations work as follows.

- **Sink**$(i, V)$: First we sort the items in $V$. Then we distribute those items among the float sets of level $i + 1$ by simultaneously scanning the sorted items in $V$ and visiting the float sets $F_{i+1,1}, F_{i+1,2}, \ldots, F_{i+1,k_{i+1}}$ in that linked list order. When visiting $F_{i+1,j}$ for some $j < k_{i+1}$ we keep copying items from $V$ (in sorted order) to the end of $F_{i+1,j}$ until we encounter an item with key larger than the key of the splitter $f_{i+1,j}$, and at that point we move to $F_{i+1,j+1}$ and continue copying. Items with keys larger than the largest key in the last float set are inserted into $S_{i+1}$. If at any point a float set overflows (i.e., contains $2n_i$ items), it is split into two float sets of size $n_i$ each. If the number of float sets after the split does not exceed $k_{i+1}$, the new float set is stored in any empty float set slot for that level and the linked list is updated accordingly. Otherwise, we move the at most $2n_i - 1$ items of the last float set to $S_{i+1}$, store the new float set in the newly freed space and update the linked list. If $S_{i+1}$ itself overflows because of the insertion, i.e., it contains more than $n_{i+1}$ items, we move the $n_{i+1}$ items with the largest $n_{i+1}$ keys from $S_{i+1}$ to a temporary array $V'$ leaving the remaining at most $n_{i+1}$ items in $S_{i+1}$. We then call Sink$(i + 1, V')$ to push the items in $V'$ into level $i + 2$.

- **Lift**$(i, V)$: If $F_{i+1,1}$ has at least $n_i$ items, we sort those items, copy the $n_i$ items with the smallest keys to $V$, and leave the remaining items (if any) in $F_{i+1,1}$.

If $F_{i+1,1}$ has $m < n_i$ items but $F_{i+1,2}$ is nonempty (i.e., has at least $n_i$ items), we first remove $F_{i+1,1}$ by moving all its items to $V$. Then $F_{i+1,2}$ takes the role of $F_{i+1,1}$. The new $F_{i+1,1}$ has between $n_i$ and $2n_i - 1$ items. We sort those items, move the $n_i - m$ items with the smallest keys to $V$ and leave the remaining items in this new $F_{i+1,1}$.

If $F_{i+1,1}$ is the only nonempty float set in level $i + 1$, and it has $m < n_i$ items, we move those $m$ items to $V$ leaving $F_{i+1,1}$ empty, and recursively extract $n_{i+1}$ items with the smallest keys from level $i + 2$ by calling Lift$(i + 1, V')$ where $V'$ is a temporary array into which the extracted items are copied. Let $m' (\leq n_{i+1})$ be the number of items in $S_{i+1}$. We now move all items of $S_{i+1}$ to $V'$, sort the items in $V'$, move the $n_i - m$ items with the smallest keys from $V'$ to $V$, and move the $m'$ items with largest keys to $S_{i+1}$. The remaining $n_{i+1} - n_i + m < n_{i+1} \leq k_{i+1}n_i$ items are distributed among the float sets of level $i + 1$ with $F_{i+1,1}$ getting at most $n_i$ items and at most $k_{i+1} - 1$ other float sets getting $n_i$ items each.

Now the Insert and Extract-Min operations are performed as follows.
**Insert** (Q, x): We compare the key k of item x with the largest key in F₀. If k is larger, we simply insert x into S₀, otherwise we insert it into F₀ and move the item with the largest key in F₀ to S₀. If S₀ now contains n₀ items we empty it by calling Sink(0, S₀).

**Extract-Min** (Q): If F₀ is empty, we call Lift(0, F₀) to fill F₀ with n₀ items, and keep in F₀ the n₀ items with the smallest keys among F₀ and S₀ and leave the remaining items in S₀. We then extract and return the item with the smallest key from F₀.

We reconstruct Q periodically in order to make sure that N₀ = Θ(N) always holds, where \( \frac{N₀}{2} \) is the number of items in the data structure at the time of the latest rebuild and N is the number of items currently in Q. The structure is rebuilt right after the \( \frac{N₀}{4} \)-th Insert/Extract-Min operation is performed on it. First we find the largest value \( \mu < \mu_{\text{max}} \) such that \( \mu^l = 2N \) holds for some integer l, and this value of l is our new L meaning that the rebuilt priority queue will have \( l + 1 = L + 1 \) levels. We first sort the N items currently in Q. Then we rebuild Q top-down starting from level 0 and going up to level L while maintaining Invariants 1–4 as follows. We leave F₀ and S₀ empty. Then for each level \( i \in [1, L] \), the sink set Sᵢ and Fᵢ,₁ will be left completely empty while exactly \( nᵢ₋₁ \) items will be stored in each Fᵢ,j for \( 2 \leq j \leq kᵢ \). The remaining items are stored in level L such that sink set Sₗ is empty while Fₗ,₁ contains at most \( nₗ₋₁ \) item and at most \( kₗ \) of the remaining float sets of level L contains \( nₗ₋₁ \) items each.

Now answer the following questions.

(a) [ 4 Points ] Argue that Q uses only \( \Theta(N) \) space.

(b) [ 14 Points ] Show that both Lift( i, V ) and Sink( i, V ) maintain invariants 1–3.

(c) [ 6 Points ] Use your results from part (b) to argue that both Insert( Q, x ) and Extract-Min( Q ) operations maintain invariants 1–4.

(d) [ 14 Points ] What is the worst-case running time of Lift( i, V ) without considering any recursive use of this operation? What about Sink( i, V )?

(e) [ 6 Points ] Use your results from part (d) to find the worst-case time needed for performing the Insert( Q, x ) and Extract-Min( Q ) operations.

(f) [ 36 Points ] Argue that the amortized running time of Lift( i, V ) is only \( O(nᵢ \log nᵢ) \) without considering any recursive use of this operation. Show the same for Sink( i, V ).

(g) [ 20 Points ] Use your results from part (f) to show that Argue that the amortized running time of and Insert( Q, x ) operation is only \( O(\log N) \). Show that same for Extract-Min( Q ).