Homework #2  
( Due: Nov 8 )

Task 1. [ 80 Points ] Average Case Analysis of Median-of-3 Quicksort
Consider the median-of-3 quicksort algorithm given in Figure 1.

Figure 1: A variant of standard quicksort algorithm that uses the median of the first three numbers in its input (sub-)array as the pivot.

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) [ 10 Points ] Show how to implement steps 4 and 5 of Figure 1 to get a stable partitioning of the numbers in $A[1:n]$ using only $n - \frac{1}{3}$ 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 all $n!$ possible permutations of the numbers in $A$. Show that
\( t_n = \begin{cases} 
0 & \text{if } n < 2, \\
1 & \text{if } n = 2, \\
\frac{6}{n(n-1)(n-2)} \sum_{k=1}^{n} (k-1)(n-k)(t_{k-1} + t_{n-k}) & \text{otherwise.}
\end{cases} \)

(c) [20 Points] Let \( T(z) \) be a generating function for \( t_n \):
\[
T(z) = t_0 + t_1z + t_2z^2 + \ldots + t_nz^n + \ldots .
\]
Show that \( T'''(z) = \frac{12}{(1-z)^2} T'(z) - \frac{8}{(1-z)^4} + \frac{24}{(1-z)^5} \).

(d) [20 Points] Solve the differential equation from part (c) to show that
\[
T(z) = -3 \left( 4 \ln (1-z) + \frac{28}{9}z + \frac{29}{63} (1-z)^{-2} - \frac{2}{735} (1-z)^5 + \frac{1}{5} \right).
\]

(e) [15 Points] Use your solution from part (d) to show that for \( n \geq 0 \),
\[
t_n = \frac{12}{7} (n+1) H_n - \frac{159}{49} n - \frac{29}{147} - (-1)^n \frac{2}{735} \left( \frac{5}{n} \right) + \frac{1}{5} \left( \frac{0}{n} \right),
\]
where \( H_n = \sum_{k=1}^{n} \frac{1}{k} \) is the \( n \)th Harmonic number.\(^1\)

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. [60 Points] A Linear Sieve

In this task we will analyze the running time of a Linear Sieve which is a variant of the original Sieve of Eratosthenes modified to mark each composite exactly once. In contrast, the number of times the original sieve marks a composite \( C \) is equal to the number of prime factors of \( C \), and hence for finding all primes in \([2, N]\) it marks all composites in that range around \( N \log \log N \) times in total. The linear sieve we will consider in this task is known as the Sieve of Gries and Misra or the GM Linear Sieve.

Figure 2 shows an implementation of the GM linear sieve which uses a supporting data structure \( D \) composed of two priority queues and a stack. Indeed, one can show that when external-memory priority queues are used this implementation becomes more I/O-efficient than the standard implementation that does not use priority queues. Of course, in this task we are not concerned about I/O-efficiency. So, we will analyze the internal-memory running time of the implementation shown in Figure 2 when internal-memory priority queues (e.g., binary heaps, binomial heaps) are used.

The GM linear sieve uses the following property of composite numbers to reduce the number of times it marks them: each composite number \( C \) can be represented uniquely as \( C = p^r q \) where \( p \) is the smallest prime factor of \( C \), \( r \) is a positive integer, and either \( q = p \) or \( q > p \) is not divisible by

\(^1\)Compare this with \( t_n = 2(n+1) H_n - 4n \) which we obtained when we analyzed standard quicksort in Lecture 7.
Figure 2: An implementation of GM linear sieve using two priority queues and a stack.
p. Hence, one can generate all composites in a lexicographical order using a triply nested loop with
p in the outermost loop, q in the middle and r in the innermost loop, and this will generate/mark
every composite exactly once.

The support data structure \( D \) has three components: two priority queues \( D.Q_1 \) and \( D.Q_2 \) and
one stack \( D.S \). The priority queues support three operations: \textsc{Insert}, \textsc{Find-Min} and \textsc{Extract-Min}. The stack supports \textsc{Push} and \textsc{Pop}. The data structure \( D \) itself supports the following four
operations (see Figure 2 for details): \( D.\text{Insert} \), \( D.\text{InvSucc} \), \( D.\text{Save} \) and \( D.\text{Restore} \). It also
has an initialization function \( D.\text{Init} \). When called with parameter \( N \), the \textsc{Linear-Sieve} function
shown in Figure 2 uses this data structure to find all prime numbers in \([2, N]\).

Now answer the following questions.

(a) [10 Points] Assuming that \( D.Q_1 \) and \( D.Q_2 \) are standard binary heaps that support \textsc{Insert},
\textsc{Find-Min} and \textsc{Extract-Min} operations in \( O(\log n) \), \( O(1) \) and \( O(\log n) \) worst-case time,
respectively, where \( n \) is the number of items in the heap, find the worst case running times
of \( D.\text{Insert} \), \( D.\text{InvSucc} \), \( D.\text{Save} \) and \( D.\text{Restore} \) in terms of \( N \).

(b) [5 Points] Based on your results from part (a) give an upper bound on the worst-case running time of \textsc{Linear-Sieve}(\( N \)).

(c) [30 Points] Under the assumption that \( D.Q_1 \) and \( D.Q_2 \) are standard binary heaps as in
part (a), show that the amortized times complexities of \( D.\text{Insert} \), \( D.\text{InvSucc} \), \( D.\text{Save} \) and
\( D.\text{Restore} \) are \( \Theta(\log N) \), \( \Theta(1) \), \( \Theta(\log N) \) and \( \Theta(1) \), respectively.

(d) [5 Points] Based on your results from part (c) give an upper bound on the worst-case running time of \textsc{Linear-Sieve}(\( N \)).

(e) [10 Points] Suppose \( D.Q_1 \) and \( D.Q_2 \) are binomial heaps that support \textsc{Insert}, \textsc{Find-Min}
and \textsc{Extract-Min} operations in \( O(1) \), \( O(1) \) and \( O(\log n) \) amortized time, respectively,
where \( n \) is the number of items in the heap. Then what amortized bounds do you get for
\( D.\text{Insert} \), \( D.\text{InvSucc} \), \( D.\text{Save} \) and \( D.\text{Restore} \)? Based on those bounds give an upper
bound on the worst-case running time of \textsc{Linear-Sieve}(\( N \)).

Task 3. [40 Points] A Binomial Heap Variant Supporting Decrease-Key Operations

We modify the lazy binomial heap implementation (with doubly linked list representation) to
support \textsc{Decrease-Key} operations as follows.

Let’s denote the modified heap by \( \mathcal{H} \). Each node \( x \) of \( \mathcal{H} \) will now have a flag called \textit{dirty}. We will
say that node \( x \) is \textit{clean} provided \( x.\text{dirty} = \text{false} \), otherwise it’s \textit{dirty}. Initially, \( x.\text{dirty} \) is set to
\text{false}. Only a \textsc{Decrease-Key} operation performed on \( x \) can set \( x.\text{dirty} \) to \text{true}.

An \texttt{Insert}(\( \mathcal{H}, x \)) operation sets \( x.\text{dirty} = \text{false} \), creates a \( B_0 \) containing \( x \), and adds the new \( B_0 \)
to the doubly linked list containing all binomial trees of \( \mathcal{H} \).

A \texttt{Decrease-Key}(\( \mathcal{H}, x, k \)) operation is performed provided \( x.\text{dirty} = \text{false} \) and \( k < x.\text{key} \). It
sets \( x.\text{dirty} = \text{true} \), creates a new node \( y \) and sets \( y.\text{key} = k \). Then it performs \texttt{Insert}(\( \mathcal{H}, y \)).
An Extract-Min(\( \mathcal{H} \)) operation first performs a cleanup of \( \mathcal{H} \). The way the cleanup phase works depends on the percentage of dirty nodes in \( \mathcal{H} \). If the data structure contains more dirty nodes than clean nodes then the cleanup phase involves removing all dirty nodes from \( \mathcal{H} \) and inserting each clean node as a separate \( B_0 \) into the linked list. Otherwise, the cleanup phase proceeds as follows. It scans the doubly linked list in one direction and when it encounters some \( B_k \) with a dirty root it removes that root from \( \mathcal{H} \) and inserts its \( k \) children into the doubly linked list right in front of the current scan location (meaning that the scan will encounter these \( k \) trees before encountering any other tree currently in the linked list). The scan stops when the linked list no longer has a tree with a dirty root. Note that the trees can still have dirty (internal) nodes, but there will be no dirty roots.

After the cleanup phase an Extract-Min operation proceeds in exactly the way we saw in the class: convert the doubly linked list representation to the array representation, perform Extract-Min on the array representation, and finally convert the array representation back to the doubly linked list representation.

Now answer the following questions.

(a) [30 Points] Suppose we want to show that the amortized costs of Insert and Decrease-Key operations are \( O(1) \) and \( O(f(n)) \), respectively, where \( n \) is the number of clean nodes in \( \mathcal{H} \) and \( f(n) \) is any non-decreasing positive function of \( n \). Then what is the best amortized (upper) bound you can get for the cost of an Extract-Min operation?

(b) [10 Points] How will you modify the implementation above to also support Find-Min operations in amortized \( O(1) \) time?