CSE548, AMS542: Analysis of Algorithms, Fall 2017 **Date:** Oct 26

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.

```
MEDIAN-OF-3-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
2.     if A[2] < A[1] then swap A[1] and A[2]
3. elif n > 2 then
4.     x ← median of A[1], A[2] and A[3]
5.     rearrange the numbers of A[1 : n] such that
           (i) A[k] = x for some k ∈ [1, n],
           (ii) A[i] < x for each i ∈ [1, k − 1], and
           (iii) A[i] > x for each i ∈ [k + 1, n],
6.     MEDIAN-OF-3-QUICKSORT( A[1 : k − 1], k − 1 )
7.     MEDIAN-OF-3-QUICKSORT( A[k + 1 : n], n − k )
8. return
```

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

$$t_n = \begin{cases} 0 & \text{if } n < 2, \\ 1 & \text{if } n = 2, \\ n - \frac{1}{3} + \frac{6}{n(n-1)(n-2)} \sum_{k=1}^{n} (k-1)(n-k)(t_{k-1} + t_{n-k}) & \text{otherwise.} \end{cases}$$

(c) [**20 Points**] Let $T(z)$ be a generating function for t_n:

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

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

$$T(z) = -\frac{3}{7}\left(4 \ln{(1-z)} + \frac{28}{9} z + \frac{29}{63}\right)(1-z)^{-2} - \frac{2}{735}(1-z)^5 + \frac{1}{5}.$$

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

$$t_n = \frac{12}{7}(n+1)H_n - \frac{159}{49}n - \frac{29}{147} - (-1)^n \frac{2}{735}\binom{5}{n} + \frac{1}{5}\binom{0}{n},$$

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 $t_n = \Theta\left(n \log n\right)$.

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 $\mathcal{D}$ 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.

```
LINEAR-SIEVE( N )                                        {find all prime numbers in [2, N].}
create support data structure 𝒟
𝒟.INIT( N ),  p ← 1                                      {initialize support data structure 𝒟}
while p ≤ √N do                                          {output all primes ∈ [2, √N]}
    p' ← 𝒟.INVSUCC( p )        {assuming that all composites with value ≤ N and divisible by primes ∈ [2, p]
                 are already in 𝒟, find the smallest integer p' > p that does not appear as a composite in 𝒟}
    print p'                                             {then this p' must be a prime}
    p ← p',  q ← p'
    while pq ≤ N do
        for r ← 1 to ⌊log_p (N/q)⌋ do          {insert each composite of the form p^r q with value ≤ N into 𝒟,
            𝒟.INSERT( p^r q )                       where either q = p or q > p but is not divisible by p}
        q ← 𝒟.INVSUCC( q )                               {find the next q that is not divisble by p}
        𝒟.SAVE( q )                     {save q as we do not yet know if it's a prime or a composite}
    𝒟.RESTORE( )                                         {restore all saved q's}
while p ≤ N do                                           {output all primes ∈ (√N, N]}
    p ← 𝒟.INVSUCC( p )
    if p ≤ N then print p                                {p must be a prime}
```

```
𝒟.INIT( N )                      {initialize support data structure 𝒟 for computing primes in [2, N]}
𝒟.Q₁ ← {2, ..., N}               {𝒟.Q₁ is a priority queue containing numbers not yet known to be composites}
𝒟.Q₂ ← ∅                         {𝒟.Q₂ is a priority queue containing composites we have
                                   discovered that are yet to be deleted from 𝒟.Q₁}
𝒟.S ← ∅                          {𝒟.S is a stack}
```

```
𝒟.INVSUCC( x )       {return the smallest number larger than x which is not yet known to be a composite}
while FIND-MIN( 𝒟.Q₁ ) ≤ x do                            {get rid of all numbers ≤ x from 𝒟.Q₁}
    EXTRACT-MIN( 𝒟.Q₁ )
while FIND-MIN( 𝒟.Q₂ ) ≤ FIND-MIN( 𝒟.Q₁ ) do     {keep removing numbers from 𝒟.Q₁ (in increasing
    if FIND-MIN( 𝒟.Q₂ ) = FIND-MIN( 𝒟.Q₁ ) then     order of value) which also belong to 𝒟.Q₂ (i.e., known
        EXTRACT-MIN( 𝒟.Q₁ )                          to be composites) until finding one that does not belong to 𝒟.Q₂}
    EXTRACT-MIN( 𝒟.Q₂ )              {remove composites from 𝒟.Q₂ which have already been removed from 𝒟.Q₁}
y ← EXTRACT-MIN( 𝒟.Q₁ )                   {y is the smallest number in 𝒟.Q₁ which does not belong to 𝒟.Q₂
                                           and thus not known to be a composite}
return y
```

```
𝒟.INSERT( x )                                {x is a composite to be deleted from 𝒟.Q₁}
INSERT( 𝒟.Q₂, x )                            {store x in 𝒟.Q₂ for deletion from 𝒟.Q₁ at a convenient time later}
```

```
𝒟.SAVE( x )                                  {save x as we do not yet know if it's a prime or not}
PUSH( 𝒟.S, x )                               {store x in stack 𝒟.S}
```

```
𝒟.RESTORE( )                                 {empty the contents of 𝒟.S into 𝒟.Q₁}
while 𝒟.S ≠ ∅ do
    x ← POP( 𝒟.S )                           {return the contents of stack 𝒟.S
    INSERT( 𝒟.Q₁, x )                         to the priority queue 𝒟.Q₁}
```

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 $\mathcal{D}$ has three components: two priority queues $\mathcal{D}.Q_1$ and $\mathcal{D}.Q_2$ and one stack $\mathcal{D}.S$. The priority queues support three operations: INSERT, FIND-MIN and EXTRACT-MIN. The stack supports PUSH and POP. The data structure $\mathcal{D}$ itself supports the following four operations (see Figure 2 for details): $\mathcal{D}$.INSERT, $\mathcal{D}$.INVSUCC, $\mathcal{D}$.SAVE and $\mathcal{D}$.RESTORE. It also has an initialization function $\mathcal{D}$.INIT. When called with parameter N, the 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 $\mathcal{D}.Q_1$ and $\mathcal{D}.Q_2$ are standard binary heaps that support INSERT, FIND-MIN and EXTRACT-MIN operations in $\mathcal{O}(\log n)$, $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ worst-case time, respectively, where n is the number of items in the heap, find the worst case running times of $\mathcal{D}$.INSERT, $\mathcal{D}$.INVSUCC, $\mathcal{D}$.SAVE and $\mathcal{D}$.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 LINEAR-SIEVE(N).

(*c*) [**30 Points**] Under the assumption that $\mathcal{D}.Q_1$ and $\mathcal{D}.Q_2$ are standard binary heaps as in part (*a*), show that the amortized times complexities of $\mathcal{D}$.INSERT, $\mathcal{D}$.INVSUCC, $\mathcal{D}$.SAVE and $\mathcal{D}$.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 LINEAR-SIEVE(N).

(*e*) [**10 Points**] Suppose $\mathcal{D}.Q_1$ and $\mathcal{D}.Q_2$ are binomial heaps that support INSERT, FIND-MIN and EXTRACT-MIN operations in $\mathcal{O}(1)$, $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ amortized time, respectively, where n is the number of items in the heap. Then what amortized bounds do you get for $\mathcal{D}$.INSERT, $\mathcal{D}$.INVSUCC, $\mathcal{D}$.SAVE and $\mathcal{D}$.RESTORE? Based on those bounds give an upper bound on the worst-case running time of 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 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 *dirty*. We will say that node x is *clean* provided $x.dirty = false$, otherwise it's *dirty*. Initially, $x.dirty$ is set to *false*. Only a DECREASE-KEY operation performed on x can set $x.dirty$ to *true*.

An INSERT($\mathcal{H}$, x) operation sets $x.dirty = false$, 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 DECREASE-KEY($\mathcal{H}$, x, k) operation is performed provided $x.dirty = false$ and $k < x.key$. It sets $x.dirty = true$, creates a new node y and sets $y.key = k$. Then it performs 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 $\mathcal{O}(1)$ and $\mathcal{O}(f(n))$, 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 $\mathcal{O}(1)$ time?