CSE548, AMS542: Analysis of Algorithms, Fall 2017 Date: Nov 10

Homework 2

Group Number: 52

Name	SBU ID	% Contribution
Rahul Bhansali	111401451	33.33
Kiranmayi Kasarapu	111447596	33.33
Neha Indraniya	111499447	33.33

Collaborating Groups

Group Number	Specifics Number (e.g., specific group member? specific task/subtask?)
61	1(d), 3(a), Question 2
26	1(c)
74	Question 2
19	Question 2, Question 1

External Resources Used

	Specifics (e.g., cite papers, webpages, etc. and mention why each was used)

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

(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 − 1 3 element comparisons on average, where the average is taken over all n! possible permutations of the input numbers.

4. x ← median of A[1], A[2] and A[3]

As we can see, Step 4 involves finding the median of 3 numbers.
We can find the median using this:

```
def Median(a, b, c):
    if b>a:
        if b>c:
            if(a>c):
                return a
            else:
                return c
         else:
             return b
     else:
         if b<c:
             if a>c:
                 return c
             else:
                 return a
         else:
             return b
```

Consider the first three numbers.
There can be 3! combinations of those numbers
In these 6 possible combinations, using above implementation of finding the median:
-> 4 combinations will need 3 comparisons. (when the middle element is not the median)
-> 2 combinations will need 2 comparisons. (when the middle element is the median)

Therefore, total number of comparisons in the 6 possible combinations:

(4x3 + 2x2) = 16.

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],

Step 5 will be done by simply comparing each of the remaining (n-3) numbers to the pivot (found in Step 4) and then placing it in either to the left or right of the pivot. (A standard quicksort partition function).
Thus, each case will require (n-3) comparisons.
There are 6 such cases, so total comparisons in the 6 cases = 6x(n-3).

Therefore, average number of comparisons of the 6 cases
= (16+ 6x(n-3))/6 = 8/3 + n-3 =(n - ⅓) comparisons.

We can extend above result to find the average over all n! permutations, which is:

((n-⅓)*nC3*3!*(n-3)!) /n! = n-⅓ comparisons.

Hence, proved.

(b) [10 Points] Let tn be the average number of element comparisons performed by the algorithm given in Figure 1 to sort A[1 : n], where n ≥ 0 and the average is taken over all n! possible permutations of the numbers in A. Show that 1 tn = 0 if n < 2, 1 if n = 2, n − 1 3 + 6 n(n−1)(n−2) Pn k=1 (k − 1)(n − k)(tk−1 + tn−k) otherwise.

For n<2: t(n) = 0 , as no comparisons are needed.
For n=2: t(n) =1 , as a single comparison can be used to sort the element.

For n>2:

For step 4 and step 5, we need n-⅓ comparisons, on average, as proved above.

Now suppose, **k** is the median i.e. it ends up being selected as the pivot.
How many possible permutations are there where k is selected as the median i.e. it ends up being selected as the pivot?
To count this we can do the following:
-> We need to make k as the median, so in the first three numbers, we need to select one number smaller than k and one number larger than k and the third one k.

If k is the median, then there are (k-1) smaller numbers than k and (n-k) larger numbers than k.
So,
The number of ways to select a number smaller than k = (k-1)C1.
The number of ways to select a number larger than k = (n-k)C1.
The number of ways to select k = 1C1.

Therefore, the total number of permutations where k occurs as the median (becomes pivot) =
((k-1)C1)x((n-k)C1)x(1C1)x3!x(n-3)!

Now, step 6 and step 7 are two subproblems of size (k-1) and (n-k).

Therefore, for a given k:
-> Comparisons needed to solve one permutation where k is the median = (t_{k-1} +t_{n-k})
-> Number of permutations where k is the median=
((k-1)C1)x((n-k)C1)x(1C1)x3!x(n-3)!

-> So total number of comparisons for all permutations where k is the median =
((k-1)C1)x((n-k)C1)x(1C1)x3!x(n-3)!x(t_{k-1} +t_{n-k})

-> Average number of comparisons for all permutations where k is the median =
((k-1)C1)x((n-k)C1)x(1C1)x3!x(n-3)!x(t_{k-1} +t_{n-k}) / n!

But k itself can vary from 1 to n:

Therefore, average number of comparisons needed over all possible n! permutations =

=> n-⅓ + Σ (k=1 to n) ((k-1)C1)x((n-k)C1)x(1C1)x3!x(n-3)!x(t_{k-1} +t_{n-k}) / n!
=> n-⅓ + Σ (k=1 to n) (k-1)x(n-k)x3!x(n-3)!x(t_{k-1} +t_{n-k})/n!
=> n-⅓ + 3!x(n-3)!/n! Σ (k=1 to n)(k-1)x(n-k)(t_{k-1} +t_{n-k})
=> n-⅓ + (6/(n)(n-1)(n-2)) Σ (k=1 to n) (k-1)(n-k)(t_{k-1} +t_{n-k})

Hence, proved.

(c) [20 Points] Let T(z) be a generating function for tn: T(z) = t0 + t1z + t2z 2 + . . . + tnz n + ... Show that T(z) = 12 (1−z) 2 T 0 (z) − 8 (1−z) 4 + 24 (1−z) 5 .

$$Tn = n - \tfrac{1}{3} + \tfrac{6}{n(n-1)(n-2)} \sum_{k=1}^{n} (k-1)(n-k)(t_k + t_{n-k}) + [n=2]$$

$$Let\, g(x) = \sum_{n=0}^{\infty} \tfrac{6}{n(n-1)(n-2)} \sum_{k=1}^{n} (k-1)(n-k)(t_k + t_{n-k}) z^n$$

$$T(z) = \sum_{n=0}^{\infty} nz^n - \tfrac{1}{3} \sum_{n=0}^{\infty} z^n + g(x) + z^2$$

$$T(z) = \tfrac{z}{(1-z)^2} - \tfrac{1}{3(1-z)} + z^2 + g(x\,)$$

$$g'''(x) = \sum_{n=3}^{\infty} 6 \sum_{k=1}^{n} (k-1)(n-k)(t_k + t_{n-k}) z^{n-3}$$

$$= \sum_{n=3}^{\infty} 6 * 2 \sum_{k=1}^{n-2} (k)(n-k-1)(t_k) z^{n-3}$$

$$= 12 \sum_{n=3}^{\infty} \sum_{k=1}^{n-2} (k)(n-k-1)(t_k) z^{n-3}$$

n/k	1	2	3
3	t_1		
4	$2t_1z$	$2t_2z$	
5	$3t_1z^2$	$4t_2z^2$	$3t_3z^2$

$$g'''(x) = 12t_1(1 + 2z + 3z^2 ...) + 2t_2z(1 + 2z + 3z^2 ...) + ...$$

$$= 12(1 + 2z + 3z^2 ...)(\, t_1 + 2t_2z + 3t_3z^2 + ...)$$

$$12 \sum_{n=1}^{\infty} (n+1) z^n * (\, t_1 + 2t_2z + 3t_3z^2 + ...)$$

$$= 12 \tfrac{1}{(1-z)^2} * (t_1 + 2t_2z + 3t_3z^2 + ...)$$

$$= 12 \tfrac{1}{(1-z)^2} \sum_{n=0}^{\infty} (n+1) t_{n+1} z^n$$

$$= 12 \tfrac{1}{(1-z)^2} \sum_{n=1}^{\infty} (n) t_n z^{n-1}$$

$$= \tfrac{12}{(1-z)^2} T'(z)$$

$$T'''(z) = \frac{d^3}{dz^3}\frac{z}{(1-z)^2} - \frac{d^3}{dz^3}\frac{1}{3(1-z)} + \frac{d^3}{dz^3}z^2 + g'''(x)$$

$$\frac{d}{dz}\frac{z}{(1-z)^2} = \frac{(1-z)^2 + 2z(1-z)}{(1-z)^4} = \frac{(1-z) + 2z}{(1-z)^3} = \frac{(1+z)}{(1-z)^3}$$

$$\frac{d^2}{dz^2}\frac{z}{(1-z)^2} = \frac{(1-z)^3 + (1+z)3(1-z)^2}{(1-z)^6} = \frac{(1-z) +(1+z)^3}{(1-z)^4} = \frac{1-z+3+3z}{(1-z)^4} = \frac{2(2+z)}{(1-z)^4}$$

$$\frac{d^3}{dz^3}\frac{z}{(1-z)^2} = \frac{(1-z)^4 2 + 2(2+z)4(1-z)^3}{(1-z)^8} = \frac{2(1-z) + 8(2+z)}{(1-z)^5} = \frac{18+6z}{(1-z)^5}$$

$$\frac{d^3}{dz^3}\frac{1}{3(1-z)} = \frac{6}{3(1-z)^4} = \frac{2}{(1-z)^4}$$

$$T'''(z) = \frac{18+6z}{(1-z)^5} - \frac{2}{(1-z)^4} + \frac{12}{(1-z)^2}T'(z)$$

$$T'''(z) = \frac{12}{(1-z)^2}T'(z) + \frac{18}{(1-z)^5} + \frac{6z}{(1-z)^5} - \frac{2}{(1-z)^4}$$

$$= \frac{12}{(1-z)^2}T'(z) + \frac{24-6}{(1-z)^5} + \frac{6z}{(1-z)^5} - \frac{2}{(1-z)^4}$$

$$= \frac{12}{(1-z)^2}T'(z) + \frac{24}{(1-z)^5} + \frac{6z}{(1-z)^5} - \frac{6}{(1-z)^5} - \frac{2}{(1-z)^4}$$

$$\Rightarrow T'''(z) = \frac{12}{(1-z)^2}T'(z) + \frac{24}{(1-z)^5} - \frac{8}{(1-z)^4}$$

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

$$T'''(z) = \frac{12}{(1-z)^2}T'(z) + \frac{24}{(1-z)^5} - \frac{8}{(1-z)^4}$$

$$(z-1)^2T'''(z) - 12T'(z) = \frac{-24}{(z-1)^3} - \frac{8}{(z-1)^2}$$

$$\Rightarrow \int (z-1)^2T'''(z) - \int 12T'(z) = \int \frac{-24}{(z-1)^3} - \int \frac{8}{(z-1)^2}$$

$$\Rightarrow (z-1)^2T''(z) - 2\int (z-1)\,T''(z) - 12T(z) = \frac{12}{(z-1)^2} + \frac{8}{(z-1)} + C$$

$$\Rightarrow (z-1)^2T''(z) - 2[(z-1)\,T'(z) - \int T'(z)] - 12T(z) = \frac{12}{(z-1)^2} + \frac{8}{(z-1)} + C$$

$$\Rightarrow (z-1)^2T''(z) - 2[(z-1)\,T'(z) - T(z)] - 12T(z) = \frac{12}{(z-1)^2} + \frac{8}{(z-1)} + C$$

$$\Rightarrow (z-1)^2T''(z) - 2(z-1)\,T'(z) - 10T(z) = \frac{12}{(z-1)^2} + \frac{8}{(z-1)} + C$$

$$We\ know\ T(0) = 0,\ T'(0) = 0,\ T''(0) = 2$$

$$Substituting\ z = 0$$

$$T''(0) - 2(-1)T'(0) - 10T(0) = 12 - 8 + C$$

=> C = 2 + 8 - 12 = -2

So substituting back C in above equation

$$(z-1)^2T''(z) - 2(z-1)\,T'(z) - 10T(z) = \frac{12}{(z-1)^2} + \frac{8}{(z-1)} + -2$$

$$(z-1)^3T''(z) - 2(z-1)^2T'(z) - 10(z-1)\,T(z) = \frac{12}{(z-1)} + 8 + -2(z-1)$$

$$[(z-1)^3T''(z) + 3(z-1)^2T'(z)] - [5(z-1)^2T'(z) - 10(z-1)\,T(z)] = \frac{12}{(z-1)} + 8 + -2(z-1)$$

$$\frac{d}{dz}((z-1)^3T'(z)) - 5\,\frac{d}{dz}((z-1)^2T\,(z)) = \frac{12}{(z-1)} + 8 + -2(z-1)$$

Integrating again on both sides

$$\int \frac{d}{dz}((z-1)^3T'(z)) - 5\int \frac{d}{dz}((z-1)^2T\,(z)) = \int \frac{12}{(z-1)} + \int 8 + -2\int(z-1)$$

$$(z-1)^3T'(z) + 5(z-1)^2T\,(z) = 12ln(z-1) + 8z - (z-1)^2 + C1$$

Substitute z = 0 to find C1

We get C1 = 1

$$(z-1)^3T'(z) + 5(z-1)^2T\,(z) = 12ln(z-1) + 8z - (z-1)^2 + 1$$

Dividing by $(z-1)^8$

$$\frac{T'(z)}{(z-1)^5} + \frac{5T(z)}{(z-1)^6} = \frac{12ln(z-1)}{(z-1)^8} + \frac{8z}{(z-1)^8} - \frac{1}{(z-1)^6} + \frac{1}{(z-1)^8}$$

$$\frac{d}{dz}\left(\frac{T(z)}{(z-1)^5}\right) = \frac{12ln(z-1)}{(z-1)^8} + \frac{8z}{(z-1)^8} - \frac{1}{(z-1)^6} + \frac{1}{(z-1)^8}$$

$$\frac{d}{dz}\left(\frac{T(z)}{(z-1)^5}\right) = \frac{12ln(z-1)}{(z-1)^8} + \frac{8}{(z-1)^7} + \frac{9}{(z-1)^8} - \frac{1}{(z-1)^6}$$

Integrating again on both sides

$$\int \frac{d}{dz}\left(\frac{T(z)}{(z-1)^5}\right) = \int \frac{12ln(z-1)}{(z-1)^8} + \int \frac{8}{(z-1)^7} + \int \frac{9}{(z-1)^8} - \int \frac{1}{(z-1)^6}$$

$$\frac{T(z)}{(z-1)^5} = \int \frac{12ln(z-1)}{(z-1)^8} - \frac{4}{3(z-1)^6} - \frac{9}{7(z-1)^7} + \frac{1}{5(z-1)^5} + C2$$

$$\int \frac{12ln(z-1)}{(z-1)^8} = 12\left[\frac{-ln(z-1)}{7(z-1)^7} - \int \frac{1}{7(z-1)^8}\right]$$

$$= 12\left[\frac{-ln(z-1)}{7(z-1)^7} + \frac{1}{49(z-1)^7}\right]$$

$$=> \frac{T(z)}{(z-1)^5} = -\frac{4}{3(z-1)^6} - \frac{9}{7(z-1)^7} + \frac{1}{5(z-1)^5} - \frac{12ln(z-1)}{7(z-1)^7} + \frac{12}{49(z-1)^7} + C2$$

Substituting z = 0, we get C2 = $\frac{2}{735}$

$$\frac{T(z)}{(z-1)^5} = -\frac{4}{3(z-1)^6} - \frac{9}{7(z-1)^7} + \frac{1}{5(z-1)^5} - \frac{12ln(z-1)}{7(z-1)^7} + \frac{12}{49(z-1)^7} + C2$$

$$T(z) \;=\; \tfrac{-4}{3}(z-1) \;-\; \tfrac{9}{7}(z-1)^{-2} \;+\tfrac{1}{5} \;-\tfrac{12}{7}ln(z-1)(z-1)^{-2} \;+\tfrac{12}{49}(z-1)^{-2} \;+\tfrac{2}{735}(z-1)^{5}$$

On rearranging, we get the following

$$T(z) \;=\; \tfrac{-3}{7}[4ln(1-z) \;+\; \tfrac{28}{9}z \;+\; \tfrac{29}{63}]\;(z-1)^{-2} - \tfrac{2}{735}(z-1)^{5} + \tfrac{1}{5}$$

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

We have T(z) from above solution.

-ln(1-z) expansion is as follows

$$-ln(1-z) \;= z \;+\; z^2/2 \;+z^3/3 \;+z^4/4 \;+.... = \sum_{k=1}^{\infty} z^k/k$$

$$(1-z)^{-2} = 1 + 2z \;+3z^2 \;+4z^3 + ... \;=\; \sum_{j=0}^{\infty}(j+1)z^j$$

Substitution these expansions in T(z) we get

$$T(z) \;=\; \tfrac{3}{7}[4\sum_{k=1}^{\infty} z^k/k \;-\; \tfrac{28}{9}z \;-\; \tfrac{29}{63}]\;\sum_{j=0}^{\infty}(j+1)z^j \;-\; \tfrac{2}{735}(z-1)^5 + \tfrac{1}{5}$$

We have to find coefficient of Z^n from the above equation

Coefficient of x^k in $(x-1)^n$ = (-1)k*($^n c_k$), we will apply this below

$$t_n = \;\tfrac{3}{7}[4\sum_{k=1}^{n} 1/k*(n-k+1) \;-\; \tfrac{28}{9}(n) \;-\; \tfrac{29}{63}(n+1)] - \tfrac{2}{735}(-1)^n(5Cn) \;+\tfrac{1}{5}(0Cn)$$

Solving the first summation

$$\sum_{k=1}^{n} 1/k*(n-k+1) = \sum_{k=1}^{n}\tfrac{(n+1-k)}{k} =$$

$$\sum_{k=1}^{n}\tfrac{(n+1)}{k} - \sum_{k=1}^{n} 1 = (n+1)\sum_{k=1}^{n}\tfrac{1}{k} - n = (n+1)H_n - n$$

Substitute this in the previous equation and we get the following

$$t_n = \;\tfrac{3}{7}[4(n+1)H_n - 4n \;-\; \tfrac{28}{9}(n) \;-\; \tfrac{29}{63}(n+1)] - \tfrac{2}{735}(-1)^n(5Cn) \;+\tfrac{1}{5}(0Cn)$$

$$= \tfrac{12}{7}(n+1)H_n - \tfrac{12}{7}n \;-\; \tfrac{3}{7}*\tfrac{28}{9}n \;-\; \tfrac{3}{7}*\tfrac{29}{63}n - \tfrac{3}{7}*\tfrac{29}{63} - \tfrac{2}{735}(-1)^n(5Cn) \;+\tfrac{1}{5}(0Cn)$$

$$= \tfrac{12}{7}(n+1)H_n - \tfrac{n}{7}(12 \;-\; \tfrac{28}{3} \;-\; \tfrac{29}{21}) - \tfrac{29}{147} - \tfrac{2}{735}(-1)^n(5Cn) \;+\tfrac{1}{5}(0Cn)$$

$$= \tfrac{12}{7}(n+1)H_n - \;\tfrac{159}{49}n - \tfrac{29}{147} - \tfrac{2}{735}(-1)^n(5Cn) \;+\tfrac{1}{5}(0Cn)$$

For n = 0:
H0 is not defined for n = 0, and we know T(0) = 0
So t0 = 0

For n = 1:

$$t1 \;=\; \tfrac{12}{7}(1+1)1 - \;\tfrac{159}{49} - \tfrac{29}{147} + \tfrac{2}{735} * 5 + \tfrac{1}{5} * 0 \;\;=\;\; 0$$

For n = 2:

$$t2 \;=\; \tfrac{12}{7}(2+1)(\tfrac{1}{1} + \tfrac{1}{2}) - \;\tfrac{159}{49} * 2 - \tfrac{29}{147} - \tfrac{2}{735} * 10 + \tfrac{1}{5} * 0 \;\;=\;\; 1$$

For n = 3:

$$t3 \;=\; \tfrac{12}{7}(3+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3}) - \;\tfrac{159}{49} * 3 - \tfrac{29}{147} + \tfrac{2}{735} * 10 + \tfrac{1}{5} * 0 \;\;=\;\; 8/3$$

For n = 4

$$t4 \;=\; \tfrac{12}{7}(4+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4}) - \;\tfrac{159}{49} * 4 - \tfrac{29}{147} - \tfrac{2}{735} * 5 + \tfrac{1}{5} * 0 \;\;=\;\; 14/3$$

For n = 5

$$t5 \;=\; \tfrac{12}{7}(5+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5}) - \;\tfrac{159}{49} * 5 - \tfrac{29}{147} + \tfrac{2}{735} * 1 + \tfrac{1}{5} * 0 \;\;= 106/15$$

For n = 6

$$t6 \;=\; \tfrac{12}{7}(6+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \tfrac{1}{6}) - \;\tfrac{159}{49} * 6 - \tfrac{29}{147} - \tfrac{2}{735} * 0 + \tfrac{1}{5} * 0 \;\;= 146/15$$

For n=7

$$t7 \;=\; \tfrac{12}{7}(7+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \tfrac{1}{6} + \tfrac{1}{7}) - \;\tfrac{159}{49} * 7 - \tfrac{29}{147} \;\;= 1328/105$$

For n=8

$$t8 \;=\; \tfrac{12}{7}(8+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \tfrac{1}{6} + \tfrac{1}{7} + \tfrac{1}{8}) - \;\tfrac{159}{49} * 8 - \tfrac{29}{147} \;\;= 3313/210$$

For n=9

$$t9 \;=\; \tfrac{12}{7}(9+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \tfrac{1}{6} + \tfrac{1}{7} + \tfrac{1}{9}) - \;\tfrac{159}{49} * 9 - \tfrac{29}{147} \;\;= 401/21$$

Fo n=10

$$t10 \;=\; \tfrac{12}{7}(10+1)(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \tfrac{1}{6} + \tfrac{1}{7} + \tfrac{1}{9} + \tfrac{1}{10}) - \;\tfrac{159}{49} * 10 - \tfrac{29}{147} \;\;= 28.085$$

(f)[5 Points] Use your solution from part (e) to show that tn = Θ (n log n).

From above solution we have

$$tn \;=\; \tfrac{12}{7}(n+1)H_n - \;\tfrac{159}{49}n - \tfrac{29}{147} - \tfrac{2}{735}(-1)^n(5Cn) \;+ \tfrac{1}{5}(0Cn)$$

In the class we have seen that Hn is $\Theta(logn)$. Replacing this the above equation we get

$$tn = \frac{12}{7}(n+1)\Theta(logn) - \frac{159}{49}n - \frac{29}{147} - \frac{2}{735}(-1)^n(5Cn) + \frac{1}{5}(0Cn)$$

=

$$\Theta(nlogn)$$

As other terms are lower order terms and can be ignored.

Question 2: [60 Points] A Linear Sieve

(a) [10 Points] Assuming that D.Q1 and D.Q2 are standard binary heaps that support Insert, Find-Min and 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.Insert, D.InvSucc, D.Save and D.Restore in terms of N.

D.Insert(x):

As for the given priority queue Q2 the complexity of insert is O(log n). D.Insert has only insert operation into the priority queue, therefore, the complexity of D.Insert of linear-sieve will be O(log N)

D.InvSucc(x):

Analyzing 1st while loop:

The complexity of line number 2 will be O(log n), as extract-min from priority queue takes logn time. Now we'll have to check how many times, the while loop is executed in worst case time. It is going over D.Q1, and we will only execute till elements which are less than n. The number of elements we extract from D.Q1 can never be more than logn as we remove all multiples of a particular prime in every run and adds back everything which we are not sure of. But definitely after 1st run, i.e when 2 is prime number, we remove all multiples of 2. So the number of elements reduce by half, in the first iteration itself, but we never extracted any elements greater than 2 from D.Q1 in the first iteration. Also, the distance between 2 prime numbers is logn. So we will only scan logn elements in worst case. Therefore the time complexity of first while loop is O(logn * logn) = O($(\log n)^2$)

Analyzing 2nd while loop:

Between FindMin(Q1) and FindMin(Q2), there will again be log(n) apart. So at max, the while loop runs logn times. We are extracting minimum inside while loop which takes logn time. Therefore here also we get O($(\log n)^2$) complexity.

So the total time complexity is O($(\log n)^2$) for D.InvSucc()

D.Save(x):

As S is a Stack, hence the runtime complexity of line no. 1 will be O(1). Hence, worst case time complexity of Save(x) will be:-
T (N) = O (1)

D.Restore( ):
As S is s stack, the time complexity of pop(D.S) of line no. 2 will be O(1), and its given that Q1 is a priority queue hence, complexity of insert of line no. 3 will be O(logn). Now we'll have to check, how many maximum elements can the stack have. The maximum elements it can have is number of primes or composites(which are not yet known to be composites) which are less than current prime. The number of primes which are <=N are always N/logN, as each prime is logn apart. These are max number of primes we may insert.
Given that we push every q into stack of the form $p^r q<=N$, the maximum number of q's we insert happens when p is minimum. The minimum value of p is 2. When we analyze this we get maximum of N/logN elements only can be inserted.

Therefore the complexity of D.Restore is $\frac{N}{logN} * logN = N$
T(N) = O(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).

Code

```
1.  create support data structure D
2.  2. D.Init( N ), p ← 1
3.  while p ≤ √N do
4.      p' ← D.InvSucc( p )
5.      print p'
6.      p ← p' , q ← p'
7.      while pq ≤ N do
8.          For r ← 1 to log_p (N/q) do
9.              D.Insert( p^r q )
10.         q ← D.InvSucc( q )
11.         D.save(q)
12.     D.Restore( )
13. while p ≤ N do
14.     p ← D.InvSucc( p )
15.     if p ≤ N then print p
```

Lets looks at the 3 while loops from lines 3-12
The outer while loop runs $\sqrt{N}$ / logN times as p is always prime and there will only $\sqrt{N}$ / log $\sqrt{N}$ prime numbers less than $\sqrt{N}$ and the outer loop will not run any more than this. There are 2 loops inside of this loop. If we analyze the 2nd loop from lines 7-11, It will execute maximum of N / $(\log N)^2$ times
Line 9, where insertion is done, it executes only n times, as we will never insert more than n number into the queue, so it is O(NlogN), as insert of 1 elements takes logN time.
We increase p from 2 to $\sqrt{N}$, q decreases accordingly. We get highest r when q is minimum, and r also decreases when p or q increase. This balances out and the overall loop runs only O(n) times. Line 12 D.Restore also only for a maximum of O(N) items, because the loop runs only for N / $(\log N)^2$ times, and the outer loop runs for N/logN. So overall time complexity of the algorithm comes out to be O(NlogN)

(c) [30 Points] Under the assumption that D.Q1 and D.Q2 are standard binary heaps as in part (a), show that the amortized times complexities of D.Insert, D.InvSucc, D.Save and D.Restore are Θ (log N), Θ (1), Θ (log N) and Θ (1), respectively.

D.Insert(x):
In the question, we are already given that Q1 and Q2 are binary heaps, so insertion into Q1 and Q2 takes logn time as taught in class. But we will overcharge the insertion operation and puts c*logn+1 extra coins at each node when we do the insertion. Still the amortized cost would not change as it will be c*logn+1 = O(logn) itself
Hence, amortized time complexity= O(log N).

D.Save(x):
It simply pushes the element into the stack which takes O(1) time. But we will overcharge here again and keep (logn) with it for each nodes that it pushed into the stack. So the amortized cost will be O(logn) time

D.Restore( ):
It pops each element from the stack and inserts it into the queue. Insertion into the queue takes O(logn) time. But in the save operation, for every element we pushed into the stack, we saved logn coins(or charges logn extra time). We make use of these coins for insert operation. When the stack is empty, we still have to do one last comparision which takes O(1). We can charge this extra constant time to D.Restore.

So D.Restore, uses the logn coins for each popped node, and do not charge anything for insert from it self. So amortized cost is O(1).

D.InvSucc(x):
We have seen previously that the each while loop in this function cannot run more than logn times.
We know extract-min takes O(logn) time for a binary heap. But during insert we have overcharged the insert operation and kept c*logn extra coins with it. Extract-min will make use of these 1*log(n) extra coins that we save at each node during insert. Now we will have to pay for the while loop also. We can use (c-1) *logn remaining coins for traversing the while loop.

(d) [5 Points] Based on your results from part (c) give an upper bound on the worst-case running time of Linear-Sieve(N).
From part (c) , we can see that D.Insert,D.Save,D.Restore, D.InvSucc have O(log N) amortised costs. And as we N elements, so the upper bound on worst case running time will be O(N log N)

(e) [10 Points] Suppose D.Q1 and D.Q2 are binomial heaps that support Insert, Find-Min and 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.Insert, D.InvSucc, D.Save and D.Restore? Based on those bounds give an upper bound on the worst-case running time of Linear-Sieve(N).

D.Insert: Now the priority queue is binomial heap, hence it will take O(1) insert time to insert. While inserting we will be allotting c coins to each element. Hence, amortised time complexity will be O(1).

D.InvSucc: In this we have two queues : Q1 and Q2. Q1 will have maximum of N elements and Q2 will have composite number, unvisited till k at any given time. So, any number is extracted from these queues exactly once**.** The complexity of find-min and extract-min is given as O(1) and O(log n). Since, for each element we already have one coin with each element. Hence amortised cost of each element will be O(log N). The two while loop could run maximum of log N times. Hence, the amortised time complexity of InvSucc will be $O((log\ N)^2)$

D.Save: This function uses push internally, hence each element will be using one coin each time Save is called. Let us allot two coins to each element. Hence the amortised cost will be O(1)

D.Restore: In this function we are doing two things 1.) pop from stack 2.) insert in queue. Each of this will has complexity of O(1). Since there are coins stored with each element in stack because of Save function. So, the amortised time complexity will be O(N)

Question 3:

[30 Points] Suppose we want to show that the amortized costs of Insert and DecreaseKey operations are O (1) and O (f(n)), respectively, where n is the number of clean nodes in 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?

DecreaseKey is being done in O(f(n)) amortized time.
So, we can keep O(f(n)) coins to the node being marked as dirty during DecreaseKey.
Now there are two cases for DecreaseKey:

1. No. of dirty nodes are more than the number of clean nodes.
2. No. of clean nodes are more than the number of dirty nodes.

Let's do an analysis for each case:

Whenever we do a DecreaseKey we add O(f(n)) coins to the dirty node.

1. No. of dirty nodes are more than the number of clean nodes:
We see that whenever we perform DecreaseKey the count of the number of clean nodes remains constant but the number of dirty nodes increases by 1.
Let the number of dirty nodes be d.
=>d>n/2 (where n is the total number of nodes).
Therefore, we will have d*O(f(n)) coins in total.
Let's assume that f(n)>=2.
So, in this case, in total, the dirty nodes will have enough coins to pay for the cost of linking each node to the doubly linked list. So, in this case, we can do the cleanup operation for free using the O(f(n)) coins in the dirty nodes.
The amortized cost for this case will just be of converting the linked list to array, doing an extract min and converting linked list back to array. -> O(log n)

2. No. of clean nodes are more than the number of dirty nodes:
In this case we remove a dirty node from the doubly linked list and then add its children to the doubly linked list.
Suppose the degree of such a node is k. Then we will link k children to the doubly linked list.
Now, at each dirty node, we already have O(f(n)) coins.
Let O(f(n)) = c*f(n) (where c>0)
So, we can pay for c*f(n) links. But there will be (k- c*f(n)) links which need to be paid for. We will charge this cost to ExtractMin.

Let’s try to get an upper bound to this cost.
So, for a node of degree k, we will charge cost at most = (no. of dirty nodes of degree k)*(k - c*f(n)), as this cost can’t be paid for using the O(f(n)) coins in the dirty nodes.

But what can be an upper bound on the no. of dirty nodes of degree k. The max value can occur when we mark all the nodes of the “original” binomial heap as dirty.

So, we just need to figure out at most how many nodes are there of degree d?

Initially, before any operations were performed, we had a “completely clean” binomial heap.
So, if we had total number of nodes n, then in this heap, we had at most one B_0, B_1, B_2…. upto B_k
where k = log n.
Now, before doing an ExtractMin we can do only Insert or DecreaseKey operations.
In each of these operations, we are simply adding a B0 to the doubly linked list. We do not change the **structure** of the “original” binomial heap.
So, if we can find out an upper bound on how many nodes of degree d exists in the “original” binomial heap. It will not change after performing Inserts and DecreaseKeys.
Simply the number of B_0s will increase via Inserts and DecreaseKeys.

Now suppose we want to find how many nodes are there of degree d in the original binomial heap ->
For B_0 to B_{d-1} there will be 0 such nodes.
For B_d => 1 node
For B_{d+1} =-> 1 node
For B_{d+2} => 2 nodes
For B_{d+3} => 4 nodes
For B_{d+4} => 8 nodes

.
.
.
For B_{d+i} => $2^{(i-1)}$ nodes

Now at max,
d+i = log n (max B_k, n here is the number of nodes in the “completely clean” Binomial tree i.e. the number of clean nodes)
Therefore => i = log n - d

Therefore sum of above equations => 1+1+2+4+8+.....+ $2^{(i-1)}$ = 2^i
Substituting i = log n - d =>
$2^i = 2^{(\log n - d)}$
$= (n/2^d)$

So, upper bound total cost that ExtractMin needs to pay for, of a dirty node of degree d =
(No. of nodes of degree d)*(Cost of 1 node of degree d)

$(n/2^d)*(d- c*f(n))$

Now, d can vary from 0 to log n.

But for d=0 to c*f(n) , we can do it for free as we have enough number of coins (we had assumed that k>=c*f(n)).

So, we need to find the cost from d=c*f(n) to log n.

So total cost that will be charged to extract min =

$$\sum_{d=c*f(n)}^{\log n} (n/2^d)*(d-c*f(n)) = \sum_{d=c*f(n)}^{\log n} (n*d)/2^d - \sum_{d=c*f(n)}^{\log n} (c*n*f(n))/2^d$$

For $\sum_{d=c*f(n)}^{\log n} (n*d)/2^d$

$$\Rightarrow \quad n^* \sum_{d=c*f(n)}^{log\ n} d/2^d \quad \text{(taking n outside as it is independent of d)}$$

$$\leq n^*2 \qquad \text{(Since } \sum_{d=0}^{infinity} d/2^d = 2\text{)}$$

and we are concerned with finding only an upper bound)

For $\sum_{d=c*f(n)}^{log\ n} (c^*n^*f(n))/2^d$

$$\Rightarrow c^*n^*f(n) \sum_{d=c*f(n)}^{log\ n} 1/2^d \qquad \text{(taking c*n*f(n) outside as it is independent of d)}$$

$$\Rightarrow c^*n^*f(n)\ (\ (1/2^{c^*f(n)}\ )\ ^* \ 2^*(\ 1 - (½)^{(log\ n\ -\ c^*f(n)\ +\ 1\ )}\ )$$

$$\Rightarrow c^*n^*f(n)^*2\ (\ 2^{-c^*f(n)} - 2/n\ )$$

$$\Rightarrow 2^*c^*n^*f(n)^*2^{-c^*f(n)} - 4c^*f(n)$$

Combining above results

$$\sum_{d=c*f(n)}^{log\ n} (n^*d)/2^d - \sum_{d=c*f(n)}^{log\ n} (c^*n^*f(n))/2^d = 2^*n - 2^*c^*n^*f(n)^*2^{-c^*f(n)} - 4c^*f(n)$$

Therefore, overall amortized cost =

$$O\ (\ 2^*n - (\ 2^*c^*n^*f(n)^*\ (2^{-c^*f(n)} - 4c^*f(n))\ )\ ) + O(log\ n)$$

[the O(log n) is for converting linked list to array representation, performing an extract min task and back again]

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

Maintain a min pointer and keep updating the pointer, when decrease key or insert is done. This way we do not have to spend any extra time finding the minimum. During extract min, we

can lose track of the min pointer in the cleaning phase, but when converting the linked list to array form, we go through the trees in the Heap and keep adding to the array, and linking and carrying if necessary. In the sequence of operation we can again keep track of min pointer here. After extract min is done, we can update the min pointer.
So at point of time(except cleaning phase, but it should be fine, as it will be followed by extract min from lazy union), we will have the min pointer. So no extra time is taken to maintain this pointer. And finding minimum, will be just returning this pointer.