Homework #2

(Due: Nov 10)

Group Number: 19

Group Members		
Name	SBU ID	% Contribution
Chaitanya N. Kalantri	111446728	33.33
Neel Paratkar	111483570	33.33
Rohan Karhadkar	111406429	33.33

Collaborating Groups

Group Number	Speci cs (e.g., speci c group member? speci c task/subtask?)
52	Rahul Bhansali, Kiran Kasarapu (Discussed about Question 1).
4	Siddharta Chhabra (Discussed about question 2)

External Resources Used

	Speci cs (e.g., cite papers, webpages, etc. and mention why each was used)
1.	http://www.geeksforgeeks.org/binomial-heap-2/
2:	http://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/

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

(a)

Step 4:
In the step 4 for finding the median. We would require to compute these steps, considering 3 numbers: (Median of A[1], A[2], A[3])

```
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
```

Hence, there will be 3! Combinations of the above algorithm.
Of which 6 combinations will be:
4 combinations require 3 comparisions, when the middle element is not the median.
2 combinations require 2 comparisions, when middle element is the median.

Therefore the total number of combinations will be:
2*2 + 4*3 = 4 + 12 = 16

Step 5:
In the step 5, we have the remaining A[3:] numbers. Using Quick Sort algorithm method we can arrange the data in the format.
Numbers before A[k] is less than A[k] and numbers after A[k] are greater than A[k].
Which can be written as:
$A[k]=x$, where $k \in [1, n]$
$A[i] < x$, where $i \in [1, k-1]$

A[i] > x, where i ∈ [k + 1, n]

Therefore, total number of combinations will be:
Over here we will be comparing (n-3) elements. Thus, each case will require (n-3) comparisons. And there are 6 combinations. Hence, total number of comparisons is 6 *(n-3).

Therefore, average number of comparisons of the 6 combinations is:
*= (16+ 6 * (n-3)) / 6*
= 8/3 + n-3
= (n - ⅓)

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

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

Hence, Proved.

(b)

The algorithm can be divided into 3 steps:
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 an average, as proved above.
Here, if k is the median. That is the pivot is k. Then, the possible combinations will be calculated as mentioned in the below steps:
There will be k-1 elements smaller and n-k larger elements.
Number of ways to select an element smaller than k is: *(k-1)C1*
Number of ways to select an element larger than k is: *(n-k)C1*
Number of ways to select the kth element is : *1C1*

Therefore, the total number of permutations where k occurs as the median:
*((k-1)C1) * ((n-k)C1) * (1C1) * 3! * (n-3)!*

Generalizing for all the elements; from step 6 and 7; we get:
For a given k:
Permutations when k is the
median = *(t_{k-1} + t_{n-k})*

Number of permutations
where k is the median
= *((k-1)C1) * ((n-k)C1) * (1C1) * 3! * (n-3)!*

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

Average number of comparisons for all permutations where k is the median
= *((k-1)C1) * ((n-k)C1) * (1C1) * 3! * (n-3)! * (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) * ((n-k)C1) * (1C1) * 3! * (n-3)! * (t_{k-1} +t_{n-k}) / n!*
*n-⅓ + Σ (k=1 to n) (k-1) * (n-k) * 3! * (n-3)! * (t_{k-1} +t_{n-k}) / n!*
*n-⅓ + 3! * (n-3)!/n! Σ (k=1 to n)(k-1) * (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)

To Prove: T(z) = 12 (1−z) 2 T 0 (z) − 8 (1−z) 4 + 24 (1−z) 5
Given: tn: T(z) = t_0 + t1$_{z1}$ + t2$_{z2}$ + . . . + tn$_{zn}$ + ... (T(n) is the generating function)

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

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

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

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

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

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

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

Expanding the summation; we get,

$$g'''(x) = 12t_1(1 + 2z + 3z^2...) + 2t_2 z(1 + 2z + 3z^2...) + ...$$

$$= 12(1 + 2z + 3z^2...)(t_1 + 2t_2 z + 3t_3 z^2 + ...)$$

$$= 12 \sum_{n=1}^{\infty} (n+1) z^n * (t_1 + 2t_2 z + 3t_3 z^2 + ...)$$

$$= 12 \frac{1}{(1-z)^2} * (t_1 + 2t_2 z + 3t_3 z^2 + ...)$$

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

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

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

Similarly, for T(z):

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

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

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

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

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

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

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

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

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

(d)

$$T'''(z) = \frac{12}{(1-z)^2}T'(z) + \frac{24}{(1-z)^5} - \frac{8}{(1-z)^4}$$
$$(z-1)^2T'''(z) - 12T'(z) = \frac{-24}{(z-1)^3} - \frac{8}{(z-1)^2}$$

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

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

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

Given: $T(0) = 0,\ T'(0) = 0,\ T''(0) = 2$

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

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

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

Substitute z = 0
We get C1 = 1

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

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 = $\frac{2}{735}$

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

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

On rearranging, we get the following

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

Hence, Proved.

(e)

From part(d); we get:

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

Taking Binomial coefficient of Z^n in different elements of the above equations;
The element $\frac{2}{735}(z-1)^5$ will become $(-1)^n \frac{2}{735}$ (5Cn)
The element ⅕ will become ⅕ (0Cn)

The rest of the element $= \frac{-3}{7}[4ln(1-z) + \frac{28}{9}z + \frac{29}{63}] (z-1)^{-2}$

$T(z) = \frac{-12}{7}ln(1-z)(z-1)^{-2} - \frac{4}{3}z(z-1)^{-2} - \frac{29}{147}(z-1)^{-2}$
$= \frac{-12}{7}ln(1-z) - \frac{12}{7}z)(z-1)^{-2} + (\frac{12}{7}z)(z-1)^{-2} - \frac{4}{3}z(z-1)^{-2} - \frac{29}{147}(z-1)^{-2}$

As we know that, the coefficient of Z^n in $(ln(1-z)-z)(z-1)^{-2}$ = $(n+1)H_n - 2n$
Therefore, the coefficient of the above term is:
$(\frac{-12}{7}(n+1)Hn) + \frac{12}{7}n$
And the coefficient for $-\frac{4}{3}z(z-1)^{-2}$ is $-\frac{4}{3}n$
The coefficient for $-\frac{29}{147}(z-1)^{-2}$ is $-\frac{29}{147}(n+1)$
Therefore, taking coefficient of Z^n of all the terms will give,
$t_n = \frac{12}{7}(n+1)H_n - \frac{159}{49}n - \frac{29}{147} - (-1)^n \frac{2}{735}(5Cn) + \frac{1}{5}(0Cn)$
where $Hn = \sum_{k=1}^{n} (1/k)$ is the n^{th} Harmonic number.

(f)

We know that, $H_n = ln(n) + 0(1)$
Putting value of H_n to the solution of part (e)
$t_n = \frac{12}{7}(n+1)H_n - \frac{159}{49}n - \frac{29}{147} - (-1)^n \frac{2}{735}(5Cn) + \frac{1}{5}(0Cn)$
where $H_n = \sum_{k=1}^{n} (1/k)$
We get,
$t_n = \frac{12}{7}(n+1)(ln(n) + 0(1)) - \frac{159}{49}n - \frac{29}{147} - (-1)^n \frac{2}{735}(5Cn) + \frac{1}{5}(0Cn)$

This will give us the result that,
$t_n = Ø(nlogn)$

Q2. (a) Following are the time complexities of every line of code:

D.Insert(x):

Line No	Code	Time Complexity
1	Insert(D.Q 2 , x)	O(log N)

Since the enqueue for standard binary heap is O(log N), line 1 will take O(log N) time for its execution.

Therefore, the total time complexity will be given by
T(N) = **O(log N)**

D.InvSucc(x):

Line No	Code	Time Complexity
1	while Find-Min(D.Q 1) ≤ x do	O(log N)
2	Extract-Min(D.Q 1)	O(log N)
3	while Find-Min(D.Q 2) ≤ Find-Min(D.Q 1) do	O(log N)
4	if Find-Min(D.Q 2) = Find-Min(D.Q 1) then	O(1)
5	Extract-Min(D.Q 1)	O(log N)
6	Extract-Min(D.Q 2)	O(log N)
7	y ← Extract-Min(D.Q 1)	O(log N)
8	return y	O(1)

Here, the while loops at line 1 and 3 run for a max of log N times because we remove all the composites from Q1 less than the min of Q2 and later remove them from Q2. So basically this function returns the next prime number or a multiple of prime number not yet found. And since the distance between 2 primes is about log N, the loops will run log N times.

Therefore, the total time complexity will be given by
T(N) = O(log N * log N) + O(log N + log N * log N + log N * log N) + O(log N) + O(1)
= **$O((\log N)^2)$**

D.Save(x):

Line No.	Code	Time Complexity
1	Push(D.S, x)	O(1)

Therefore, the total time complexity will be given by
T(N) = **O(1)**

D.Restore( ):

Line No.	Code	Time Complexity
1	while D.S = ∅ do	O(N)
2	x ← Pop(D.S)	O(1)
3	Insert(D.Q 1 , x)	O(log N)

Therefore, the total time complexity will be given by
T(N) = O(N + N log N)
= **O(N log N)**

Q2. (b) Following are the time complexities of each line of code:

Line No.	Code	Time Complexity
1	create support data structure D	O(1)
2	D.Init(N), p ← 1	O(N log N)
3	while p ≤ $\sqrt{N}$ do	O(log N)
4	p' ← D.InvSucc(p)	O(N log N)
5	print p'	O(1)
6	p ← p' , q ← p'	O(1)
7	while pq ≤ N do	O(log log N)
8	for r ← 1 to $\log_p (N/q)$ do	O(log log N)
9	D.Insert(p^r q)	O(log N)
10	q ← D.InvSucc(q)	O(N log N)
11	D.Save(q)	O(log log N)

12	D.Restore( )	O(N log N)
13	while p ≤ N do	O(N)
14	p ← D.InvSucc(p)	O(N log N)
15	if p ≤ N then print p	O(1)

From the code, we observe that the line 8 runs only for the multiples of prime numbers. The distance between two prime numbers is log N, so line 8 runs for log log N times, taking into consideration the outer loop as well.

We also observe that whatever values we have pushed into the queue2, we pop them back into queue1 as they may not be composites and we want to process them again to check if they are prime numbers.

Note that we do not insert all the values in the stack. Also, these values will only be of the order of log log N. We only save those q values in the stack that may be prime numbers or that are not prime but a multiple of some prime not yet found.

Lines 3 and 13 in total will run for N times looping once over all non primes(composites) and multiplying it with the Insert operation, we get a total time complexity of T(N) = **O(N log N).**

Q2. (c) Let us assume that the **D.Insert** function, is given 2 * log N coins. As the underlying data structure is a binary heap, inserting an element or the enqueue operation requires log N units of work. So, in the insertion process we will spend log N coins. Also, we will be keeping the remaining log N coins in the position where we inserted the element. Thus, the system gains log N coins each time we call the D.Insert function. And we can say that the amortized time complexity is θ **(log N)**.

The **D.InvSucc** function requires log N units of work at line 2, 5, 6 and 7. It doesn't matter how many times we dequeue an element, just note that every dequeue will require log N amount of work to be done. However, for every position in either queue 1 or queue 2, there are log N coins already present as they were added when we inserted these elements in the queue. So, the D.InvSucc function itself did not spend any coins of its own. This is because there will be no dequeue if there is no enqueue. So, we can say that the amortized cost of this function is θ **(1)**.

Now, assume that each of the **D.Save** operation is given 2 + 2 * log N coins. We require 1 unit of work to either push the element into stack or pop it. So, when we push the element into stack inside the D.Save function, we are using 1 coin. Now, we keep the other 1 + 2 * log N coins in the stack's position where it is pushed. This way, the system accumulates coins every time we use D.Save function. (Note that 1 + 2 * log N coins are gained for every push operation.) So, the amortized complexity of D.Save will be θ **(log N)**.

Now, assume that **D.Restore** is not provided with any coin for its operations. When we use the D.Restore function, it goes on popping the elements and inserting it to the first queue. However, we will need to do some work to pop the element and insert it to the queue. This work worth 1 unit and log N units respectively and will require 1 + log N coins. However, this function will not have to spend any coins of its own. This is because as mentioned earlier, the system already has 1 + 2 * log N coins for every value in stack. We can use 1 coin from this to pop and log N coins to enqueue the value to the first queue. Also, we will keep the remaining log N coins in the position where we just inserted the element. Thus, the system will lose coins as the stack's contents decreases. In this process, as the D.Restore does not spend any coins of its own, so we can say that the amortized time complexity of this function is θ **(1)**.

Q2. (d) Following are the time complexities of each line of code:

Line No.	**Code**	**Time Complexity**
1	create support data structure D	θ (1)
2	D.Init(N), p ← 1	θ (log N)
3	while p ≤ $\sqrt{N}$ do	θ (N)
4	p' ← D.InvSucc(p)	θ (1)
5	print p'	θ (1)
6	p ← p' , q ← p'	θ (1)
7	while pq ≤ N do	θ (log log N)
8	for r ← 1 to $\log_p$ (N/q) do	θ (log log N)
9	D.Insert(p^r q)	θ (log N)
10	q ← D.InvSucc(q)	θ (1)
11	D.Save(q)	θ (log N)
12	D.Restore( )	θ (1)
13	while p ≤ N do	θ (N)
14	p ← D.InvSucc(p)	θ (1)
15	if p ≤ N then print p	θ (1)

From the code, we observe that the line 8 runs only for the multiples of prime numbers. The distance between two prime numbers is log N, so line 8 runs for log log N times, taking into consideration the outer loop as well.

We also observe that whatever values we have pushed into the queue2, we pop them back into queue1 as they may not be composites and we want to process them again to check if they are prime numbers.

Note that we do not insert all the values in the stack. Also, these values will only be of the order of log log N. We only save those q values in the stack that may be prime numbers or that are not prime but a multiple of some prime not yet found.

Therefore, the total time complexity will be given by

$T(N) = \theta(1) + \theta(\log N) + \theta(N^* (1 + 1 + 1 + \log\log N + 1 + \log N + 1)) + \theta(N * (1 + 1))$

$= \theta$ **(N log N)**

Q2. (e) Now, the underlying data structure is a binomial heap with Insert, Find-Min and Extract-Min operations in O (1), O (1) and O (log N) amortized time, respectively.

Let us assume that the **D.Insert** function, is given 1 coin for every time it is called. As the underlying data structure is a binomial heap, inserting an element requires 1 unit of work. So, in the insertion process we will spend 1 coin. Thus, we use constant amount of coins each time we call the D.Insert function. And we can say that the amortized time complexity is **O(1)**.

The **D.InvSucc** function requires log N units of work at line 2, 5, 6 and 7. Note that every dequeue will require log N amount of work to be done as the amortized cost of Extract-Min operation for binomial heap is O(log N). Let the number of elements dequeued from queue1 be c1 and number of elements dequeued from queue2 be c2. So, total elements dequeued will be equal to (c1 + c2) * log N. But, total number of elements dequeued (c1 + c2) will always be upper bounded by log N as the function returns the next possible prime number or a multiple of prime number not yet found; and the distance between two prime numbers is log N. So let us assume that this function is given log N * log N coins whenever it is called. So, we can say that the amortized time complexity of D.InvSucc function is $\mathbf{O((\log N)^2)}$.

Now, assume that each of the **D.Save** operation is given 1 coin each time we call it. We require 1 unit of work to either push the element into stack or pop it. So, when we push the element into stack inside the D.Save function, we are using 1 coin. So, the amortized complexity of D.Save will be **O(1)**.

When we call the **D.Restore** function, it goes on popping the elements and inserting it to the first queue. However, we will need to do some work to pop the element and insert it to the queue. This work worth 1 unit each requiring 2 coins. We can use 1 coin to pop and 1 coin to enqueue the value to the queue1. In this process, the D.Restore spends 2 coins for every element in the stack. However, note that the stack only contains elements that may be prime or the multiples of primes that are not yet found. This quantity will always be less than or equal to log N as the distance between two prime numbers is approximately equal to log N. So we can say that the total coins spent by this function is 2 * log N; in other words, the amortized time complexity of this function is **O(log N)**.

Following are the time complexities of each line of code:

Line No.	Code	Time Complexity
1	create support data structure D	O(1)
2	D.Init(N), p ← 1	O(N)
3	while p ≤ $\sqrt{N}$ do	O(N)
4	p' ← D.InvSucc(p)	$O((\log N)^2)$
5	print p'	O(1)
6	p ← p' , q ← p'	O(1)
7	while pq ≤ N do	O(log N)
8	for r ← 1 to $\log_p (N/q)$ do	O(log log N)
9	D.Insert(p^r q)	O(1)
10	q ← D.InvSucc(q)	$O((\log N)^2)$
11	D.Save(q)	O(1)
12	D.Restore( )	O(log N)
13	while p ≤ N do	O(N)
14	p ← D.InvSucc(p)	$O((\log N)^2)$
15	if p ≤ N then print p	O(1)

As in the previous cases, the value of amortized time complexity will be **O(N log N)**.

Solution 3.1
Lets assume that we add c.f(n), where c is a constant, [O(f(n))] coins at each decrease-key operation.
For calculating the amortized time for extract-min() operation, lets divide the problem into two parts.

Part1. Clean nodes < Dirty nodes
Consider a Binomial heap with all clean nodes formed after some insert operations, say m nodes. After certain k decrease-key() operations, there will be m clean nodes and i dirty nodes. After each decrease-key() operation, we will add c.f(n) coins at the dirty node. Hence there will be i.c.f(n) coins present in the heap after k decrease-key() operations. Let us assume f(n) $\geq$ 2. Hence, for f(n) = 2, say, there will be atleast 2i coins in the heap. As mentioned in the question, in this case, we simply create a B_0 for every node and attach it to the heap. As clean nodes < dirty nodes, we know that 2i will always be greater than the number of clean nodes. Hence the operations of creating and attaching clean nodes as B_0 to the heap will be paid using these coins. As a result the amortized cost for doing this will be same as extract-min() operation of lazy-union binomial heal with no decrease-key() operation. It will be O(log n)

Part2. Clean nodes > Dirty nodes
According to the question, in such a case, while traversing through the linked list, when we encounter a dirty root, it removes the root and adds k children to the linked list. As dirty nodes is less than clean nodes, then there is possibility that the cost of inserting k children into the linked list cannot be paid by c.f(n) (as clean nodes > dirty nodes) coins at the dirty node. Hence, there can be k – c.f(n) children that cannot be places without incurring any cost. Hence for this step of the extract-min process, the cost incurred will be,

$$\sum_{k=1}^{\log N} A.(k - cf(n))$$

Eq. 1

Here, N = Total number of nodes in heap.
A = no. of dirty nodes of degree k.
For the worst cast amortized cost, we need to maximize A.
For a degree k, the number of nodes with degree k (which will be a B_k) can be calculated as follows:
In the linked list, we can disregard all the B_x for x < k.
For B_k tree -> 1 B_k tree
For B_{k+1} tree -> 1 B_k tree
For B_{k+2} tree -> 2 B_k trees
For B_{k+3} tree -> 4 B_k trees
Going on until , k+i = log N, for B_{k+i} tree -> 2^{i-1} B_k trees
Hence maximum number of trees with degree K can be -> $1+1+2+4+8\ldots+2^{i-1} = 1 + (2^i - 1) = 2^i$
But we know, k+i = log N. Hence i = log N - k
Hence maximum number of trees with degree K can be -> $2^{\log N - k} = 2^{\log N}/2^k = n/2^k$
$A = n/2^k$
Substituting in eq. 1:
$\sum_{k=1}^{\log N} A.\left(k - cf(n)\right) = \sum_{k=1}^{\log N} \frac{n}{2^k} \times (k - cf(n))$

L = n x $\sum_1^{\log N} \frac{k}{2^k}$ - c.n.f(n) $\sum_1^{\log N} \frac{1}{k}$
Lets consider first part of the subtraction:
B = n x $\sum_1^{\log N} \frac{k}{2^k}$
We know that for $\sum_1^{\infty} \frac{k}{2^k}$ = 2. So B is upper bounded by 2.

Let D = c.n.f(n) $\sum_1^{\log N} \frac{1}{2^k}$
Here, we know that for children of the k degree binary tree can be placed in the lined list using c.f(n) keys. Hence for k = 1 to cf(n) – 1, there will be no cost incurred.
Hence we can write D as: c.n.f(n) $\sum_{cf(n)}^{\log N} \frac{1}{2^k}$

The sigma part of D is geometric progression. Solving it we get:
D = c.n.f(n) ($2^{1- c.f(n)}$ + $2^{1- c.f(n)}$. N)
Hence
L = O(2.n) - c.n.f(n) ($2^{1- c.f(n)}$ + $2^{1- c.f(n)}$. N)
The cost after expanding the trees into linked list can takes upto $\theta(\log N)$ time as it is same as cost of extract-min() operation in lazy-union binomial heap.
Hence total Amortized cost for extract-min operation will be :
Cost = cost for expanding the linked list + cost for extract min after expanding the linked list (converting to array and back to linked list)
Amortized cost = O(2n – c.n.f(n) . $2^{1- c.f(n)}$(1+N)) + $\theta(\log N)$
After considering asymptotic bounds : Amortized cost = O(n.f(n).$2^{1- c.f(n)}$. N)

Solution 3.2:
We will maintain a pointer to the min value in the linked list. For every B_k tree that is added to the heap, the root of that tree will be the minimum value in that tree.
1.For insert operation, for first insertion, when the linked list has no node, the first node will have the minimum value. As we insert, we can compare and update the value of minimum in constant time. O(1)
2. For every decrease key operation, we can again compare the new key with the already maintained minimum pointer and update the pointer in constant time.
3. For extract-min()operation, after creating the binomial heap in array format, we remove the minimum number. During the phase of creating the linked list from the heap representation, we do a sweep of every element in the binomial heap in atmost O(log N) time. During this sweep, we can keep checking the root of tree B_k at each stage and update the pointer to the min value in the binomial heap. Thus even during the extract-min() operation we can maintain the min pointer in O(1) time.

When a call to Find-min() is made, the minimum value can be simply returned from the maintained pointer in constant time.
Hence in each operation as we can see, we need constant time to maintain the pointer to the minimum value.

As a result maintaining the pointer to minimum value and updating it during each operation INSERT, EXTRACT-MIN, DECREASE-KEY will enable us to execute the FIND-MIN operation in O(1) time.