Soham Niranjan Sawant
ssawant3

1. We are going to prove the correctness algorithm of the bubble sort. This algorithm has two nested for loops. The bubble sort basically tries the sort the array by shifting the smallest value to left of array and largest value to the right of array.

```
BUBBLE-SORT(A)
    for i = 1 to A.length-1
        for j = A.length downto i + 1
            if A[j] < A[j-1]
                exchange A[j] with A[j-1]
```

a) The inner loop shifts the numbers based on their magnitude in the array and the inner loop runs from A.length to i+1. Basically, the inner loop is trying to find a smallest element in the subarray and shift it to leftmost index possible in the sub array. Hence for inner loop the loop invariant property will be that there exists a value smaller (can be more than 1 if values with same value exist) than the other elements.

   Initialization: Before the first iteration of the loop, smallest value exists in that subarray A[i+1..A.length]. This is true

   Maintenance: The loop runs from A.length to i+1 where the value of j decrements by 1 on every iteration. If the value in the first iteration at j is smaller than the value at j-1 then it is swapped. Same condition is checked for all the iterations of for loop and if found true swapping is done. In the end we can see that the small element of the subarray gets shifted to the left side of the subarray therefore the loop invariant is preserved.

   Termination: The loop terminates after the iteration when value of j=i+1 and result is that smallest value is placed at a[i] of the array.

b) The inequality states that array should be such that the array is sorted. In the outer loop the smallest element is placed in the first index in the first iteration and so on. Hence, we can say that at the start of each iterations i-1 elements are in the sorted order. Hence this our loop invariant property.

   Initialization: When i=1, which means that i-1 elements are sorted which is true.

   Maintenance: After the first iteration, the smallest element in the A[1: A.length] is at A[1]. After the iteration where i=2 the smallest value in A[2: A.length] is placed at A[2] which is the second smallest value in the whole array. This goes on for all the iterations hence preserving our loop invariant property.

   Termination: The loop end after the i reaches A.length which means that the smallest A. length-1 values of the array are placed in sorted order. Hence the array is sorted.

c) The worst case for the algorithm will be that outer loop runs from 1 to n-1 where n is A.length. when i=1 the inner loop will do n-1 comparisons, i=2 the inner loop will do n-2 comparisons. When i=n-1 then the inner loop will do 1 comparison. Hence the worst-case scenario will be (n*(n-1))/2. Therefore, it gives O(n^2). But as compared to insertion sort the bubble sort does way more write functions. Hence for the same input size insertion sort will do better than bubble sort. But in terms of comparisons same number of comparisons takes place in both algorithms.

2.

a)

n	1	2	3	4	5
Return value k	3	6	12	12	24
Return value l	0	4	12	16	30
No of Multiplications	1	4	9	12	20

b) The k value is initialised to 1 after every iteration of for loop. And the while loop runs log n times because k value is doubled after every iteration. And the condition of while loop k<n. Hence the value of k:
3*2^ (no of times while loop runs)

No of times while loop runs will be an integer value hence, we use floor function to log n value. Then in the example where n=3 we get the answer as 3*2^floor (log n) which is 3*2^1. Which gives us 6 but we want the answer as 12 which can be got by the using the next power of 2. Hence, we need to add 1 to floor value. So, floor(n)+1 is nothing but the ceiling value. Hence, we use the ceiling function.

Value of k= 3*2^ceil (log n)

c) The l value is incremented every iteration of while loop and also for every iteration of for loop when the while condition is true. And the while loop runs log n times because k value is doubled after every iteration and the for loop runs n times. Hence the value of l:

Value of l=2*n*ceil (log n)

d) The no of multiplications is nothing but the sum of number of times the value of k is changed which is n times in for loop and log n times in while loop. Hence the no of multiplications is:

no of mulitplications= n+log n

*log n means log n to base 2

3.

n^$2n$	n^3 $\log(n!)$	(n-2)!	n^3 $+ n$^2 $\log n$	$n^2\log(n)$	$n + \log(\log(n))*$	$\frac{n}{2} + (\log(n))^{0.5}*$	$2^{\log(n)*}$	$n/\log(n)$

4.

a)

$nlg(n) + lg(n) \in \omega(lg(n))$

$f(n) \in \omega(g(n))\ iff\ for\ every\ c > 0\ there\ is\ an\ n0 > 0\ such\ that\ 0 \leq cg(n) < f(n)\ for\ all\ n \geq n0.$

Hence $f(n) = nlgn + lg(n)$ and g(n)=$lg(n)$

$nlgn + lgn > clgn$

$n + 1 > c$

$n > c - 1$

$Therefore\ n_0 = c - 1$

$Also, clgn >= 0\ hence\ n >= 2$

Therefore, we have $n_0 = c - 1$ and n>=2. To make sure that for all values of c>0 we have values of n which are greater than 2 we add scalar value 3 to RHS of $n_0 = c - 1$

$Finally\ we\ have\ n_0 = c + 2$

$Hence\ the\ inequality\ is\ proved\ true\ as\ c > 0\ and\ for\ values\ of\ n >= n0$

b)

$4n^2$ + n + nlg(n)∈Θ(n^3)

$f(n) = \Theta(g(n)$ *if there are two positive constants* $c1$ *and* $c2$ *and a value* $n0$*, such that* $c1.g(n) >= f(n) >= c2.g(n)$ *for all* $n \geq n0$

$f(n) = 4n^2$ + n + nlg(n)

$g(n) = n^3$

4n^2+n + nlg(n)>=c2.n^3 *where* $n > 0$

Let's assume $lg(n)$ *as* n *and divide the LHS and RHS by* $n3$

4/n + 1/n^2+1/n>=c2

5/n + 1/n^2>=c2

If we assume n *tends to* ∞*. Both the values* $5/n$ *and* $1/n2$ *tends to zero*

$0 >= c2$

But according to the definition $c2$ *is positive constant hence we prove that*

$4n^2$ + n + nlg(n)∈ Ω(n^3) is false

Finally, as *(*g(n)) = O(g(n)) ∩ Ω(g(n)) *but* Ω(g(n)) does not stand true. So, $4n^2$ + n + nlg(n)∈Θ(n^3) is false.

c)

$2n - n/lg(n) \in \Omega(n)$

According to formal definition of big omega we need to prove that

$f(n) \in (g(n))$ *iff there exist* $c, n0 > 0$ *such that* $f(n) \geq cg(n) \geq 0$ *for all* $n \geq n0$.

$2n - n/lg(n) >= cn$ $n > 0$

We assume $n/lg(n)$ *as* n *to make this assumption* $n >= 2$

$2n - n >= cn$

$n >= cn$

We got $c = 1$ *and* $n0 = 2$ *hence the inequality is proved true such that* $n >= 2$

$2n - n/lg(n) \in \Omega(n)$

5.

<u>Pseudo Code</u>

```
 x=1
 def findpower(a, n):
    if n is equal to 0:
        return 1
    else n is odd:
        x = findpower(a,(n-1) / 2)
        return a*x**2
    else n is even:
        x = findpower(a, (n)/ 2)
        return x**2
```

Example in the case of findpower(4,4)

1. In the first step as the value of n=4 which is even the else n is even condition is selected and findpower(4,2) is added in the stack.
2. Then the second step the value of n=2 which is also an even hence the function called will be findpower(4,1)
3. Then the function findpower(4,1). In these case n=1 which is odd in which the else n is odd condition is run which will call the findpower(4,0)is called
4. Findpower(4,0) the base case is called and it returns 1 and doesn't call any additional functions.
5. Then the next function in the line findpower(4,1) returns 4 and updates the x value to 4
6. Then the next function which is findpower(4,2) and it returns 4^2 which is 16 and it updates the x as 16
7. Then the next function findpower(4,4) returns 16^2 which is 256 our final and required value.

The recursion divides the problem in the size 2. The sub problems computes one sub problem per recursive call. Hence the recurrence relation for the above problem would be

T (n)= {c_0 if n < s
{a*T (n/b) + D(n) + C(n) otherwise

a=1……………………no of sub problems in per recursive call.
b=2 ……………………size of sub problem
D(n)=1 ………………. time required for deciding which sub problem to solve (checking if it is odd or even)
C(n)=0 ……………not combining solution.

T(n)= {1 n=0
{1* T(n/2) +1 otherwise

By using master theorem, we will find big-theta of the code

$$T(n) = \begin{cases} c_0 & \text{if } n < s \\ aT(n/b) + f(n) & \text{otherwise} \end{cases}$$

Master Theorem

$f(n) \in O(n^{\log_b a - \epsilon})$ (for some $\epsilon > 0$)	$T(n) \in \Theta(n^{log_b a})$
$f(n) \in \Theta(n^{\log_b a})$	$T(n) \in \Theta(n^{log_b a} \lg n)$
(for some $\epsilon > 0$) $f(n) \in \Omega(n^{\log_b a + \epsilon})$ and (*)	$T(n) \in \Theta(f(n))$

Our $\log_b a = \log_2 1 = 0$ and $n^{\log_b a} = n^0 = 1$ and $f(n)=1$. The master theorem case 2 applies here.

So, $T(n)=\Theta(n^{0} * \lg n)$

$T(n)=\Theta(\lg n)$