<u>DAA Assignment</u>

NAME: AAKARSH SATISH **UNITY ID: 200536723** **mailid:asatish2@ncsu.edu**

1) Learn about bubblesort and practice how loop invariants are used to prove the correctness of an algorithm. Please re-read Section 2.1 in our textbook and solve problem 2-2 Correctness of bubblesort subproblems b-d. In d) use the compare operation ‘<’ as the only basic operation. Compute and justify the worst-case running time of bubblesort and argue which of the two sorting algorithms you would prefer.

b) **<u>Loop Invariant</u>**: The subarray A[j…n] has the values from A[j….n] at the start of each loop but may or may not be in different order. Here the smallest element is the first element.

<u>Initialisation</u>: When J is equal to A.length that is Length of A, the last element is already sorted which is the smallest number in a list of 1 element. The subarray only consists of one element which has already been sorted which inturn shows that the loop invariant holds prior to the first iteration of the loop.

<u>Maintenance</u>: This loop makes sure that if A[j-1] is greater than A[j] then a swap is made. After the whole iteration is made the smallest element is the first element in the array and the subarray increases by 1.

<u>Termination:</u> The loop terminates when j is equal to i+1. This happens so because in every iteration of the loop j value decrements from A.length (Length of the size of A) down to i+1. While this is happening the length of the subarray increases by 1 and at the end of the iteration we find the smallest value number to be present as the first element of the array.

c) **<u>Loop Invariant:</u>** A[1…i-1] contains elements that will be smaller than the sorted sub array A[j…n-1]

<u>Initialisation:</u> Initialisation at first holds true because the array A[1,…,i-1] is an empty array. We consider an empty array to be sorted array. So this condition holds true.

<u>Maintenance:</u> In this after the execution of the innermost loop we see that the smallest element is placed in the first position of the array. A[i] will be the smallest element in the sub array A[i, …..n]. These will be sorted by the end of the loop.

Termination: The outer loop ends when the value of i is equal to A.length - 1. Once the loop terminates we can see that A [1…A.length-1] is arranged in such a way that the elements in the array in a sorted order where the first element being the smallest value element and the last element being the highest valued element in the array.

d) The Worst case time complexity of the bubble sort is n^2. But the comparison '<' basic operation for bubble sort comparison operation for each element is n for each element when the array is sorted in a reserve order. So the big O of bubble sort is O(n^2). Even in case of insertion sort the number of comparisons '<' is the same as the bubble sort in worst case. **But the insertion sort performs much better in the best case scenario. Hence I would prefer using insertion sort rather than bubble sort.**

2) Practice counting basic operations and analyzing algorithms. Assume n > 0 and consider the following algorithm.
a) For n = 1,2,3,4,5, what values for k and l are returned in line 8? How many multiplications ("*") does the algorithm perform for computing these values?

n	1	2	3	4	5
Return Value(k)	3	6	12	12	24
Return value (l)	0	4	12	16	30
Number of Multiplications	1	4	9	12	20

b) As a function of n, what is the value of k returned in line 8? Justify your results.

Here we can see that when:
n = 1 k = 1*3
n = 2 k = 1*2*3
n = 3 k = 1*2*2*3
n = 4 k = 1*2*2*3
n = 5 k = 1*2*2*2*3
n = 6 k = 1*2*2*2*3
n = 7 k = 1*2*2*2*3
n = 8 k = 1*2*2*2*3

We get the pattern as shown above
Since this pattern follows as shown above with multiplications of 2 repeated.

So with the pattern of two's repeating we can see that it is 2^(log(n)). But it cant be only 2^(log(n)) as it gives out decimal values. So we need to add ceiling function to get those 2 patterns. Now if we multiply by 3 we get a function of n which satisfies the above pattern.

F(n) = 3* ceiling (2^(log n))
F(n) = 3* ⌈ 2^(log n)⌉

c) As a function of n, what is the value of l returned in line 8? Justify your results
Here we can see that when :
n = 1 l = 0
n = 2 l = 2*2
n = 3 l = 1*(2+2)*3
n = 4 l = 1*(2+2)*4
n = 5 l = 1*(2+2+2)*5
n = 6 l = 1*(2+2+2)*6
n = 7 l = 1*(2+2+2)*7
n = 8 l = 1*(2+2+2)*8

We get the pattern as above
Here the 2 is added in such a way for every n value. The 2's can be obtained as shown above with ceiling of (2*log(n)) as shown. If we multiply this value by n we get the equation as a function of n

F(n) = n *2* ceiling (log(n))
F(n) = n*2*⌈log(n)⌉

d) As a function of n, how many multiplications ("*") does the algorithm perform? Justify your results.
Here we can see that when :
n = 1 m = 1 + 0
n = 2 m = 2 + (1)*2
n = 3 m = 3 + (1+1)*3
n = 4 m = 4 + (1+1)*4
n = 5 m = 5 + (1+1+1)*5
n = 6 m = 6 + (1+1+1)*6
n = 7 m = 7 + (1+1+1)*7
n = 8 m = 8 + (1+1+1)*8

In order to get this pattern we get the function of n as: **n*⌈log(n)⌉ + n so n(⌈log(n)⌉ + 1)**

3) Practice working with asymptotic notation. Rank the following functions by order of asymptotic growth; that is, find an arrangement g1, g2, … of the below functions with g1(g2), g2(g3) …. If functions are asymptotically equivalent, i.e., gk (gk+1) mark them by a "*". The function lg indicates the binary logarithm

1	2	3	4	5	6*	7*	8*	9
n^(2n)	(n-2) !	n^3 lg(n!)	n^3+n^2lg(n)	nlg(n^n)	n+lg(lg(n))	2^lg(n)	n/2+sqrt(lg(n))	n/lg(n)

6, 7, 8 has the time complexity which is Linear.

4) Practice working with asymptotic notation. Prove or disprove rigorously (i.e., give values for c and n0 that will make your argument work, or show a contradiction that disproves the statement) using the formal definitions of Θ, , and . The function lg indicates the binary logarithm

a) nlg(n) + lg(n) ∈ ω(lg(n))

According to the definition of little omega **ω :**
f(n) ∈ ω(g(n)) iff for every c > 0 there is an n0 > 0 such that 0 ≤ cg(n) < f (n) for all n ≥ n0

Now in the given equation

0 <= clog(n) < nlog(n) + log(n)

Lets take the first condition :

0 <= clog(n)
From this we can see that n can be equal to 1 as well, to satisfy the condition.
If n =1
0 <= c* log 1
0< = c* 0
0 = 0
In this case so here we can get to know c> 0 and n >=1

But now in the second equation :
$nlog(n) + log(n) > c * log(n)$
From this we can verify that n cant be equal to 1 as the equation becomes
$1*log(1) + log(1) > c * log(1)$
$1*0 + 0 > c*0$
0 + 0 is not greater than 0 so n cant be 0
Now divide by log on both sides
$n + 1 > c$

N > 1 and c < 3
If we substitute n =2

So n0 = 2
And c =2

b) $4n^2+n+nlg(n) \in \Theta(n^3)$

According to the definition of theta **Θ:**
$\mathbf{\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$

For Big 0 condition
According to the definition of Big 0:
$f(n) \in O(g(n))$ iff there exist c, n0 > 0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$

$c1*n^3 <= 4n^2 + n + nlog(n) <= c2*n^3$
Lets take the equation
$4n^2 + n + nlog(n) <= c2*n^3$
We can write this equation as :
$4n^2 + n + nlog(n) <= 4n^3 + n^3 + n^3$
$4n^2 + n + nlog(n) <= 6n^3$
$n >= 1$
c2 = 6, n0 = 1

Now for **Ω**
According to the definition of omega Ω :
$f(n) \in \Omega(g(n))$ iff there exist c, n0 > 0 such that $f(n) \geq cg(n) \geq 0$ for all $n \geq n0$.
$4n^2 + n + nlog(n) >= c1n^3$
Now we put the higher order power for the equation i.e (n^3)

$4n^2 + n + nlog(n) >= 4n^3 + n^3 + n^3$

$4n^2 + n + n\log(n) >= 6n^3$

Now c1 = 6

But if we replace n as 1 the equation does not satisfy and for all values of n > 1 , the equation fails to satisfy.

So the equation $4n^2+n+n\lg(n) \notin \Theta(n^3)$

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

According to the definition of omega **Ω :**

f (n) ∈ Ω(g(n)) iff there exist c, n0 > 0 such that f (n) ≥ cg(n) ≥ 0 for all n ≥ n0.

2n - n/log(n) >= c1n

Here n cant be equal to 1 as n/log(n) term becomes infinity.

So n > 1

Now with the equation :

2n - n/log(n) >= 2n - n

2n - n/log(n) >= n

Now if we replace n by 2

We satisfy the equation

2*2 - 2/log2 >= 2

4 - 2

2 = 2

Here we can verify that n0 = 2 , c = 1

5) How to design, analyze, and communicate algorithms. Describe a recursive O(lg(n)) algorithm which computes (a)n, given a and n. You may assume that a is a positive real number, and n is a positive integer, but do not assume that n is a power of 2. Please follow the above instructions to describe your algorithm. The function lg indicates the binary logarithm.

<u>Pseudocode:</u>

Function a_power_n (a,n)

```
If n = 0                                   # 1 line (In our case we do not need
    then return 1                              this condition as n > 0)
else If n = 1                              # 3 line
    Then return a
m = Function a_power_n (a,n / 2)           # 5 line
If n % 2 == 0                              #even values of n
   ans = m * m
else                                       #odd values of n
   ans = a* m*m
Return ans
```

Tracking value and dry run :

Lets take a as 2 and n as 4.

- Lets call the function Function a_power_n(2,4) where a = 2 and n = 4
- Now this code runs and the line now 5 is run.
- The next stack call would be Function a_power_n (2, 4/2) = Function a_power_n (2, 2)
- And the next stack call would be Function a_power_n (2, 2/2) = Function a_power_n (2,1)
- Now we see that n becomes 1, so line number 3 is run and returns the a value which is 2 in our case.
- Now the case is even when n = 2 for the function call 'Function a_power_n (2, 2)' so ans will be assigned to 2*2 = 4
- Now the value 4 will be returned from the function call 'Function a_power_n (2, 2)'.
- Now the ''Function a_power_n (2, 4)' is traced back and m value is 4.
- Now as n = 4 , the code block for even is run again.
- Ans will be assigned to as 4*4 = 16
- The ans will be finally returned as 16 which is the correct answer for 2^4.

Complexity Analysis:

So the problem basically is tackled by divide and conquer method where the value of n is divided by 2 for every recursion call. So dividing the problem would take T(n/2) time and to conquer the problem it would take constant time. So the T(n) would be :

$T(n) = T(n/2) + 1$

So now we use master theorem to evaluate the function:

Here a = 1, b = 2 and the driving function is 1.

Calculate watershed function $\log_b a$
And replace a with 1 and b with 2.
We get 0. So n^0 is the watershed function which equals 1.

This falls under the second case and the Θ(n^($\log_b a$) lg n)
So in our case it would be T(n) ∈ Θ(n^0 log n)
= Θ(log n)

Proof with induction:

- For the base case n = 1,
  Let us assume that a = 3 for simplicity. So now the function call
‘Function a_power_n (a,n)’ would be ‘Function a_power_n (3,1)’. In this case it would return a which is 3. So 3^1 is 3. Hence this result holds true.

- Now lets assume that a^ k where k starts from 1 to n holds true. —-Step 2
- Inducution: In this case we need to prove that a^(n+1) holds true for all n values.
  Now we have 2 cases in the pseudocode mentioned above.
  Even values of n+1: For all the even values of n+1, this recursion stack calls
  ‘‘Function a_power_n (3,n+1/2)* Function a_power_n (3,n+1/2)“
  Odd values of n+1: For all the odd values of n+1, this recursion stack calls
  ‘‘Function a_power_n (3,n+1/2)* Function a_power_n (3,n+1/2)*a“.

  So based on the above assumption made in step 2, we see that k holds true for the function starting from 1 to n values. As n+1/2 is less than n, we can see that the function holds true and good for all values of k till n values. So we can say that the function holds good for n+1 value as well.

- **Hence by the proof of mathematical induction we can say that the function holds true for n+1 value based on the assumption made in step 2.**