Assignment 1

1. Correctness of bubble sort

Bubble sort pseudo code:

```
   BUBBLESORT(A)
1. for i = 1 to A.length - 1
2.    for j = A.length down to i + 1
3.        If A[j] < A[j - 1]
4.            exchange A[j] with A[j - 1]
```

a. We need to show that elements in every instance of A` form a permutation of the elements of A and in each iteration it moves the smallest element in the selected sub-array to its left most position, accumulating all the smaller elements sorted to the left part of the array until it reaches the end. This is how the above mentioned bubble sort code sorts the entire array.

b. Loop invariant for lines 2-4

Loop invariant: At the beginning of each iteration of the for loop, A[j] = minimum element of the subarray A[j. .n] and the subarray A[j. .n] is a permutation of the values that were in subarray A[j..n] at the time of the loop start.

Initialization: Initially, j = n and the subarray A[j. .n] contains only a single element A[n]. Since the array has only one element and the element present is considered the smallest one, the loop invariant condition holds.

Maintenance: By the loop invariant, A[j] is the smallest element in subarray A[j. .n]. At the code execution lines 3-4, swap A[j] and A[j - 1] if A[j] is less than A[j - 1], which makes A[j - 1] the smallest element in subarray A[j - 1. .n] then onwards. Since the only change to the subarray A[j - 1. .n] is the swap of elements, and the subarray A[j. .n] is a permutation of the values that were in A[j. .n] at the time of the loop start, we see that subarray A[j - 1. .n] is also a permutation of the values that were in subarray A[j − 1. .n] at the time of the loop start. Gradually decrementing j for the next iteration maintains the loop invariant condition.

Termination: The loop terminates when j gradually decreases and reaches i. From the above loop invariant condition, A[i] = minimum element in the subarray A[i. .n] and subarray A[i. .n] is a permutation of the values that were in subarray A[i. .n] at the time of the loop start.

c. Loop invariant for lines 1-4

Loop invariant: At the beginning of each iteration of the outer for loop at lines 1-4, the subarray A[1. .i - 1] consists of the (i - 1) smallest elements in array A[1. .n] in sorted order, and the subarray A[i. .n] consists of the remaining (n - i + 1) values same as in original array A[1. .n].

Initialization: Initially before the execution starts, i = 1. The subarray A[1. .i − 1] is empty, and since there is no element, we can simply say that the loop invariant condition holds.

Maintenance: For a given value of i in each iteration, from the above loop invariant condition, the subarray A[1. .i −1] consists of the i smallest values in the array A[1. .n] in sorted order. The above question (b) showed that after executing the for loop of lines 2-4, A[i] is the smallest value in subarray A[i. .n], and subarray A[1. .i] is now the i smallest values originally in array A[1. .n] in sorted order. Even after the for loop at lines 2-4 permutates the subarray A[i. .n], the subarray A[i + 1. .n] still consists of the n − i remaining elements, same as in the original array A[1. .n]. This process happens for each iteration of i and maintains the loop invariant condition.

Termination: When i = n + 1, the for loop at lines 1-4 terminates so that i − 1 = n. From the above loop invariant condition, the subarray A[1. .i − 1] is the entire array A[1. .n], and it consists of the original array A[1. .n] elements, in sorted order. Hence on the termination of the outer for loop the entire array is sorted.

d. The running time depends on the number of times the loops are being executed in the above code. For each value of i in the outer loop at lines 1-4, the inner loop at lines 2-4 executes n - i iterations, and i takes on the values 1,2,. . . n - 1. The total number of iterations in the inner loop correspondingly are as follows:

{Formula used for below summation: Sum of first n integers starting from 1 = n * (n+1)/2}

(n - 1) + (n - 2) + ... + 3 + 2 + 1 = n * (n - 1)/2
= n^2/2 - n/2.

Thus the running time of bubble sort is Θ(n^2) in all cases (best, worst and average). So, the worst-case running time of bubble sort is the same as that of insertion sort worst-case running time.

2. Counting Basic Operations and Analyzing Algorithms

Algorithm

```
1. l=0
2. k=0
3. for i=1 to n do
4.     l=l + i*3
5.     if k<n do
6.         k=4*i*i
7. return (k,l)
```

a.

n	1	2	3	4	5
Return value (k)	4	4	4	4	16
Return value (l)	3	9	18	30	45
# multiplications ("*")	3	4	5	6	9

b. Value of k = f(n) = 4 * (ceil(sqrt(ceil(n/4))))^2.

Justification:
From the above code it is evident that the algorithm executes for 1 to n times for a given value of n. And based on the condition present at line-5 it enters into the if block and updates the value of 'k'.

Number of times k is updated = Number of times the condition at line-5 is satisfied.

Condition at line-5 is (k<n). We can replace k with (4 * i * i) since it is the value being updated in it at line-6.

=> 4 * i * i < n
=> i^2 < n/4
=> i < $\sqrt{n/4}$

Since the initial value of k is 0, it will directly enter into the if block in line-5 initially. So we can modify the above condition as follows

=> i = $\sqrt{n/4}$

And this will work for all the numbers exactly divisible by 4. But for the numbers which are not divisible by 4 we get decimal values. So in order to avoid them and to get the closest possible integer we place a ceiling.

=> i = ceil($\sqrt{ceil(n/4}$) -> (Equation: 1)

Now from code line-6 k = 4*i*i. Replace i with the above value

=> k = 4 * (ceil($\sqrt{ceil(n/4}$))^2)

Let us verify the same with some examples.

(i) n = 64 (which is divisible by 4)

The expected k value is 64. From the above formula k = 4 * (ceil($\sqrt{ceil(n/4)}$))^2)

=> k = 4 * (ceil($\sqrt{ceil(n/4)}$))^2)
=> k = 4 * (ceil($\sqrt{ceil(64/4)}$))^2)
=> k = 4 * (ceil($\sqrt{ceil(16)}$))^2)
=> k = 4 * (ceil($\sqrt{16}$)^2)
=> k = 4 * (ceil(4)^2)
=> k = 4 * (4 ^ 2)
=> k = 4 * 4 * 4
=> k = 64.

From the above expected = actual. Hence proved.

(ii) n = 97 (which is not divisible by 4)

The expected k value is 100. From the above formula k = 4 * (ceil($\sqrt{ceil(n/4)}$))^2)

=> k = 4 * (ceil($\sqrt{ceil(n/4)}$))^2)
=> k = 4 * (ceil($\sqrt{ceil(97/4)}$))^2)
=> k = 4 * (ceil($\sqrt{ceil(24.25)}$))^2)
=> k = 4 * (ceil($\sqrt{25}$)^2)
=> k = 4 * (ceil(5)^2)
=> k = 4 * 5 * 5
=> k = 100.

From the above expected = actual. Hence proved.

c. Value of l = f(n) = 3/2 * (n * (n+1)).

Justification:
From the above code we can conclude that the for loop at line 3-6 executes 1 to n times for a given n. The value of l is being updated at line-4 where it is basically adding up all the values of (3*i) where (1<= i <=n).

=> l = $\sum_{i=1}^{n}(3 * i)$

=> l = 3 * $\sum_{i=1}^{n}(i)$

=> l = 3 * (n * (n+1))/2
=> l = 3/2 * (n * (n+1))

Let us verify the same with some examples.

(i) n = 75

The expected value is l = 8550. From the above formula l = 3/2 *(n * (n+1))

=> 3/2 * (75 * (76))
=> 8550.

From the above expected = actual. Hence proved.

(ii) n = 86

The expected value is l = 11223. From the above formula l = 3/2 *(n * (n+1))

=> 3/2 * (86 * (87))
=> 11223.

From the above expected = actual. Hence proved.

d. Number of Multiplications = f(n) = n + ceil(sqrt(ceil(n/4))) * 2.

Justification:
From the above code, in total we have multiplications at 3 places. One at line-4 and two at line-6.

Let Number of multiplications = mul
Let Number of times line-4 executed = x

Let Number of times line-6 executed = y

=> mul = x + (2 * y) -> (Equation: 1)

From above question(b) (Equation: 1) line-6 is executed for ceil($\sqrt{ceil(n/4}$) times.

=> y = ceil($\sqrt{ceil(n/4)}$).

From above question(c) line-4 is executed for n times.

=> x = n.

Substitute x and y values in (Equation: 1)

=> mul = n + (2* ceil($\sqrt{ceil(n/4)}$)).

Let us verify the same with some examples.

(i) n = 11

The expected value is mul = 15. From the above formula mul = n + (2* ceil($\sqrt{ceil(n/4)}$)).

=> mul = 11 + (2* ceil($\sqrt{ceil(11/4)}$))
=> mul = 11 + (2* ceil($\sqrt{3}$))
=> mul = 11 + (2 * 2)
=> mul = 15.

From the above expected = actual. Hence proved.

(ii) n = 60

The expected value is mul = 68. From the above formula mul = n + (2* ceil($\sqrt{ceil(n/4)}$)).

=> mul = 60 + (2* ceil($\sqrt{ceil(60/4)}$))
=> mul = 60 + (2* ceil($\sqrt{15}$))
=> mul = 60 + (2 * 4)
=> mul = 68.

From the above expected = actual. Hence proved.

3. Practice working with asymptotic notation.

The decreasing order of the functions by their asymptotic growth is as follows.

1) $2\sqrt{n^3}$ is order of $\sqrt{n^3}$

   => $2\sqrt{n^3}$

   => $\sqrt{n^3}$.
2) 7 * log(n!) is the order of n * log(n)

   => n * log(n) - (n * loge) + O(logn) (Using stirling's approximation for factorials)

   => n * log(n).
3) 5 * $e^{ln(n)}$ is the order of n (“*”).

   => 5 * $e^{\log_e(n)}$

   => 5 * n.
4) n/4 is the order of n (“*”).
5) 4 * $\sqrt{n}$ is the order of $\sqrt{n}$.
6) 6 * ln(n^2) is the order of log(n).
7) 3 * 2^{-2n} is the order of $1/2^n$

   => 3 / $2^{(2*n)}$

   => (3 / 4) * ($1/2^n$).

From all the above (5 * $e^{ln(n)}$)and (n/4) are asymptotically equal.

Justification:

- Comparing (1) and (2). By applying the limits:

  => $\lim_{n \to \infty}$ (2 * $n^{3/2}$) / (7 * log(n!))

  => $\lim_{n \to \infty}$ (2 * $n^{3/2}$) / (7 * n * log(n)) (Using stirling’s approximation log(n!) = n * log(n) - (n * loge) + O(log(n)) = (n*log(n)) approximately) (Applying L'Hopital's Rule on top of it)

  => $\lim_{n \to \infty}$ (3 * $n^{1/2}$) / (7 * (log(n)+1))

=> $\lim_{n \to \infty}$ (3/2 * $n^{-1/2}$) / (7/n)

=> $\lim_{n \to \infty}$ ((3 * $\sqrt{n}$) / 14)

=> ∞ (i.e. f(n) ϵω(g(n)))

From the formal definition f(n) ϵω(g(n)) for all c, n_o > 0
then f(n) > c.(g(n)) >= 0 for all n>=$n_{o.}$(Since it is satisfying strictly greater than condition that means it will also satisfy equal to case f(n) >= c.(g(n)) >= 0, i.e. f(n) ϵΩ(g(n)))

Therefore 2$\sqrt{n^3}$ > (7 * log(n!)).

- Comparing (2) and (3):

  => $\lim_{n \to \infty}$ (7 * log(n!)) /(5 * $e^{ln(n)}$)

  => $\lim_{n \to \infty}$ (7 * n * log(n)) /(5 * $e^{\log_e(n)}$) (Using stirling's approximation log(n!) = n * log(n) - (n * loge) + O(log(n)) = (n*log(n)) approximately)

  => $\lim_{n \to \infty}$ (7 * n * log(n)) /(5 * n)

  => $\lim_{n \to \infty}$ (7 * log(n)) /(5)

  => ∞ (i.e. f(n) ϵω(g(n)))

  From the formal definition f(n) ϵω(g(n)) for all c, n_o > 0
  then f(n) > c.(g(n)) >= 0 for all n>=$n_{o.}$ (Since it is satisfying strictly greater than condition that means it will also satisfy equal to case f(n) >= c.(g(n)) >= 0, i.e. f(n) ϵΩ(g(n)))

  Therefore (7 * log(n!)) > (5 * $e^{ln(n)}$).

- Comparing (3) and (4):

  => $\lim_{n \to \infty}$ (5 * $e^{ln(n)}$) /(n/4)

  => $\lim_{n \to \infty}$ (5 * $e^{\log_e(n)}$) /(n/4)

  => $\lim_{n \to \infty}$ (5*n)/(n/4)

  => 20 > 0 (i.e. f(n) ϵ Θ(g(n)))

  From the formal definition $\Theta(g(n))$ = O(g(n)) ∩ Ω(g(n))

  Therefore (5 * $e^{ln(n)}$) and (n/4) are asymptotically equal.

- Comparing (4) and (5):

  => $\lim_{n \to \infty}$ (n/4) /(4 * $\sqrt{n}$)

  => $\lim_{n \to \infty}$ ($\sqrt{n}$) /(4 * 4)

  => $\lim_{n \to \infty}$ ($\sqrt{n}$ /16)

  => ∞ (i.e. f(n) ϵω(g(n)))

  From the formal definition f(n) ϵω(g(n)) for all c, $n_o > 0$
  then f(n) > c.(g(n)) >= 0 for all n>=$n_{o.}$ (Since it is satisfying strictly greater than condition that means it will also satisfy equal to case f(n) >= c.(g(n)) >= 0, i.e. f(n) ϵΩ(g(n)))
  Therefore (n/4) > (4 * $\sqrt{n}$).

- Comparing (5) and (6). By applying the limits:

  => $\lim_{n \to \infty}$ (4 * $\sqrt{n}$) / 6 * ln(n^2)

  => $\lim_{n \to \infty}$ (4 * $\sqrt{n}$) / (6 * log(n^2)/log(e))

  => $\lim_{n \to \infty}$ (4 * $\sqrt{n}$ * log(e)) / (6 * log(n^2)) (Applying L'Hopital's

  Rule)

  => $\lim_{n \to \infty}$ ($n^{-1/2}$ * log(e) * n) / (3 * 2)

  => $\lim_{n \to \infty}$ ($\sqrt{n}$ * log(e)) / 6

  => ∞ (i.e. f(n) ϵω(g(n)))

  From the formal definition f(n) ϵω(g(n)) for all c, $n_o > 0$
  then f(n) > c.(g(n)) >= 0 for all n>=$n_{o.}$ (Since it is satisfying strictly greater than condition that means it will also satisfy equal to case f(n) >= c.(g(n)) >= 0, i.e. f(n) ϵΩ(g(n)))
  Therefore (4 * $\sqrt{n}$) > (6 * ln(n^2)).

- Comparing (6) and (7):

  => $\lim_{n \to \infty}$ (6 * ln(n^2)) / (3 * 2^{-2n})

  => $\lim_{n \to \infty}$ (6 * log(n^2)/log(e))/((3 /4^n))

  => $\lim_{n \to \infty}$ (6 * log(n^2) * 4^n)/(3 * log(e))

=> $\lim_{n \to \infty}$ (2 * log(n^2) * 4^n)/(log(e))

=> ∞ (i.e. f(n) ϵω(g(n)))

From the formal definition f(n) ϵω(g(n)) for all c, $n_o > 0$
then f(n) > c.(g(n)) >= 0 for all n>=$n_{o.}$ (Since it is satisfying strictly greater than condition that means it will also satisfy equal to case f(n) >= c.(g(n)) >= 0, i.e. f(n) ϵΩ(g(n)))
Therefore (6 * ln(n^2)) > (3 * 2^{-2n}).

4. Prove or disprove rigorously.

a. n^3- 2n^2+ 3n ϵ $\Omega(n^2)$
Proof:

f(n) = n^3- 2n^2+ 3n

g(n) = n^2

=> n^3- 2n^2+ 3n >= 0 (for all n>=0)

=> n^3- 2n^2+ 3n >= - 2n^2+3n (for all n>=0)

=> n^3- 2n^2+ 3n >= - 2n^2+3n^2(for all n>=0)

=> n^3- 2n^2+ 3n >= n^2(for all n>= 0)

From the formal definition f(n) ϵΩ(g(n)) for all c, $n_o > 0$
then f(n) >= c.(g(n)) >= 0 for all n>=$n_{o.}$
Therefore we choose n_o = 1, and c = 1.

b. 4n^2 - n - nlog(n) - log(n) ϵω(n^2)
Proof:

f(n) = 4n^2 - n - nlog(n) - log(n)

g(n) = n^2

=> 4n^2 - n - nlog(n) - log(n) >= 0 (for all n>=0)

=> 4n^2 - n - nlog(n) - log(n) > 4n^2 - n - nlog(n) - n^2(for all n>0)

=> 4n^2 - n - nlog(n) - log(n) > 3n^2 - n^2 - nlog(n) (for all n>0)

=> 4n^2 - n - nlog(n) - log(n) > 2n^2 - nlog(n) (for all n>0)

=> 4n^2 - n - nlog(n) - log(n) > 2n^2 - n^2 (for all n>0)

=> 4n^2 - n - nlog(n) - log(n) > n^2(for all n>0)

From the formal definition f(n) $\epsilon\omega$(g(n)) for all c, $n_o > 0$
then f(n) > c.(g(n)) >= 0 for all n>=$n_{o.}$
Therefore we choose $n_o = 1$, and c = 1.

c. $\sum_{i=1}^{n} i^{3/2} \epsilon$ O($n^{5/2}$)

Proof:

f(n) = $\sum_{i=1}^{n} i^{3/2}$

g(n) = O($n^{5/2}$)

=> 0 <= $1^{3/2} + 2^{3/2} + 3^{3/2}$ + ... + $n^{3/2}$ (for all n>=0)

=> $1^{3/2} + 2^{3/2} + 3^{3/2}$ + ... + $n^{3/2}$ <= $n^{3/2} + n^{3/2} + n^{3/2}$ + ... + $n^{3/2}$

=> $1^{3/2} + 2^{3/2} + 3^{3/2}$ + ... + $n^{3/2}$ <= n* $n^{3/2}$ (for all n>0)

=> $1^{3/2} + 2^{3/2} + 3^{3/2}$ + ... + $n^{3/2}$ <= $n^{5/2}$ (for all n>0).

From the formal definition f(n) ϵO(g(n)) for all c, $n_o > 0$
then 0 <= f(n) <= c.(g(n)) for all n>=$n_{o.}$
Therefore we choose $n_o = 1$, and c = 1.

5. **Design, Analyse and communicate algorithms.**

Algorithm

```
// Method to perform matrix multiplication.
1. matrixMultiply(A, B)
2.   C = {{0,0}, {0,0}}
3.     for i=0 to A.length do
4.            for j=0 to B.length do
5.                   for k=0 to C.length do
6.                          C[i][j] += A[i][k] * B[k][j]
7.   return C

// Method to calculate A to power of n using divide and conquer.
8. multiply(A, n)
9.    if(n < 1) do
10.       return {{1,0}, {0,1}}
11.    else if(n%2 == 0) do
12.       return multiply(matrixMultiply(A, A), n/2)
```

```
13.   else do
14.        return matrixMultiply(A,
multiply(matrixMultiply(A, A), (n-1)/2))
```

Textual Explanation:

```
If n is less than 1:
        return identity matrix
Else:
        If n is odd:
                A * recursive call with parameters($A^2$, (n-1)/2)
        Else:
                recursive call with parameters($A^2$, n/2)
```

(In the recursive call with parameters, the first argument is multiping A with itself which will be used for computing A^nwhile popping off from the stack and the second argument is n/2 in case of even 'n' and (n-1)/2 in case of odd 'n' which basically reduces 'n' by half to perform a Divide and Conquer).

Example:

A = {{2, 0}, {0, 2}}

Calculate A^5

In the below explanation matrixMultiply(A^i, A^j) will be written as A^{i+j}For example matrixMultiply(A, A) is A^2.

Inorder to calculate A to the power of 5. We make a function call multiply(A, 5).

- Step 1: Initially when multiply(A, 5) is called the execution enters into the condition block at lines 13-14 and again a function call (A * multiply(A^2, 2)) is called.

- Step 2: Now in this function call multiply(A^2, 2) the execution enters into the condition block at lines 11-12 and again a function call multiply(A^4, 1) is called.
- Step 3: Now in this function call multiply(A^4, 1) the execution enters into the condition block at lines 13-14 and again a function call (A * multiply(A^8, 0)) is called.
  Now at this point of time we have all these calls in the recursion stack.

  multiply(A, 5) -> multiply(A^2, 2) -> multiply(A^4, 1) -> multiply(A^8, 0)

  Now from the next steps onwards we keep track of this stack and the return value of each function.

- Step 4: Now in this function call multiply(A^8, 0) the execution enters into the condition block at lines 9-10 and returns {{1,0}, {0,1}}. From here onwards we start popping from the stack.

- Step 5:
  stack: multiply(A, 5) -> multiply(A^2, 2) -> multiply(A^4, 1)
  Return Value from previous function call: {{1,0}, {0,1}}

  Now in this function call multiply(A^4, 1) according to the Step 3 we compute A^4 * {{1,0}, {0,1}} which is {{16,0}, {0,16}} and return it.

- Step 6:
  stack: multiply(A, 5) -> multiply(A^2, 2)
  Return Value from previous function call: {{16,0}, {0,16}}

  Now in this function call multiply(A^2, 2) according to Step 2 we simply return {{16,0}, {0,16}} as the output.

- Step 7:

stack: multiply(A, 5)
Return Value from previous function call: {{16,0}, {0,16}}

Now in this function call multiply(A, 5) according to Step 1 we simply return A* {{16,0}, {0,16}} which is {{32,0},{0,32}} and return it as the final output.

This is how the above algorithm works to calculate A^n for a square matrix.

Asymptotic runtime of the algorithm:

$$\begin{array}{ll} \mathbf{T(n) = 1} & \mathbf{if\ n<1} \\ \mathbf{T(n) = T(n/2) + 1} & \mathbf{Otherwise} \end{array}$$

For example n = 4^k

Then applying log on both sides => log(n) = log(4^k)
=> log(n) = log(2^{2k})
=> log(n) = 2k
=> k = log(n)/2. (Equation: 1)

T(0) = 1
T(4^0) = T(1) = T(0) + 1 = 1 + 1 = 2
T(2) = T(1) + 1 = 2 + 1 = 3
T(4^1) = T(4) = T(2) + 1 = 4
T(8) = T(4) + 1 = 5
T(4^2) = T(16) = T(8) + 1 = 6
T(32) = T(16) + 1 = 7
T(4^3) = T(64) = T(32) + 1 = 8
T(128) = T(64) + 1 = 9
T(4^4) = T(256) = T(128) + 1 = 10
.
.
.

.
.
.
T(4^k) = (2*k) + 2 = (2 * log(n)/2) + 2 (from Equation: 1)
= **log(n) + 2**

There the asymptotic runtime of the above algorithm is **log(n)**.

Correctness of the Algorithm:

Recursion invariant condition: At each recursive call in the execution, multiply(A,n) always returns A^k where k<=n.

Initialization (Base Condition): When the divide and conquer is reached to the lowest granular level where n = 0, the multiply(A, n) returns the Identity matrix (I) = {{1,0},{0,1}}. Because in the lowest granular level n = 0 and anything to the power of 0 should be 1 which basically represents identity in case of number, similarly in case of matrices we are defining identity as the identity matrix. Hence A^0= I where I = {{1,0}, {0,1}}.

Maintenance: The main control logic of our code deals with two cases. When n is even, then the algorithm at lines 11-12 reduces n to n/2, until n approaches to 0 and once the recursion stack starts popping out the function calls, the algorithm then returns $A^{k/2} * A^{k/2}$ = $A^k$where k<=n. And when n is odd, then the algorithm at lines 13-14 reduces n to (n − 1)/2, until n approaches to 0 and once the recursion stack starts popping out the function calls, the algorithm then returns A * $A^{(k-1)/2} * A^{(k-1)/2}$ = A^k where k<=n. Thus, the recursion invariant condition is maintained.

Termination: When the last function call in the stack is popped off, multiply(A, n) gives A^n, which is the final solution to the problem. Hence proved.