1.
Solution:

1. $1/n$
2. 2^{100}
3. $\log \log n$
4. $\sqrt{}(\log n)$
5. $\log^2 n$
6. $n^{0.01}$
7. $\sqrt{}(n)$, $3\ n^{0.5}$ (since $n^{0.5} = \sqrt{}(n)$)
8. $2^{\log n}$, $5\ n$(since $2^{\log n} = n$)
9. $n \log_4 n$, $6\ n \log n$
10. $2\ n \log^2 n$
11. $4\ n^{3/2}$ ($n^{3/2} = n^{1.5}$)
12. $4^{\log n}$ ($4^{\log n} = 2^{\log n} * 2^{\log n} = n * n = n^2$)
13. $n^2 \log n$
14. n^3
15. 2^n
16. 4^n
17. 2^{2^n}

2. Solution:
Pseudocode:

Data Input: matrices A_{nm} and B_{mk}
Result: C_{nk}
For i from 1 to n:
    For j from 1 to k:
        Let $C_{ij} = 0$
        For k from 1 to m:
            Set C_{ij} += $A_{ik} \times B_{kj}$
        End
    End
End

Time Complexity Analysis:
The naive matrix multiplication algorithm contains three for loops. For each iteration of the outer loop, the total number of runs in the inner loops would be equivalent to the matrix's length. Here integer operations take O(1) time. In general, if the length of the matrix is N, the total time complexity would be $O(N * N * N) = O(N^3)$.

3. Solution:
Pseudocode:

Let STK be the stack, rootNode be the pointer to root node, SUM be the variable to

store the summation of all elements, and C is the counter to count the number of all the elements in the tree. This pseudocode would return the Average, which is the average value of the elements in the tree.

Data Input: rootNode
STK.push(rootNode)
while (STK!=null))
Node node = STK.pop()
SUM+=node.element
C=C+1
if (node.left !=null)
STK.push(node.left)
if (node.right!=null)
STK.push(node.right)
end of while
Average = SUM / C
return Average

Time Complexity Analysis:
Since stack operations are O(1), so the time of each element in the tree taking to push and pop from stack = O(1). Let n be the number of elements in the tree, so that the time complexity = Number of the elements in the tree * O(1) = O(n).

4. Solution:

Pseudocode:

Set the array as nums:

```
def nextPermutation (nums):
#initialise the inverse point as the second element from last
n=len(nums) #The length of the array
i= n-2
#Step1: Find the largest index j greater than i such that nums[i] < nums[j].
while(i>=0 and nums[i]>= nums[i+1]):
i -=1
For j in range(n-1,i,-1):
if(nums[i] < nums[j]):
#Step2: Swap the value of nums[i] with that of nums[j].
nums[j] ,nums[i] =nums[i] , nums[j]
break
#Step3: Reverse the sequence from nums[i+1] up to and including the final element nums[n].
nums[i+1:]= reversed(nums[i+1:])

return nums
```

Time Complexity Analysis:

The worst-case complexity of step 1 is O(n). At this point, the worst-case inverse point would be the first element of the array. The worst-case complexity of step 2 is O(1) as swapping occurs in constant time. The worst-case complexity of step 3 (reversing the array) is also O(n). So overall complexity is the maximum of all three that is O(n).

5. Solution:

Both of them are $\Theta(nlgn)$. If the array is sorted in increasing order, the algorithm will need to convert it to a heap that will take O(n). Afterward, there're n-1 calls to MAX-HEAPIFY and each one will perform the full lgk operations. Since:

$$\sum_{i=1}^{n-1} lgk = \lg\big((n-1)!\big) = \Theta(nlgn)$$

The same goes for decreasing order. BUILD-MAX-HEAP would be faster (by a constant factor), but the computation time will be dominated by the loop in HEAPSORT, which is $\Theta(nlgn)$.