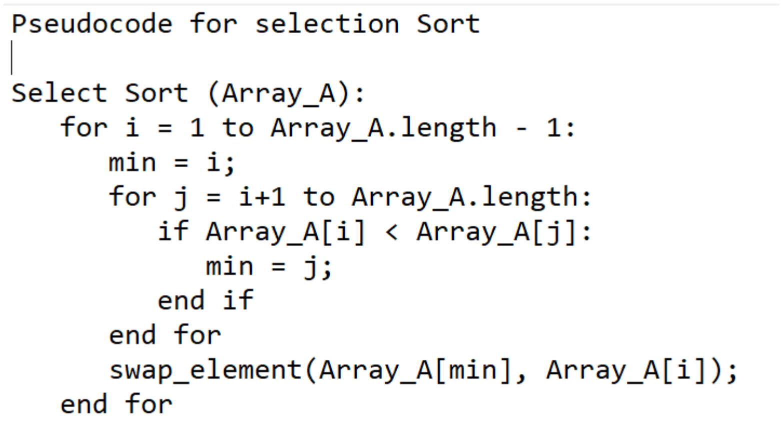
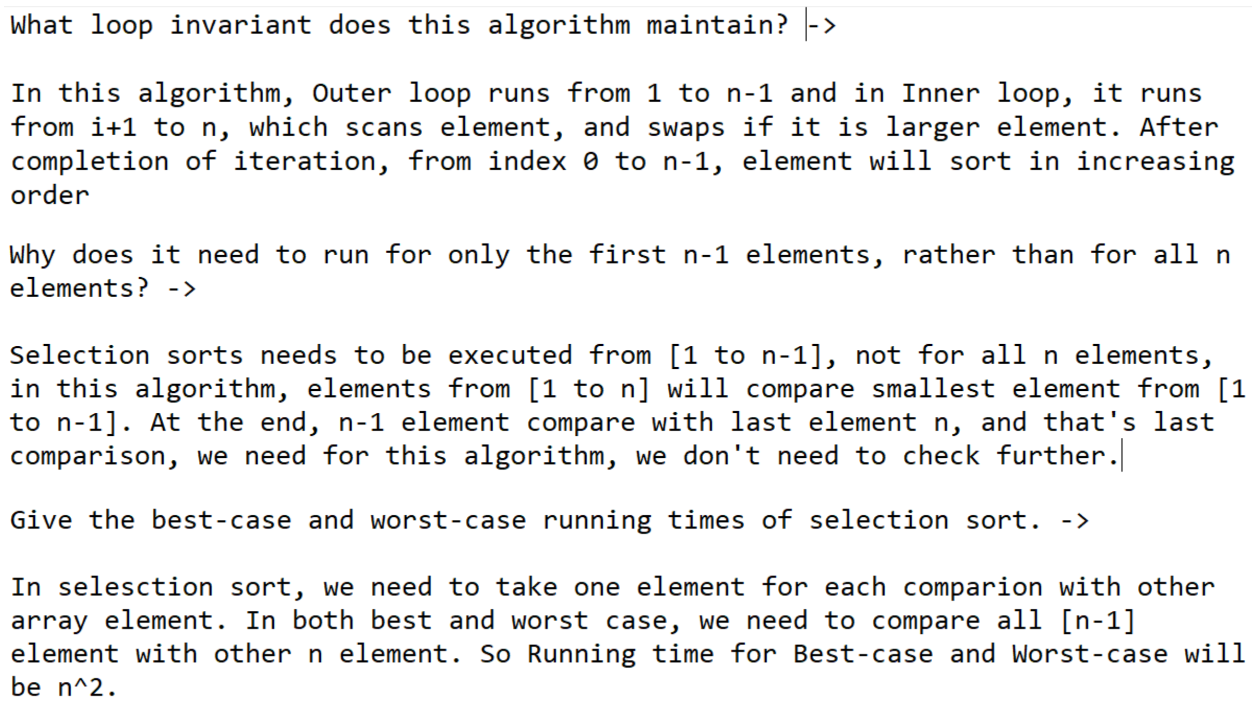
Assignment-1
Due: 11:59 pm on Friday (01/28/2022)

Please answer the questions. From 1 to 5 each question is 12 points, and from 6 to 10 each question is 8 points.

1. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A [1]. Then find the second smallest element of A and exchange it with A [2]. Continue in this manner for the first n-1 elements of A. Write pseudocode for this algorithm, which is known as *selection sort*. What loop invariant does this algorithm maintain? Why does it need to run for only the first n-1 elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort.
   Answer -



```
Pseudocode for selection Sort

Select Sort (Array_A):
   for i = 1 to Array_A.length - 1:
      min = i;
      for j = i+1 to Array_A.length:
         if Array_A[i] < Array_A[j]:
            min = j;
         end if
      end for
      swap_element(Array_A[min], Array_A[i]);
   end for
```



What loop invariant does this algorithm maintain? ->

In this algorithm, Outer loop runs from 1 to n-1 and in Inner loop, it runs from i+1 to n, which scans element, and swaps if it is larger element. After completion of iteration, from index 0 to n-1, element will sort in increasing order

Why does it need to run for only the first n-1 elements, rather than for all n elements? ->

Selection sorts needs to be executed from [1 to n-1], not for all n elements, in this algorithm, elements from [1 to n] will compare smallest element from [1 to n-1]. At the end, n-1 element compare with last element n, and that's last comparison, we need for this algorithm, we don't need to check further.

Give the best-case and worst-case running times of selection sort. ->

In selesction sort, we need to take one element for each comparion with other array element. In both best and worst case, we need to compare all [n-1] element with other n element. So Running time for Best-case and Worst-case will be n^2.

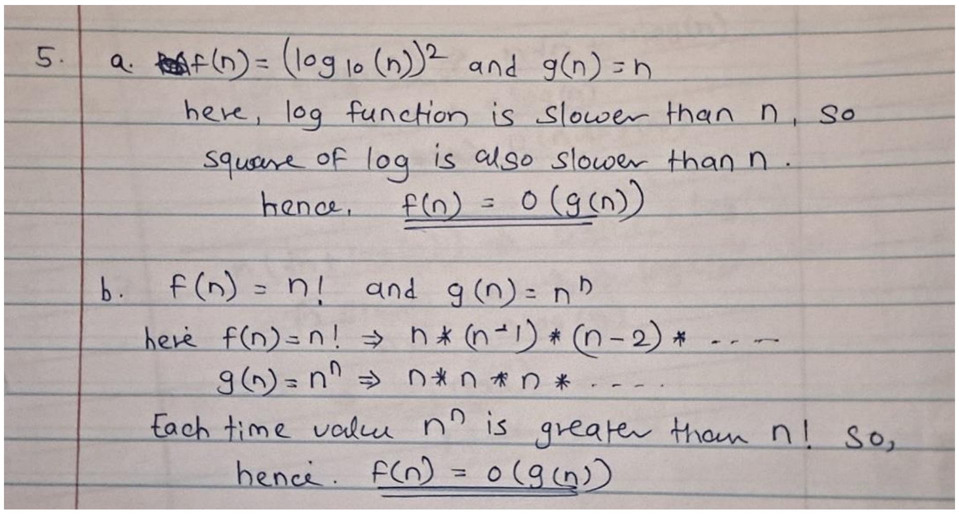
2. Although merge sort runs in O (n lg n) worst-case time and insertion sort runs in O(n^2) worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined. Show that insertion sort can sort the n/k sublists, each of length k, in O(nk) worst-case time.
   Answer -
   ```
   Show that insertion sort can sort the n/k sublists, each of length k, in O(nk)
   worst-case time. ->

   To sort k-length sublist, Insertion Sort will run for O(k^2) time. Now
   consider, to sort n element using Merge sort, we need to divide by k-lenght
   sublist, i.e. n/k sublists. And, to sort this n/k sublist using Insertion sort,
   we need O(nk) running time.
   ```

3. Considering the question 2, show how to merge the sublists in O(n lg (n/k)) worst-case time.
   Answer-
   ```
   Initially, We have to divide n-length array into k sublist, so divide n by k
   i.e. n/k lists we got in 1st iteration. Then, in 2nd iteration, Again, we have
   to divide those 2 sublist, i.e. n/2k. In 3rd, we will get n/4k lists. We have
   to do that process until, we got single element in array, which will be our
   final steps. So, we can say that, we need to divide n/k by 2 each time, until
   we got single element, so our running time O(log(n/k)) will be required for
   merging elements in array and each steps take O(n) times to execute. So our
   total running time will be O(nlog(n/k)).
   ```

4. Considering the question 2, how should we choose k in practice?
   Answer-
   ```
   K should be chosen for the maximum input size where Insertion sort is powerful
   than Merge sort. We need to calculate k for each size of sub-array and need to
   compare here Merge Sort dominates Insertion sort.
   We need to calculate k-th element using 2^(n/8) for each value of n. If 2^(n/8)
   < n, then, we can say Insertion sort dominate Merge sort, then we need to
   increment n by 1 and repeat calculation until we don't find 2^(n/8) > n, if it
   is greate than n, then we can say that Merge sort beat the Insertion sort and
   we can select than n-th number as K value for calculation.
   ```

5. For each of the following pairs of functions state whether f(n) = O(g(n)) or f(n) = Ω(g(n)) or f(n) = Θ(g(n))

   a. $f(n) = (\log_{10}(n))^2$ and $g(n) = n$.
   b. $f(n) = n!$ and $g(n) = n^n$.



5. a. $f(n) = (\log_{10}(n))^2$ and $g(n) = n$

here, log function is slower than n, so square of log is also slower than n.

hence, $f(n) = O(g(n))$

b. $f(n) = n!$ and $g(n) = n^n$

here $f(n) = n! \Rightarrow n * (n-1) * (n-2) * \dots$

$g(n) = n^n \Rightarrow n * n * n * \dots$

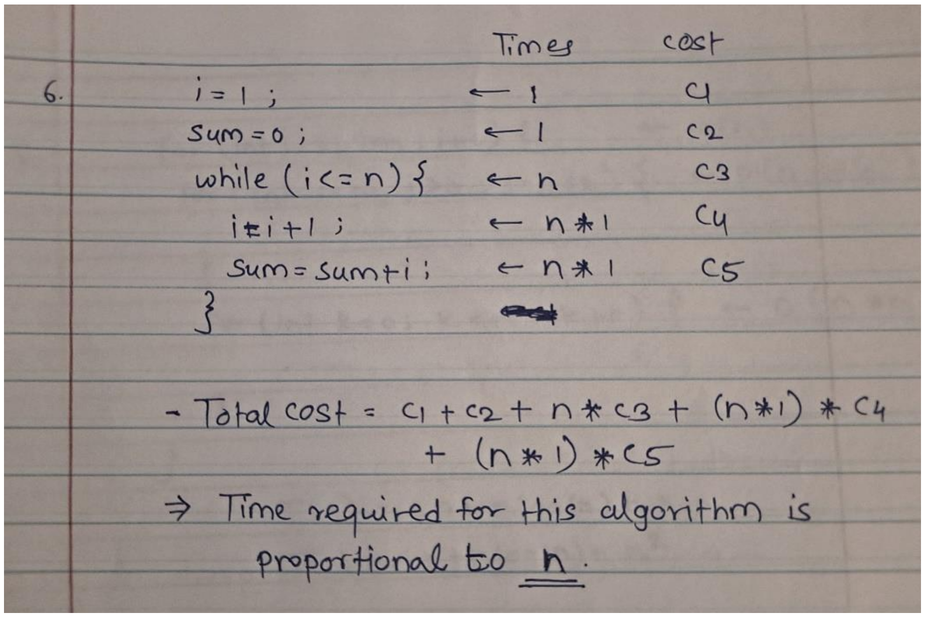
Each time value n^n is greater than $n!$ so,

hence. $f(n) = O(g(n))$

6. Find the “total cost” by finding the total number of times each statement is executed.

```
i = 1;
sum = 0;
while (i <= n) {
i = i + 1;
sum = sum + i;
}
```

Answer-



6.

	Times	cost
i = 1 ;	← 1	c1
sum = 0 ;	← 1	c2
while (i<=n) {	← n	c3
i=i+1 ;	← n*1	c4
sum=sum+i ;	← n*1	c5
}		

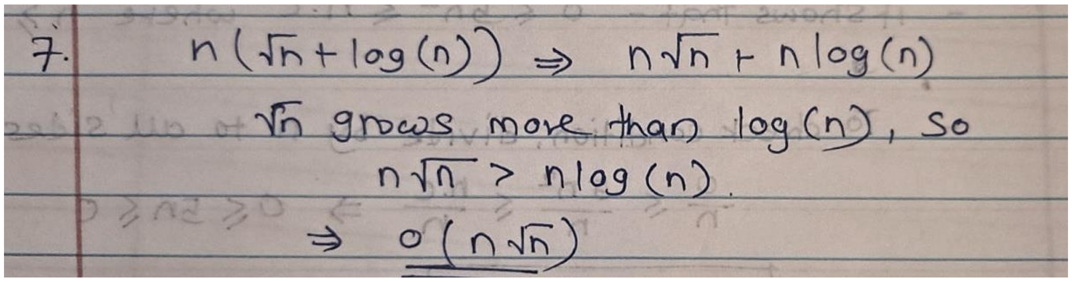
- Total cost = $c_1 + c_2 + n * c_3 + (n*1) * c_4 + (n*1) * c_5$

⇒ Time required for this algorithm is proportional to n.

7. Describe the running time using big-O notation.

   n (sqrt(n) + log n)

   Answer-



7. $n(\sqrt{n} + \log(n)) \Rightarrow n\sqrt{n} + n\log(n)$

$\sqrt{n}$ grows more than $\log(n)$, so

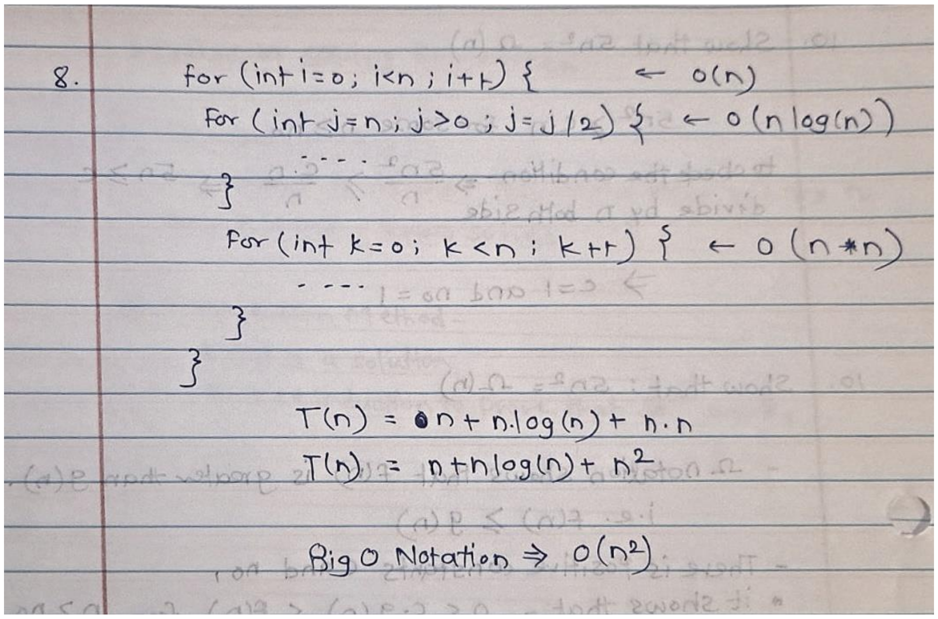
$n\sqrt{n} > n\log(n)$.

$\Rightarrow \underline{O(n\sqrt{n})}$

8. What is the big - O notation for the following lines of code

```
for (int i = 0; i < n; i++) {
for (int j = n; j > 0; j = j / 2) {
... }
for (int k = 0; k < n; k++) {
... }
}
```

Answer-



8.

```
for (int i=0; i<n ; i++) {          ← O(n)
  For (int j=n ; j>0 ; j=j/2) {     ← O(n log(n))
      ....
  }
  For (int k=0; k<n; k++) {         ← O(n*n)
      ....
  }
}
```

$T(n) = n + n.\log(n) + n.n$

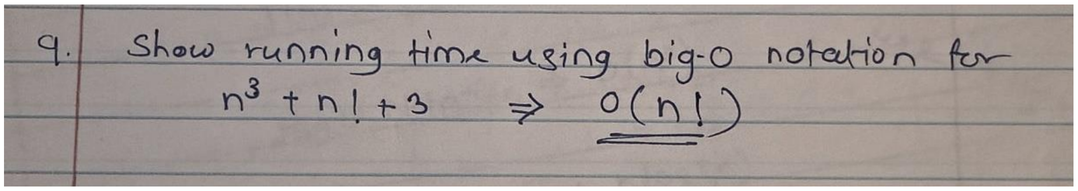
$T(n) = n + n\log(n) + n^2$

Big O Notation $\Rightarrow O(n^2)$

9. Show the running time using big- O notation for the following

   n^3 + n! + 3

   Answer -

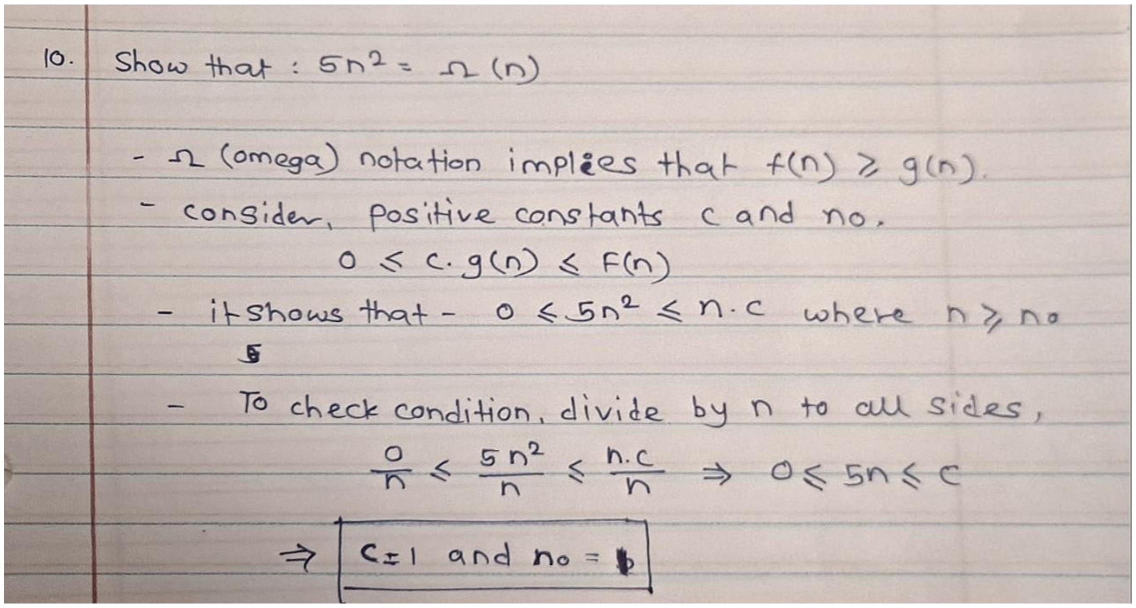


9. Show running time using big-O notation for

$n^3 + n! + 3 \quad \Rightarrow \quad \underline{\underline{O(n!)}}$

10. Show that $5n^2 = \Omega(n)$

<u>Answer-</u>



10. Show that : $5n^2 = \Omega(n)$

- Ω (omega) notation implies that $f(n) \geq g(n)$.
- consider, positive constants c and n_0.

$$0 \leq c \cdot g(n) \leq F(n)$$

- it shows that - $0 \leq 5n^2 \leq n \cdot c$ where $n \geq n_0$

5

- To check condition, divide by n to all sides,

$$\frac{0}{n} \leq \frac{5n^2}{n} \leq \frac{n \cdot c}{n} \Rightarrow 0 \leq 5n \leq c$$

$$\Rightarrow \boxed{c = 1 \text{ and } n_0 = 1}$$