Bubble sort Time complexity:

in the worst case it $o(n^2)$
because in the worst case we may need to compare and swap for each element in the list
in itration 1:
we have n-1 compares and n-1 swaps
in itration 2:
n-2 comprations and n-2 swap
⋮
until ~~itration~~ n-1:
wr have 1 compration and 1 swap
so the total ~~comp~~ is:

$(N-1)+(N-2)+\dots+1 = (N-1)\cdot(N-1+1)/2 = (N-1)\cdot N/2$

$\Rightarrow T(n) = T(n)$ * worst case comarssions and swap
so $o(n^2)$ in the worst case

in the best case it $o(n)$ and that when the list is already sorted and the algorithm can complete in a single pass without any swap

merge sort :

The time complexity of merge sort is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$T(n)$ - the time complexity of sorting a list of n element

$O(n)$ - the time complexity to merging the two sorted halves which is linear in the size of the input list.

so $N/2$ element futur divides

- $T(n) = 2 \cdot \left[2T\left(\frac{n}{4}\right) + \frac{N}{2}\right] + N = 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot N$

$$\vdots$$

$$2^k \cdot \left(T\left(\frac{N}{2^k}\right) + k \cdot N\right)$$

$\frac{N}{2^k} = 1$ $\Longrightarrow$ $k = \log_2 N$

because we divid until 1 element

- $T(N) = N * T(1) + N \log_2 N = n + n \log_2 n = n \log_2 n$

$O(n \log_2 n)$

Quick sort:

The time complexity in the best case is $o(n \log n)$ and the worst case its $o(n^2)$

1. Divide steps:

   Choose a pivot element from array then partition the array into two sub arrays
   this partitioning step takes liner time $o(n)$ n-number of elements

2. Conquer step:

   Recursively apply the quicksort algorithm to the two subarrays created in the divide step
   the Time complexity recurrence relation for quicksort is similar to that of merge sort:

   $$T(n) = T(k) + T(n-k-1) + o(n)$$

   k - number of elements in the partition with element less than pivot

when the pivot is well chosen and divides the array into two nearly equal halves the time will be $o(n \log n)$ However in the worst case when the partioning is highly imbalanced the time can become $o(n^2)$

Ex 10.1

space complexity:

the space complexity of recursive factorial is o(n) wher n is the input numbers.
this is due to the space needed for the function call stack, which grows linearly with the input size.

Ex 10.2:

- the space complexity is o(n!) and thats depends on the depth of the call stack and the call stack depends on the number of call recursive made
the recursive function is called multiple times in each recursive call adds a new frame to the call stack, storing info such as function parametes, local variables, and return addres, the max depth of call stack is equal to the length of the input string.
- The space complexity of generating permutions is o(n!) due to the factorial growth in depth.