bubble sort

```
Algorithm bubbleSort(A):
  Input: An (unsorted) array A
  Output: An sorted array A

  n = length(A)
  swapped = true
  while swapped
    swapped = false
    for i from 0 to n
      if a[i] > a[i + 1]
        swap(a[i], a[i + 1])
        swapped = true
end
```

For each element in the array, bubbleSort does $n-1$ comparisons which is $O(n)$ and there are n elements in the array so bubbleSort has a total running time of $\boldsymbol{O(n^2)}$.

merge sort

```
Algorithm merge(L, R):
  Input: Two sorted arrays L and R
  Output: An sorted array of L and R

  if L = []
    return R
  if R = []
    return L
  a = L[1], b = R[1]
  L' = L without a, R' = R without b
  if a <= b
    return [a] + merge(L', R)
  return [b] + merge(L, R')
end
```

When merge(L, R) is called, at most one recursive call is made, in which $|L| + |R|$ decreases by 1. Therefore, at most $O(n)$ recursive calls are made, where $n = |L| + |R|$ is the length of the input and since a constant number of operations are executed for each recursive call, it takes at most $O(n)$ time to run.

```
Algorithm mergeSort(X):
  Input: An (unsorted) array X
  Output: An sorted array X

  if |X| <= 1
    return X
  split X into two halves, X = L + R
  return merge(mergeSort(L), mergeSort(R))
end
```

The total lengths of lists processed at each level of recursion is constant at $|X| = n$ and the total amount of work done for each call is linear in the lengths of the arguments. The number of times X can be halved is $O(\log n)$ hence the time complexity of mergeSort is $\boldsymbol{O(n \log n)}$.

quick sort

In the algorithm, p will be our pivot.

```
Algorithm quickSort(L):
  Input: Array to be sorted L
  Output: An sorted array of L

  if length(L) <= 1
    return L
  remove first element, p, from L
  A = elements in L that are <= p
  B = elements in L that are > p
  L = quickSort(A)
  R = quickSort(B)
  return L + p + R
end
```

The worst case occurs when for each recursive call, one of A or B is empty. Let n be the size of our array L. Then n recursive calls are made, with the argument one element shorter each time. Before each recursive call, A and B must be calculated which requires $O(n)$ steps. So the total work done is $n + (n-1) + ... + 1 = \frac{1}{2}n(n+1)$. Hence quick sort is in $\boldsymbol{O(n^2)}$.

bucket sort

Suppose we wanted to sort n items whose keys are integers in the range $[0, N-1]$ for some integer $N \geq 2$. For example, we want to sort the two-digit numbers $[15, 45, 10, 30, 25, 28, 15, 50, 36]$ into ascending order of the first digit then bucket sort will return $[15, 10, 15, 25, 28, 30, 36, 45, 50]$. Some implementations will use another algorithm to sort each bucket itself.

```
Algorithm bucketSort(S):
  Input: S with keys in [0, N - 1]
  Output: S sorted in order of keys

  B array of N empty lists
  foreach x in S
    k = key of x
    remove x from S
    add x to B[k]
  for i = 1 to N
    sort(B[i])
  for i = 1 to N
    for each x in B[i]
      remove x from B[i]
      add x to end of S
end
```

The worse case for bucket sort is when all elements are allocated to the same bucket and we get $\boldsymbol{O(n^2)}$. Since individual buckets are sorted using another algorithm, if only a single bucket needs to be sorted, bucket sort will take on the complexity of the inner sorting algorithm.