Selection Sort

selection sort works by dividing the list into a sorted portion and an unsorted portion. It then sequentially scans the list, finds the smallest element (lets say for example the i^{th} element) in the unsorted portion and places it in the beginning of the unsorted array. In the next cycle, the unsorted list now begins at the $(i+1)^{th}$ element.

Code

```
# selection sort

data = []

for i in range(len(data)):
        min_idx = i

        for j in range(i+1, len(data)):

                # find the minimum of the unsorted portion
                if data[j] < data[min_idx]:
                        min_idx = j
        # swap
        temp = data[i]
        data[i] = data[min_idx]
        data[min_idx] = temp
```

Time Complexity Analysis

outer for-loop

the outer for-loop iterates through $(n-1)$ elements. This will happen in all cases of this algorithm. this carries a time complexity of $O(n)$.

Inner for-loop
the inner for-loop operates on the $(n-1)$ elements of the list at first and then decrease each time, for n elements we get:

$$(n-1)+(n-2)+(n-3)+\ldots+2+1$$

operations. From this we can see that this is the sum of (n-1) elements. We can write this explicitly with

$$\sum_{k=0}^{n-1} n = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$

from this we can see that the inner for-loop has a time complexity of $O(n^2)$

comparison
in the worst case evaluation (Big O) the worst case for selection sort would imply a back-wards sorted list. This would require the number of comparisons to equal the number of traversals as each element that is encountered next is always less than the previous one. This will result in $\frac{n^2-n}{2}$ operations as well.

we can see that the dominating term here is n^2 which gives a
time complexity of $O(n^2)$ in the worst case.

Big-O
the dominating term (from the inner for-loop) is n^2 so selection sort has a time complexity of

$$O(n^2)$$

Insertion Sort

insertion sort works by selecting a "key" element at each iteration and comparing it to the unsorted portion of the array. The first element is

presumed sorted in the beginning. The second element is selected as the "key" in the first iteration and its compared to the first element, then put in its position. In the next iteration, the range of the sorted portion of the list extends by 1 and the key is the first element in the sorted list.

Code

```
# Insertion Sort

data = []

for i in range(1, len(data)):

        key = data[i]
        comp_idx = i-1

        while key < data[comp_idx] and comp_idx > 0
                data[comp_idx+1] = data[comp_idx]
                comp_idx -= 1

        # putting the data int the right spot
        data[comp_idx+1] = key
```

Time Complexity

Inner while-loop

in the worst case, a reverse ordered list, the inner while loop will perform $1+2+3+\ldots+(n-1)$ operations so we get $\frac{n(n+1)}{2}$ operations at worst. This gives a time complexity of $O(n^2)$

Outer for-loop

the outer for-loop will always run from 1 to $(n-1)$ so well have it iterate over n elements. This gives a time complexity of $O(n)$

Big-O

the dominating term here is found that inner while loop that traverses the list and finds the correct spot for the key in the sorted portion. This is also the dominating factor in the algorithm so we get an overall time-complexity of

$$O(n^2)$$

Bubble Sort

bubble sort works by iterating over all elements and swapping them, "bubbling" the elements to the top. The sorted portion accumulates a t the end of the list and moves towards the beginning.

Code

```
# bubble sort

data = []

for i in range(len(data)):
        for j in range(0, len(data)-i-1):
                if data[j] < data[i]:
                        # swap
                        temp = data[i]
                        data[i] = data[j]
                        data[j] = temp
```

Time Complexity

outer for-loop

the outer for-loop will execute n times always. So this has a time complexity of $O(n)$

inner for-loop

the inner for-loop will execute the same amount of times as selection sort

$$\sum_{k=0}^{n-1} n = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$

this gives us a time complexity of $o(n^2)$.

swap

in the worst case, we will perform as many swaps as traversals, so we will also have $\frac{n^2-n}{2}$ swaps at worst. The time complexity is $o(n^2)$

Big O

given that the dominating term of our algorithm is n^2 we will have a time complexity of

$$O(n^2)$$