Sort Algorithms

Let's look at visualization of comparison between different sorting techniques:
https://www.youtube.com/watch?v=ZZuD6iUe3Pc
Selection, Shell, Insertion, Merge, Quick, Heap, Bubble, Comb, and Cocktail sorts.
Or https://www.youtube.com/watch?v=jrHLeKwMzfI
As you see, some are faster than the others are.
I general different sorting algorithm can be divided into different categories:

1) Transposition Sorting
    a. Bubble Sort
2) Insert and Keep Sorting
    a. Insertion Sort
    b. Tree Sort
3) Priority Queue Sorting
    a. Selection Sort
    b. Heap Sort
4) Divide and Conquer Sorting
    a. Quick Sort
    b. Merge Sort
5) Diminishing Increment Sorting
    a. Shell Sort
6) Address Calculation Sorting
    a. Proxmap Sort
    b. Radix Sort

Don't worry we are not planning to cover all of them. This is just a hint to show that there are many possibilities to choose from.
Most of you are familiar with selection, insertion, and bubble sort. However, here let us review selection sort and insertion sort.

Insertion sort: $O(N^2)$

The algorithm divides the input list into two parts: the sublist of items already sorted, and rest as unsorted. Assume first one is sorted sublist then find smallest in the rest of the list, then next smallest Sorted list will grow and unsorted will shrink until nothing left.

```
64 25 12 22 11 # this is the initial, starting state of the array
11 25 12 22 64 # sorted sublist = {11}
11 12 25 22 64 # sorted sublist = {11, 12}
11 12 22 25 64 # sorted sublist = {11, 12, 22}
11 12 22 25 64 # sorted sublist = {11, 12, 22, 25}
11 12 22 25 64 # sorted sublist = {11, 12, 22, 25, 64}
```

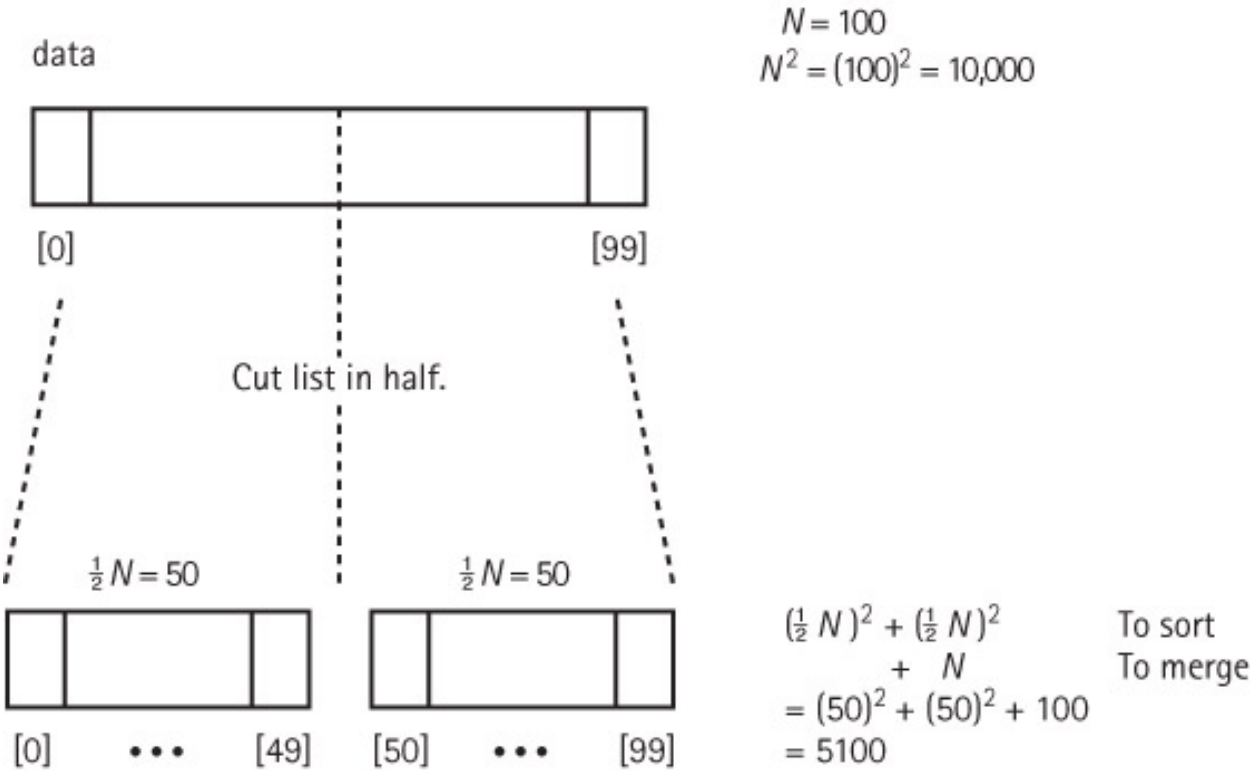
```
#ArrayLst -> ArrayLst
#sort using insertion sort
def insertion_sort(lst):
   lst_size = lst.size
   swapped = 1
   i = 0
   while i < lst_size-1 and swapped == 1:
      swapped = 0
      j=0
      while j < lst_size-i-1:
         if lst.value[j] > lst.value[j+1]:
            temp = lst.value[j]
            lst.value[j] = lst.value[j+1]
            lst.value[j+1] = temp
            swapped = 1
         j += 1
      i += 1
   return lst
```

Selection sort: O(N^2)

```
#ArrayLst -> ArrayLst
#sort using selection sort
def selection_sort(lst):
        lst_size = lst.size
        i = 0
        while i < lst_size-1:
                maxIndex = 0
                j = 1
                while j < lst_size-1-i:
                        if lst.value[j] > lst.value[maxIndex]:
                                maxIndex = j
                        j += 1
                temp = lst.value[lst_size-1-i]
                lst.value[lst_size-1-i] = lst.value[maxIndex]
                lst.value[maxIndex] = temp
                i += 1
```

Merge Sort: O (n log n)

- Use divide-and-conquer for sorting
- It is a fine example of a recursive algorithm.
- Cut the array into two pieces,
  - sort each segment, and
  - then merge the two back together
- Merge sort runs in *O(N logN) worst-case running* time, and
- The number of comparisons used is nearly optimal.



Method mergeSort(first, last)
Definition: Sorts the array elements in ascending order.
Size: last - first + 1
Base Case: If size less than 2, do nothing.
General Case: Cut the array in half.
mergeSort the left half.
mergeSort the right half.
Merge the sorted halves into one sorted array.

Merge Sort Analysis

- Divide the array in half,
  - over and over again until we reach subarrays of size 1, is O(*N*).
- Perform merging at each "level" of merging:
  - Takes O(*N*) total steps
- The number of levels of merging is equal to the number of times we can split the original array in half
  - If the original array is size *N*, we have $\log_2 N$ levels.
- Because we have $\log_2 N$ levels, and we require O(*N*) steps at each level,
  - the total cost of the merge operation is: O($N \log_2 N$).
- Because the splitting phase was only O(*N*), we conclude that Merge Sort algorithm is **O($N \log_2 N$).**

Disadvantage of Merge Sort:

- It requires an auxiliary array that is as large as the original array to be sorted.
- If the array is large and space is a critical factor, this sort may not be an appropriate choice.
- Therefore, Quick sort algorithm with **O(N $\log_2$N)** introduce that move elements around in the original array and do not need an auxiliary array.

Quick Sort: O (n log n)

- A divide-and-conquer algorithm ➔ Inherently recursive
- At each stage the part of the array being sorted is divided into two "piles", with everything in the left pile less than everything in the right pile
- The same approach is used to sort each of the smaller piles (a smaller case).
- This process goes on until the small piles do not need to be further divided (the base case).

Method quickSort (first, last)

Definition: Sorts the elements in sub array values[first]..values[last].

Size:	last - first + 1
Base Case:	If size less than 2, do nothing.
General Case:	Split the array according to splitting value. quickSort the elements <= splitting value. quickSort the elements > splitting value.

How do we select splitVal?

- One simple solution is to use the value in values[first] as the splitting value
- Or pick the middle element as splitting value
- Or compare first and last and swap if first>last and pick first as splitting value

Quick Sort Analysis

- On the first call,
  - Every element in the array is compared to the dividing value (the "split value"), so the work done is O(N).
- The array is divided into two sub arrays (not necessarily halves)
  - Each of these pieces is then divided in two, and so on.
  - If each piece is split approximately in half, there are O($\log_2 N$) levels of splits.
  - At each level, we make O(N) comparisons.
- ➔ Quick Sort is an O($N \log_2 N$) algorithm.