Measuring the runtime complexity of the algorithms using Big O notation

---

In this NoteBook you will find different algorithm implementation and their test result on plots.

Results shown on plots present `Big O notation` of each algorithms

Because some plots might look similar (but have different runtime-complexity) the plots are numbered.

You can also check test cases in `Required-Functions` Before implementation of algorithms

- *For example `Bubble Sort` goes from `1000` to `15000` with 1000 step in each iteration*

List of sorting algorithms included in the tests:

> Bubble Sort

> Insertion sort

> Merge sort

> Quick sort

> Linear Search

> Binary Search

Table of Content:

Python

In the cells below we get the environment ready and prepare needed functions

In [12]:
```
from pprint import pprint
import rx7 as rx
import matplotlib.pyplot as plt
```

In [13]:
```
# return an array[int] with length of n (between [first,last])
def make_random_list(n:int,first=0,last=0):
    if not last:  last=n
    a = []
    for i in range(n):
        a.append(rx.random.integer(first,last))
    return a

# Prepare ranges for different algorithms
```

```python
def Range(a,b): return range(a,b+1)
Groups = {
    "Bubble"    :   list(Range(1_000,15_000))[::1000],
    "Insertion" :   list(Range(1_000,20_000))[::1000],
    "Merge"     :   list(Range(50_000,1_000_000))[::50_000],
    "Quick"     :   list(Range(150_000,2_400_000))[::150_000],
    "Linear"    :   list(Range(250_000,5_000_000))[::500_000],
    "Binary"    :   list(Range(400_000,4_000_000))[::400_000]
}

# shows plot of given sort_name
def generate_plot(sort_name,TIMES):
    # plt.figure(figsize=(11,8),dpi=50)
    plt.plot(Groups[sort_name],list(i for i in TIMES))
    plt.title(f"{sort_name} Sort")
    plt.xlabel(f"Length")
    plt.ylabel("Time (s)")
    # plt.xticks(Groups[sort_name],[int(i/devide_by[0]) for i in Groups[sort_name]]
    plt.show()
```

Algorithms in python:

In [14]:

```python
def bubble_sort(array):  # O(n^2)
    n = len(array)
    for i in range(n):
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]
    return array

def insertion_sort(array):  # O(n^2)
    for i in range(1, len(array)):
        key_item = array[i]
        j = i - 1
        while j >= 0 and array[j] > key_item:
          array[j + 1] = array[j]
          j -= 1
        array[j + 1] = key_item
        # array.insert(j+1,key_item)
    return array

def merge(left, right):
    if len(left) == 0:
        return right
    if len(right) == 0:
        return left
    result = []
    index_left = index_right = 0
    while len(result) < len(left) + len(right):
        if left[index_left] <= right[index_right]:
            result.append(left[index_left])
            index_left += 1
        else:
            result.append(right[index_right])
```

```
            index_right += 1
        if index_right == len(right):
            result += left[index_left:]
            break
        if index_left == len(left):
            result += right[index_right:]
            break
    return result
def merge_sort(array):  # O(n.lg(n))
    if len(array) < 2:
        return array
    midpoint = len(array) // 2
    return merge(
        left=merge_sort(array[:midpoint]),
        right=merge_sort(array[midpoint:]))

def quicksort(array):   # O(n.lg(n))
    if len(array) <= 1:
        return array
    low, same, high = [], [], []
    pivot = array[0]
    for item in array:
        if item < pivot:     low.append(item)
        elif item == pivot:  same.append(item)
        elif item > pivot:   high.append(item)
    return quicksort(low) + same + quicksort(high)


def LinearSearch(lst, element):  # O(n)
    for i in range (len(lst)):
        if lst[i] == element:
            return i
    return -1

def BinarySearch(lst, val):  # O(log(n))
    first = 0
    last = len(lst)-1
    index = -1
    while (first <= last) and (index == -1):
        mid = (first+last)//2
        if lst[mid] == val:
            index = mid
        else:
            if val<lst[mid]:
                last = mid -1
            else:
                first = mid +1
    return index
```

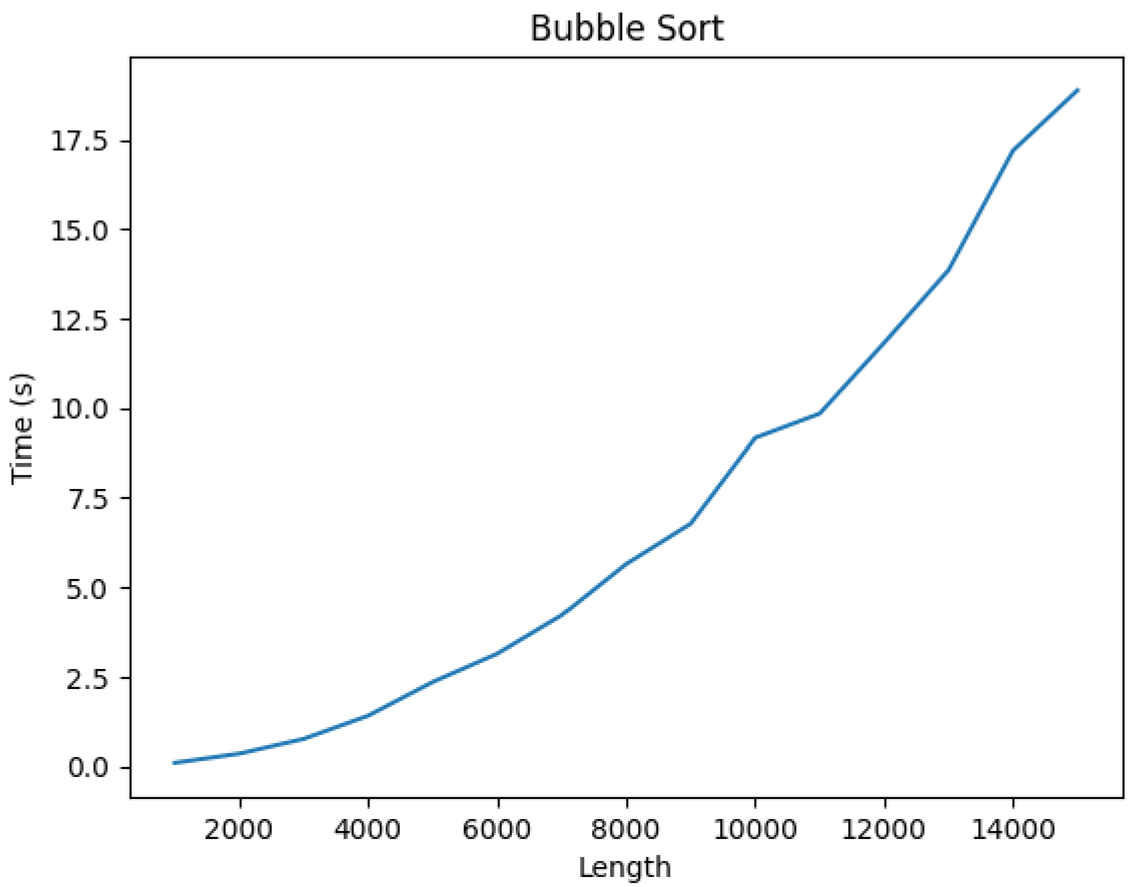
Bubble Sort (*O(n^2)*)

In [4]:

```
TIMES = []
```

```python
t = rx.record()
for n in Groups["Bubble"]:
    bubble_sort(make_random_list(n))
    TIMES.append(t.last_lap())
lap = t.lap()
```

In [5]:
```python
generate_plot("Bubble",TIMES)
print(lap)
```




```
105.512619972229
```

In [6]:
```python
pprint(dict(zip(Groups["Bubble"],TIMES)))
```

```
{1000: 0.090032815933228,
 2000: 0.34674596786499,
 3000: 0.7595036029815669,
 4000: 1.407308340072632,
 5000: 2.348259449005127,
 6000: 3.1385657787323,
 7000: 4.217178106307983,
 8000: 5.652516365051269 5,
 9000: 6.779584884643555,
 10000: 9.168732166290283,
 11000: 9.843254804611206,
 12000: 11.821351766586304,
 13000: 13.856062412261963,
 14000: 17.199060440063477,
 15000: 18.88446307182312}
```

Insertion Sort (*O(n^2)*)

In [15]:
```
TIMES = []
t = rx.record()
for n in Groups["Insertion"]:
    insertion_sort(make_random_list(n))
    TIMES.append(t.last_lap())
lap = t.lap()
```

In [16]:
```
generate_plot("Insertion",TIMES)
print(lap)
```


```
116.745201587677
```

In [17]:
```
pprint(dict(zip(Groups["Insertion"],TIMES)))
```

```
{1000: 0.041163682937622,
 2000: 0.15600085258483898,
 3000: 0.346583366394043,
 4000: 0.6440470218658451,
 5000: 0.9770770072937007,
 6000: 1.419577836990356,
 7000: 1.992505788803101,
 8000: 2.4711520671844482,
 9000: 3.1194286346435547,
 10000: 3.988490104675293,
 11000: 4.809168100357056,
 12000: 5.746230363845825,
 13000: 7.249871015548706,
 14000: 7.98595929145813,
 15000: 9.416332721710205,
 16000: 10.497888326644897,
 17000: 11.946320295333862,
 18000: 12.626477718353271,
 19000: 14.335059404373169,
 20000: 16.975867986679077}
```

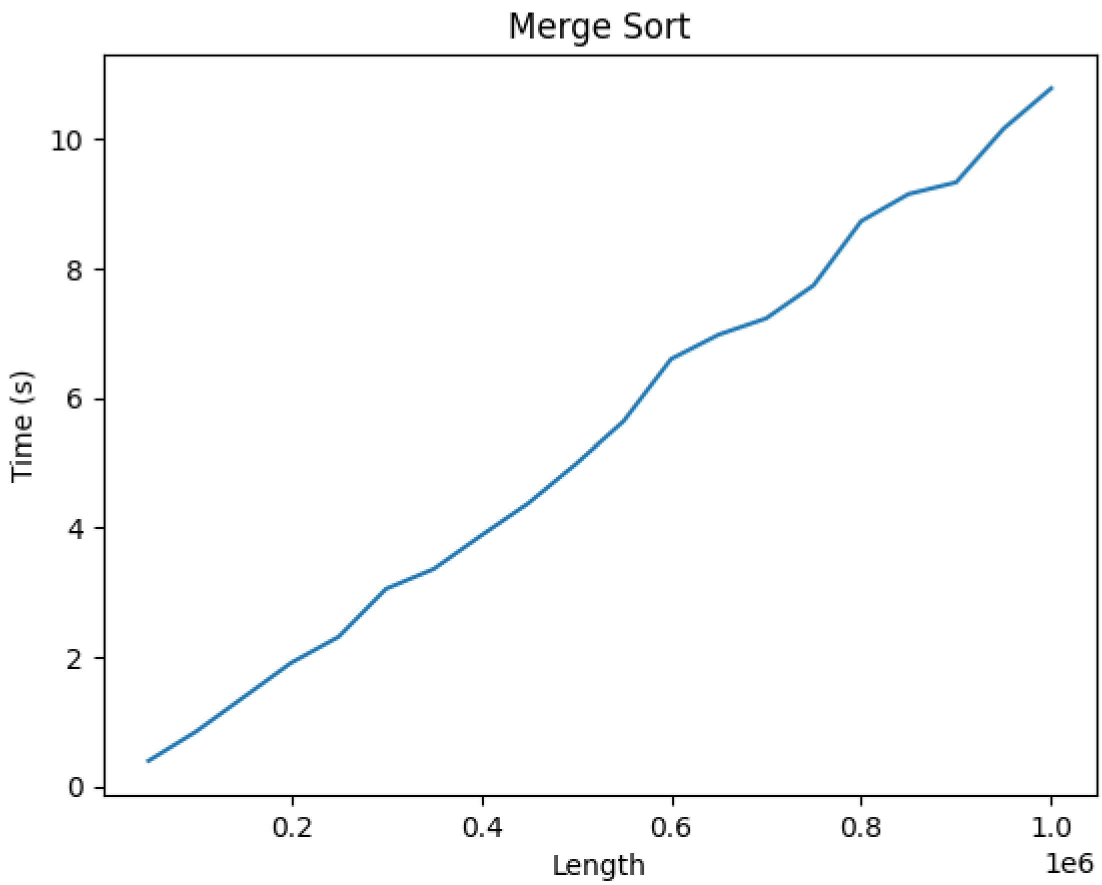
Merge Sort (*O(n.lg(n))*)

In [18]:

```
TIMES = []
t = rx.record()
for n in Groups["Merge"]:
    merge_sort(make_random_list(n))
    TIMES.append(t.last_lap())
lap = t.lap()
```

In [19]:

```
generate_plot("Merge",TIMES)
print(lap)
```




```
108.8369390964508
```

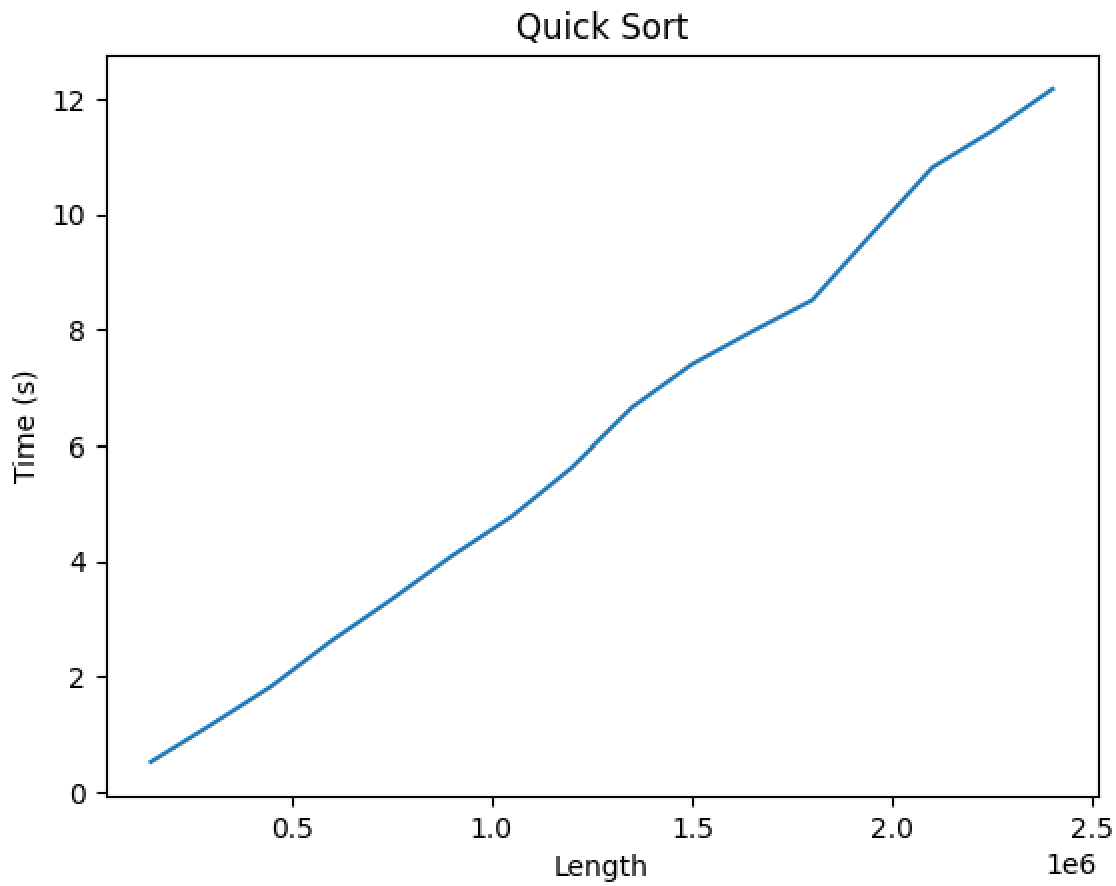
In [20]:
```python
pprint(dict(zip(Groups["Merge"],TIMES)))
```

```
{50000: 0.394999980926514,
 100000: 0.848061800003051,
 150000: 1.3780486583709721,
 200000: 1.9095399379730225,
 250000: 2.3120198249816895,
 300000: 3.0545804500579834,
 350000: 3.3596394062042236,
 400000: 3.8773605823516846,
 450000: 4.380809307098389,
 500000: 4.984855651855469,
 550000: 5.639374017715454,
 600000: 6.59976053237915,
 650000: 6.975269317626953,
 700000: 7.227179527282715,
 750000: 7.738732099533081,
 800000: 8.730546712875366,
 850000: 9.150113821029663,
 900000: 9.33069396018982,
 950000: 10.16140866279602,
 1000000: 10.783944845199585}
```

Quick Sort (*O(n.lg(n))*)

In [24]:
```
TIMES = []
t = rx.record()
for n in Groups["Quick"]:
    quicksort(make_random_list(n))
    TIMES.append(t.last_lap())
lap = t.lap()
```

In [25]:
```
generate_plot("Quick",TIMES)
print(lap)
```




```
98.58925938606262
```

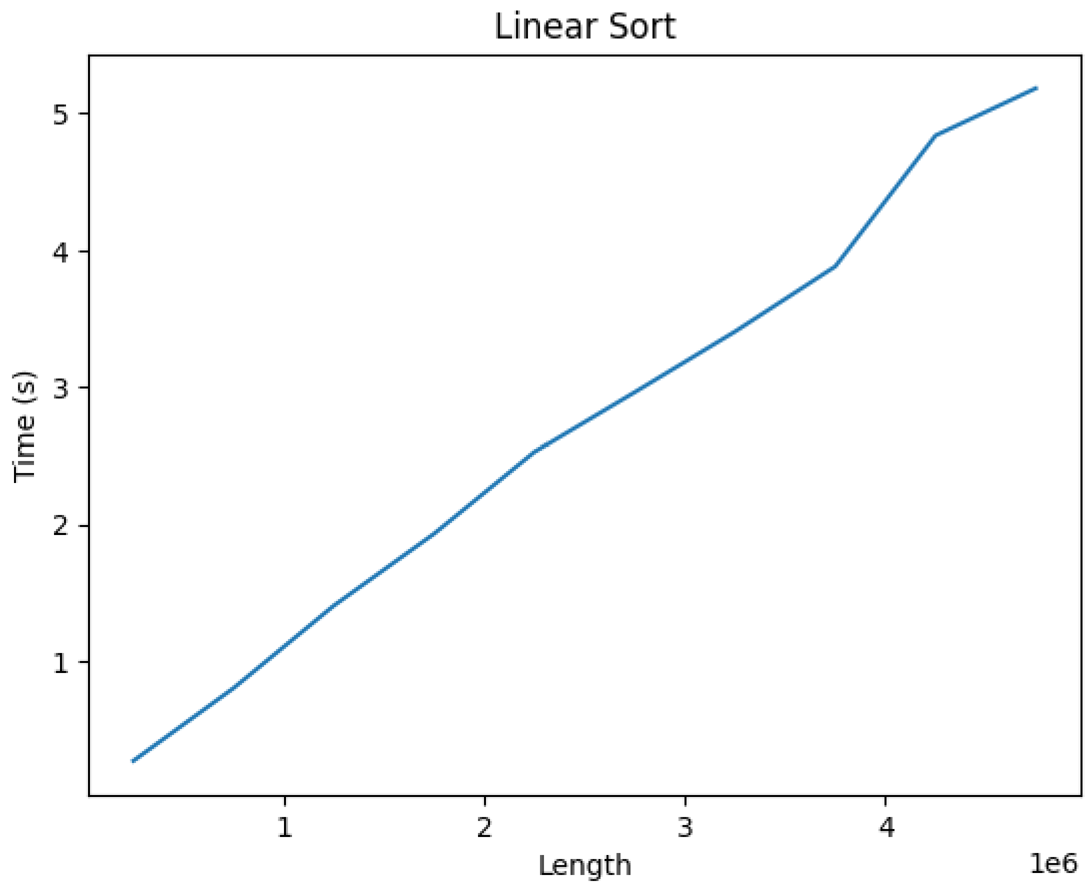
In [26]:
```
pprint(dict(zip(Groups["Quick"],TIMES)))
```

```
{150000: 0.520304441452026,
 300000: 1.162804603576661,
 450000: 1.8285608291625972,
 600000: 2.617499351501465,
 750000: 3.3348910808563232,
 900000: 4.087334632873535,
 1050000: 4.774444341659546,
 1200000: 5.621495246887207,
 1350000: 6.650914430618286,
 1500000: 7.399137496948242,
 1650000: 7.9683427810668945,
 1800000: 8.511287927627563,
 1950000: 9.672201871871948,
 2100000: 10.814805507659912,
 2250000: 11.449729681015015,
 2400000: 12.1755051612854}
```

Linear Search

In [27]:
```
TIMES = []
t = rx.record()
for n in Groups["Linear"]:
    LinearSearch(make_random_list(n),-1)
    TIMES.append(t.last_lap())
lap = t.lap()
```

In [28]:
```
generate_plot("Linear",TIMES)
print(lap)
```




```
27.219929933547974
```

Binary Search

Binary Search algorithm is so fast that needs at least `1B` long list.

(Takes around 0.002 seconds for 250,000,000 long list)

This is not possible considering the limited RAM a common computer have

Thats why it is not used in the plots but the code for algorithm is available in algorithms section

---

Using C++ for the same algorithms

Codes below run through terminal to get result from cpp code.

Results will be saved in `database.json` and in python cells, it will be interpreted to show plots

C++ Code

▶ **Click here to see C++ Algorithms**

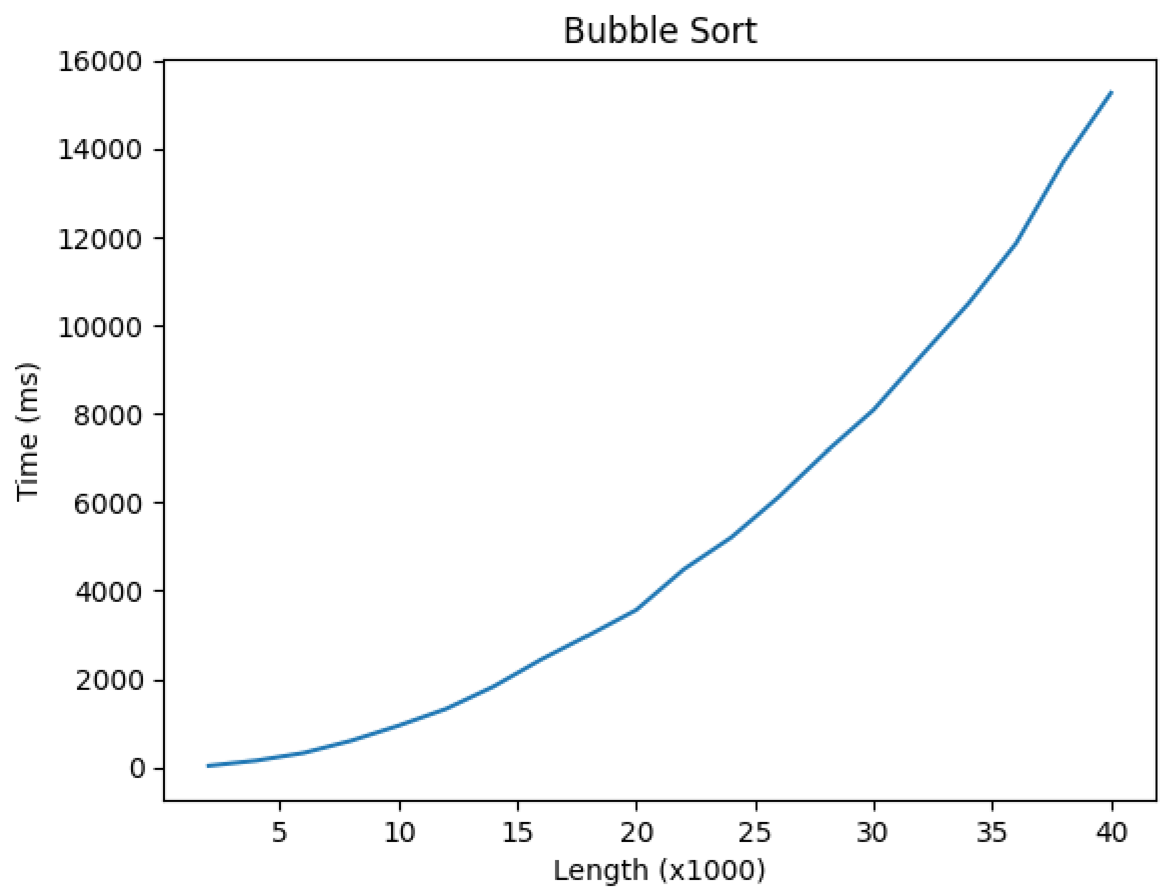
In [29]:
```
rx.terminal.run("Data_Structure.exe")
```

In [2]:
```
from collections import OrderedDict
import json

"""
    Bubbles = list(Range(2_000,40_000))[::2000],
    Insertions = list(Range(4_000,60_000))[::4000],
    Merges = list(Range(250_000,4_000_000))[::250_000],
    Quicks = list(Range(750_000,12_000_000))[::750_000],
    Linears = list(Range(1_000_000,20_000_000))[::1_000_000],
    Linears = list(Range(1_000_000,20_000_000))[::1_000_000],
"""

# Loading Database
with open("database.json") as f:
    loaded = json.load(f)
database = {}
for sort in list(loaded.keys())[:-1]:
    database[sort] = {}
    for k,v in loaded[sort].items():
        database[sort][int(k)] = v
for key,value in database.items():
    database[key] = OrderedDict(sorted(database[key].items()))
database["conclusion"] = loaded["conclusion"]

# function to generate plot for c++ database
def generate_plot(sort_name):
    db = database[sort_name]
    keys   =  [int(i)/1000 for i in db.keys()]
    values =  [i/1000 for i in db.values()]
    plt.plot(keys,values)
    plt.title(f"{sort_name} Sort")
    plt.xlabel("Length (x1000)")
    plt.ylabel("Time (ms)")
    # plt.xticks(keys)
    plt.show()
```

Algorithms Results in C++

Bubble Sort (*O(n^2)*)

In [31]:
```
generate_plot("Bubble")
```

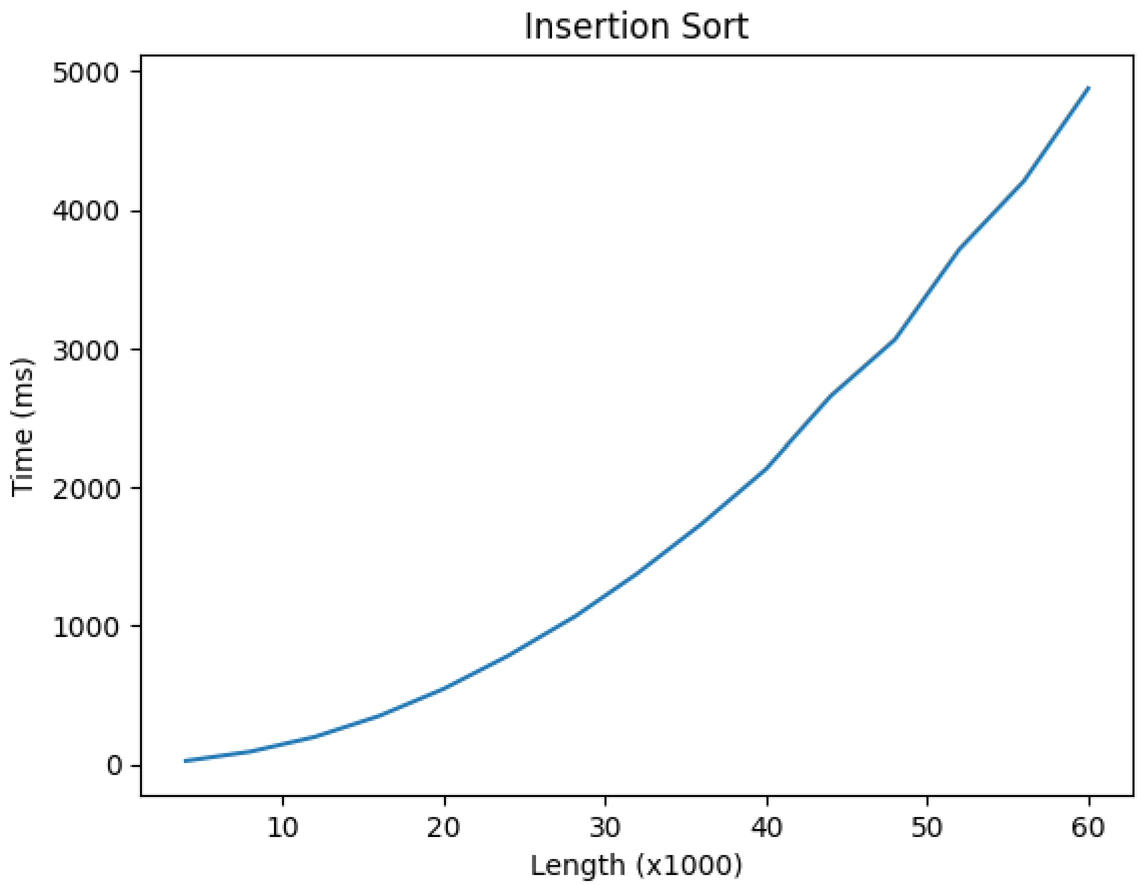


In [32]:
```
pprint(dict(database["Bubble"]))
```

```
{2000: 38278,
 4000: 155455,
 6000: 328520,
 8000: 604723,
 10000: 946157,
 12000: 1325007,
 14000: 1830161,
 16000: 2440564,
 18000: 2992593,
 20000: 3565520,
 22000: 4484009,
 24000: 5206022,
 26000: 6122150,
 28000: 7138679,
 30000: 8084628,
 32000: 9301274,
 34000: 10491659,
 36000: 11850219,
 38000: 13704406,
 40000: 15256222}
```

Insertion Sort (*O(n^2)*)

In [33]:
```
generate_plot("Insertion")
```

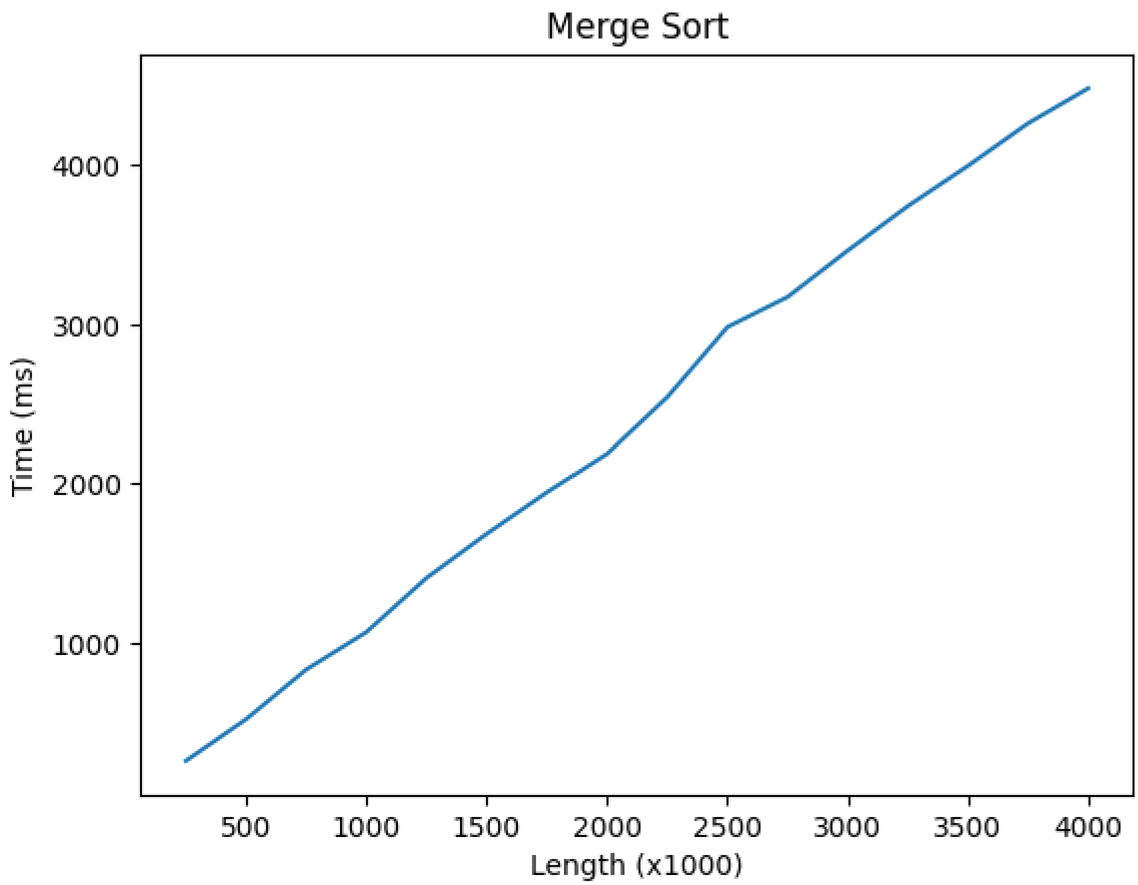


In [34]:
```
pprint(dict(database["Insertion"]))
```

```
{4000: 25430,
 8000: 91735,
 12000: 198155,
 16000: 349476,
 20000: 544733,
 24000: 783321,
 28000: 1057185,
 32000: 1378178,
 36000: 1737103,
 40000: 2132199,
 44000: 2655276,
 48000: 3064086,
 52000: 3716002,
 56000: 4207716,
 60000: 4877649}
```

Merge Sort (*O(n.lg(n))*)

In [35]:
```
generate_plot("Merge")
```

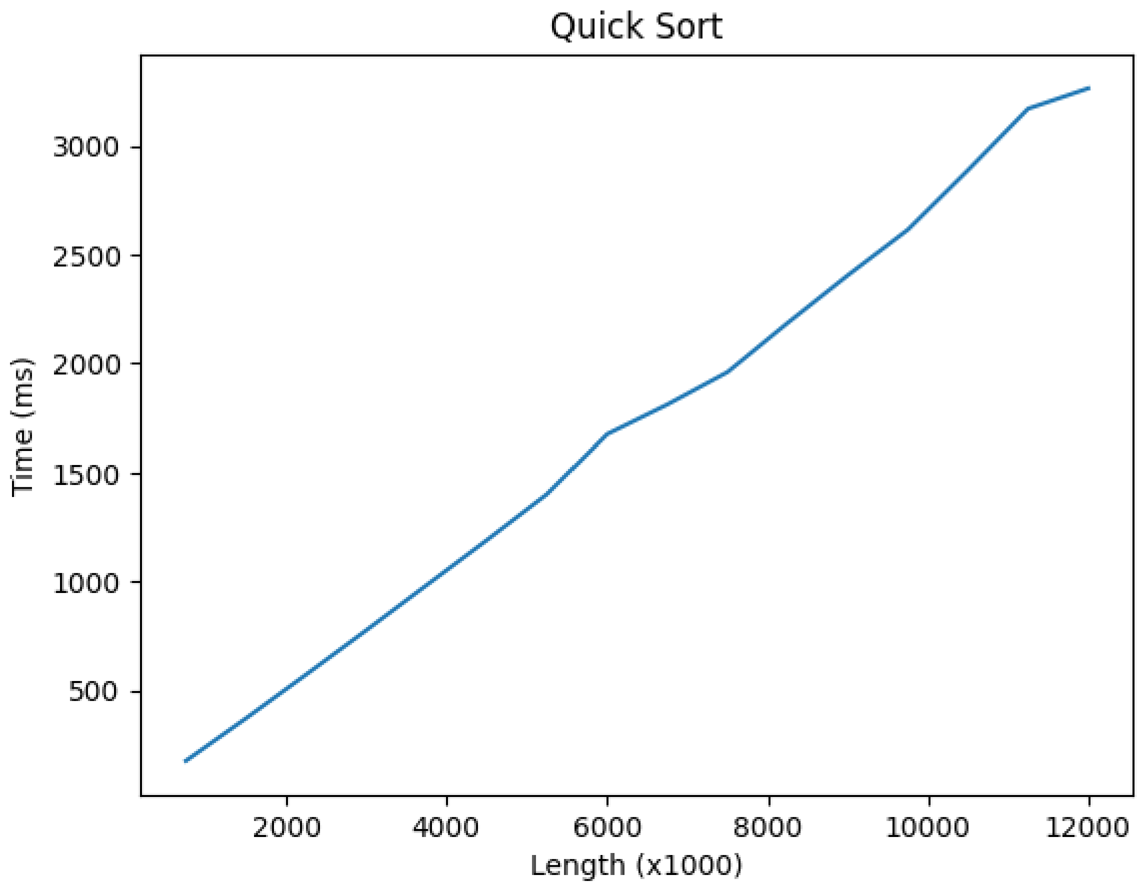


In [36]:
```
pprint(dict(database["Merge"]))
```

```
{250000: 258072,
 500000: 519523,
 750000: 831805,
 1000000: 1067436,
 1250000: 1407902,
 1500000: 1685029,
 1750000: 1945815,
 2000000: 2188528,
 2250000: 2543748,
 2500000: 2983901,
 2750000: 3173100,
 3000000: 3464910,
 3250000: 3744333,
 3500000: 3997770,
 3750000: 4265385,
 4000000: 4485192}
```

Quick Sort (*O(n.lg(n))*)

In [37]:
```
generate_plot("Quick")
```

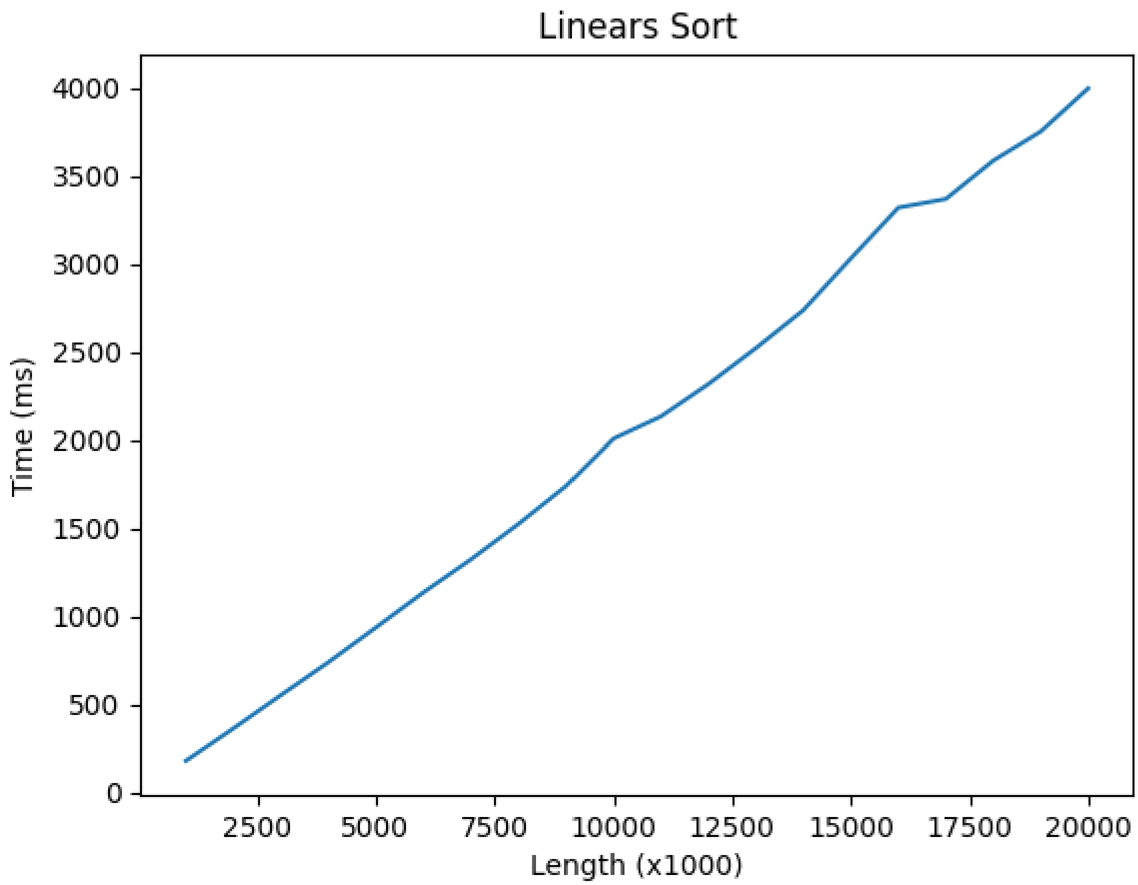


In [38]:
```
pprint(dict(database["Quick"]))
```

```
{750000: 176254,
 1500000: 371645,
 2250000: 572601,
 3000000: 775936,
 3750000: 982448,
 4500000: 1188824,
 5250000: 1399847,
 6000000: 1675284,
 6750000: 1810986,
 7500000: 1959595,
 8250000: 2185320,
 9000000: 2403979,
 9750000: 2613671,
 10500000: 2887143,
 11250000: 3169348,
 12000000: 3262918}
```

Linear Search (*O(n)*)

In [39]:
```
generate_plot("Linears")
```

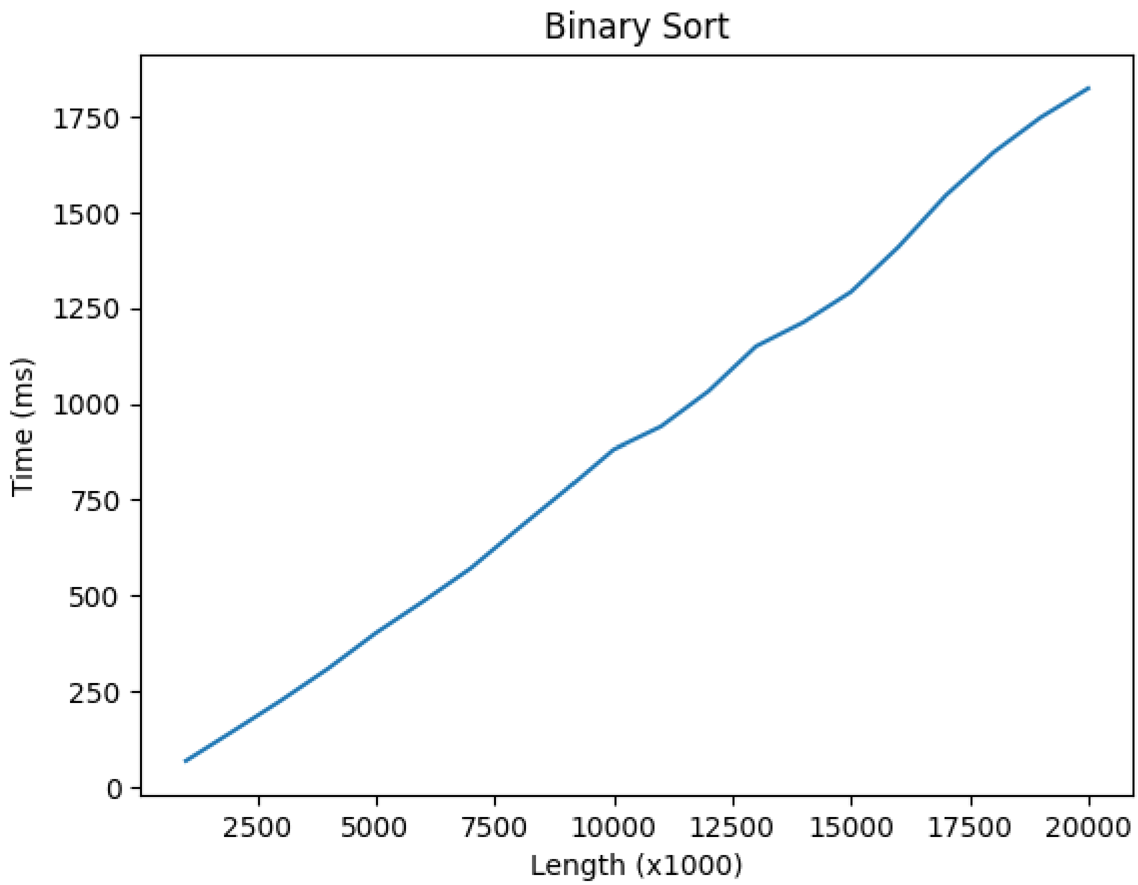


In [41]:
```
pprint(dict(database["Linears"]))
```

```
{1000000: 179465,
 2000000: 363653,
 3000000: 552598,
 4000000: 739291,
 5000000: 936401,
 6000000: 1135539,
 7000000: 1322952,
 8000000: 1523933,
 9000000: 1739115,
 10000000: 2008422,
 11000000: 2134202,
 12000000: 2318000,
 13000000: 2522028,
 14000000: 2738134,
 15000000: 3031808,
 16000000: 3320411,
 17000000: 3369133,
 18000000: 3587394,
 19000000: 3752932,
 20000000: 3998014}
```

Binary Search (*O(lg(n))*)

In [42]:
```
generate_plot("Binary")
```



In [43]:

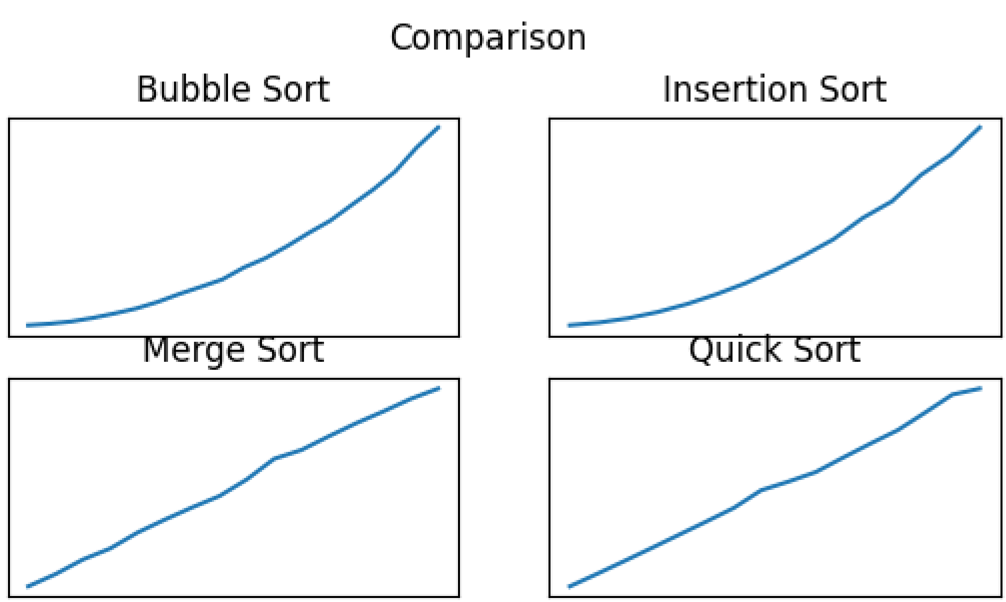
```
pprint(dict(database["Binary"]))
```

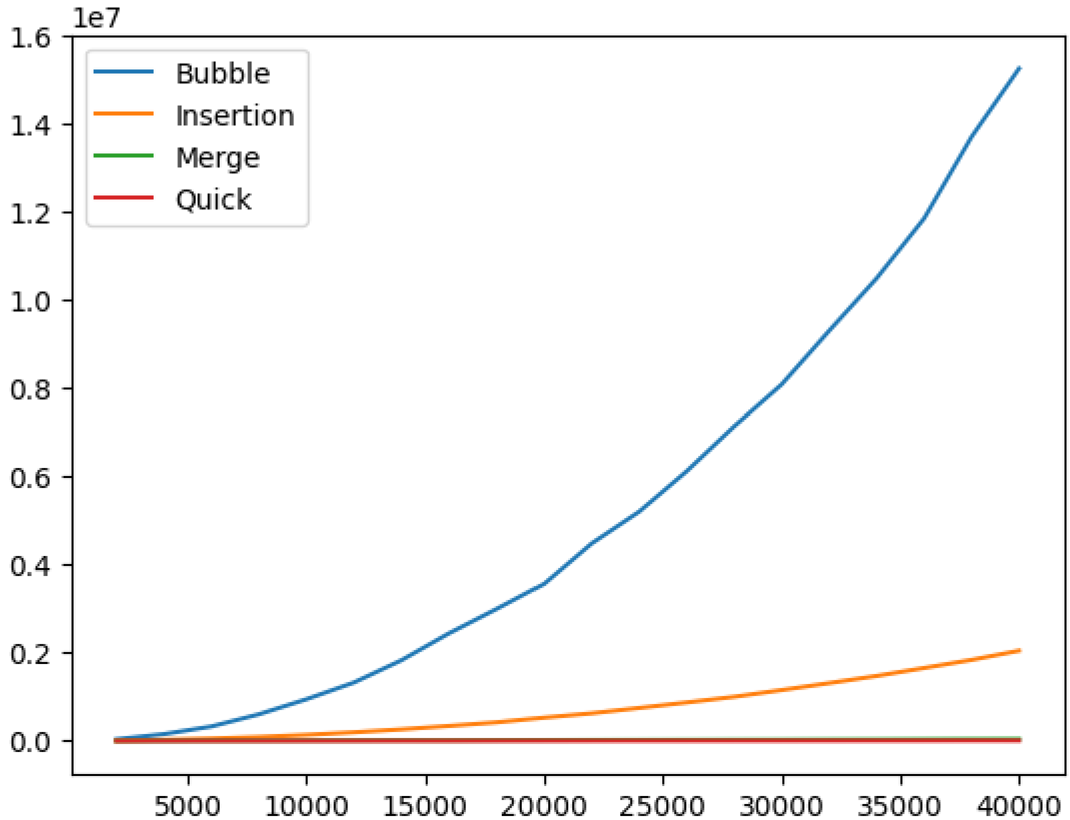
```
{1000000: 68019,
 2000000: 146102,
 3000000: 225486,
 4000000: 309447,
 5000000: 401977,
 6000000: 484795,
 7000000: 571158,
 8000000: 674534,
 9000000: 775959,
 10000000: 881124,
 11000000: 940793,
 12000000: 1032865,
 13000000: 1150035,
 14000000: 1212944,
 15000000: 1292106,
 16000000: 1409654,
 17000000: 1544635,
 18000000: 1656497,
 19000000: 1748409,
 20000000: 1824242}
```

Conclusion

In [10]:

```
i = 1
for sort in ("Bubble","Insertion","Merge","Quick"): #,"Linears","Binary"
    db = database[sort]
    keys   =  [int(i)/1000 for i in db.keys()]
    values =  [i/1000 for i in db.values()]
    plt.subplot(3,2,i)
    plt.plot(keys,values)
    plt.title(f"{sort} Sort")
    plt.xticks([], [])
    plt.yticks([], [])
    i += 1
plt.suptitle("Comparison")
plt.show()

db = {}
for sort in list(database["conclusion"].keys()):
    db[sort] = {}
    for k,v in database["conclusion"][sort].items():
        db[sort][int(k)] = v
for key,value in db.items():
    db[key] = dict(OrderedDict(sorted(db[key].items())))

# print(db["Insertion"])
print("\n")


for sort in ("Bubble","Insertion","Merge","Quick"):
    a = (db[sort])
    plt.plot(a.keys(),a.values())

plt.legend(["Bubble","Insertion","Merge","Quick"])
plt.show()
```





1e7
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
5000
10000
15000
20000
25000
30000
35000
40000
Bubble
Insertion
Merge
Quick