Sorting algorithms

Most of the time, we will used "canned" sorting algorithms when we do data science.

But, it is important to be aware of some common algorithms, so we will look at a couple sorting algorithms

We also need to understand how to characterize the run time or efficiency of an algorithm

Bubble sort is from:

https://www.geeksforgeeks.org/python-program-for-bubble-sort/

Bubble Sort

This was invented by Edward Friend in 1956.

It is not the best available sort, but it is taught in most intro to programming courses, so we need to see it.

In [97]:
```
def bubble_sort(arr):

    nouter=len(arr)

    # Outer loop to iterate through the list n times, starting at the 2nd to last i
    for n in range(nouter - 1, 0, -1):

        # Inner loop to compare adjacent elements
        # start at zero and go up to

        for i in range(n):
            if arr[i] > arr[i + 1]:
                # Swap elements if they are in the wrong order
                arr[i], arr[i + 1] = arr[i + 1], arr[i]

# the outer loop starts with n=length-1
# the inner loop then moves the largest item it finds to the end of the list,  like
# the outer loop then runs at n=length-2, and the second largest item is moved to t


# Sample list to be sorted
arr = [39, 12, 18, 85, 72, 10, 2, 18]
print("Unsorted list is:")
print(arr)

bubble_sort(arr)
```

```
print("Sorted list is:")
print(arr)
```

```
Unsorted list is:
[39, 12, 18, 85, 72, 10, 2, 18]
Sorted list is:
[2, 10, 12, 18, 18, 39, 72, 85]
```

Question/Action

Only one simple change on one line of the function is needed to make it sort in descending form

Copy and paste, then edit to bubble_sort_descending, show that this works.

Dont' alter the loop directions...

In [98]:
```
def bubble_sort_descending(arr):

    nouter=len(arr)

    # Outer loop to iterate through the list n times, starting at the 2nd to last i
    for n in range(nouter - 1, 0, -1):

        # Inner loop to compare adjacent elements
        # start at zero and go up to

        for i in range(n):
            if arr[i] < arr[i + 1]:
                # Swap elements if they are in the wrong order
                arr[i], arr[i + 1] = arr[i + 1], arr[i]

# the outer loop starts with n=length-1
# the inner loop then moves the largest item it finds to the end of the list,  like
# the outer loop then runs at n=length-2, and the second largest item is moved to t


# Sample list to be sorted
arr = [39, 12, 18, 85, 72, 10, 2, 18]
print("Unsorted list is:")
print(arr)

bubble_sort_descending(arr)

print("Sorted list is:")
print(arr)
```

```
Unsorted list is:
[39, 12, 18, 85, 72, 10, 2, 18]
Sorted list is:
[85, 72, 39, 18, 18, 12, 10, 2]
```

time

How fast is our code? How do we describe this?

Let's use %timeit to time how long a bubble sort takes at some different array lengths

The number of repetitions of the loop is n*(n-1), so the run time should be proportional to n^2

The speed of the algorithm is said to be order of n^2, written as *O*(n^2)

In Computer science, we really want to know the order of the algorithms we use

In [99]:
```
import random
import numpy as np
random_numbers = [random.random() for _ in range(50)]
```

In [100…
```
#50 values in random numbers

b50 = %timeit -o bubble_sort(random_numbers)
b50 = b50.best
```

```
43.8 µs ± 2 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

In [101…
```
random_numbers = [random.random() for _ in range(500)]
b500 = %timeit -o bubble_sort(random_numbers)
b500 = b500.best
```

```
4.09 ms ± 112 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [102…
```
random_numbers = [random.random() for _ in range(2000)]
b2000 = %timeit -o bubble_sort(random_numbers)
b2000 = b2000.best
```

```
86.4 ms ± 2.08 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [103…
```
random_numbers = [random.random() for _ in range(5000)]
b5000 = %timeit -o bubble_sort(random_numbers)
b5000 = b5000.best
```

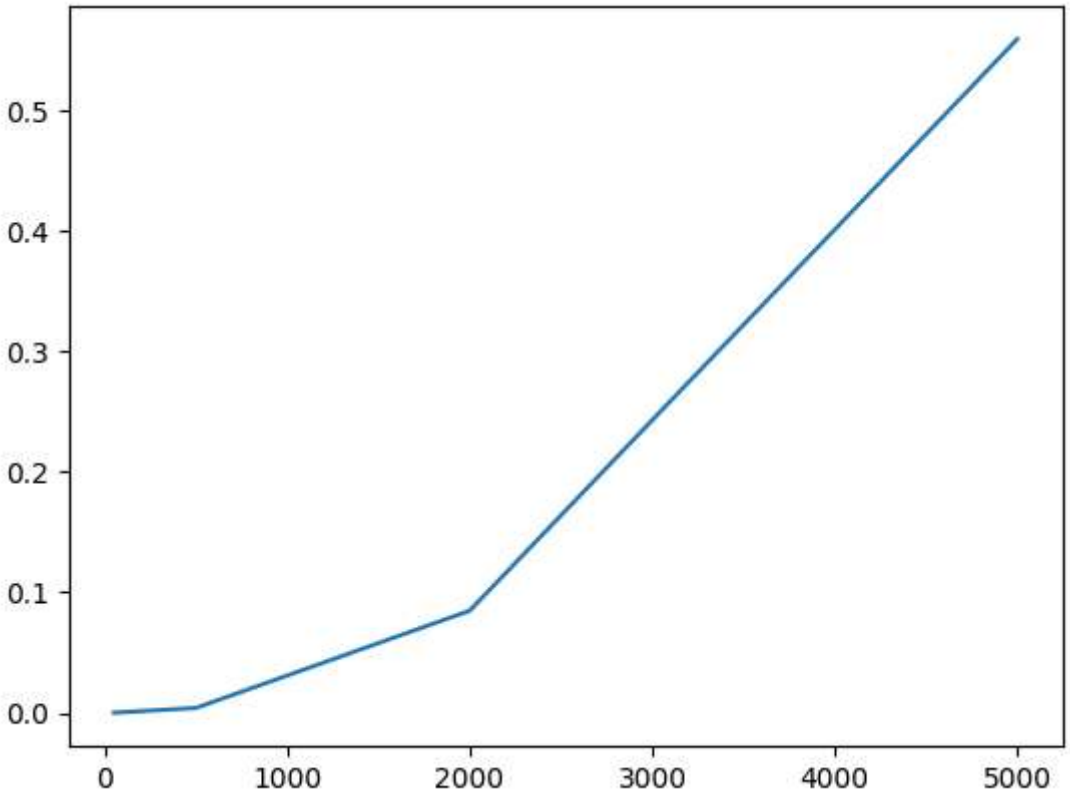
```
581 ms ± 13.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [104…
```
import matplotlib.pyplot as plt

bubble_t=[b50, b500, b2000, b5000]
bubble_n=[50,500,2000,5000]

plt.plot(bubble_n,bubble_t)
```

Out[104…
```
[<matplotlib.lines.Line2D at 0x23d559e33e0>]
```



A better sorting algorithm

The mergesort is a faster, more efficient way to sort.

In [105…

```
import math   # needed for the floor() operation

def merge(left, right):                          #left and right are two lists, each
    result = []                                  # result will hold the merged list
    i ,j = 0, 0                                  # "cursor" positions in the two lis
    while i < len(left) and j < len(right):     # repeat until we run out of items i
        if left[i] <= right[j]:                  # figure out which is smaller, the n
            result.append(left[i])               # increase i or j to the next time i
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]                           # add the remaining items in each li
    result += right[j:]
    return result                                # we have now merged left and right

def mergesort(list):
    if len(list) < 2:
        return list
    middle = math.floor(len(list) / 2)
    left = mergesort(list[:middle])               # note the recursion,   it will ke
                                                     # then merger the 1 element lis
                                                     # to the right list,
```

```
    right = mergesort(list[middle:])
    return merge(left, right)
```

In [106…
```
mergesort(arr)
```

Out[106…
```
[2, 10, 12, 18, 18, 39, 72, 85]
```

How does the Mergesort work?

Note this implementation is recursive.

If the length of a list is 1, we just return it

Other wise

1.) find the midpoint 2.) Mergesort the left half 3.) Mergestor the right half 3.) "Merge" the two sorted halves of the two lists, in the values in the two lists are combined back into a single list, always by adding the smallest value in either the two lists into the single merged list.

How fast should mergesort be?

This can get complicated, but it appears that mergesort is faster than

O($n log n + n + O(log n)$)

the lead term here is nlog(n), which is less than n^2

mergesort is a bit unpredictable in it's timing, it can vary greatly with the amount of order already present

In [107…
```
random_numbers = [random.random() for _ in range(500)]
m500 = %timeit -o mergesort(random_numbers)
m500 = m500.best
```

```
590 µs ± 10.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [108…
```
random_numbers = [random.random() for _ in range(5000)]
m5000 = %timeit -o mergesort(random_numbers)
m5000 = m5000.best
```

```
7.93 ms ± 139 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Question/Action

Generate a series of tests of the time to run a mergesort over the same range of n values we used for the bubblesort and set up lists of n_merge and t_merger.

Plot the time vs n for the bubblesort and the mergesort on a single graph.

In [109…
```
random_numbers = [random.random() for _ in range(50)]
m50 = %timeit -o mergesort(random_numbers)
m50 = m50.best
```

```
43.7 µs ± 1.54 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

In [110…
```
random_numbers = [random.random() for _ in range(2000)]
m2000 = %timeit -o mergesort(random_numbers)
m2000 = m2000.best
```

```
2.9 ms ± 64.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [111…
```
merge_t=[m50, m500, m2000, m5000]
merge_n=[50,500,2000,5000]

plt.plot(bubble_n, merge_t, label = "bubble")
plt.plot(merge_n, bubble_t, label = "merge")
```

Out[111…
```
[<matplotlib.lines.Line2D at 0x23d55ccef30>]
```