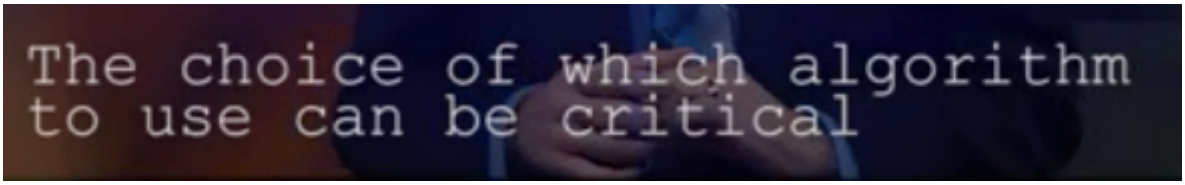
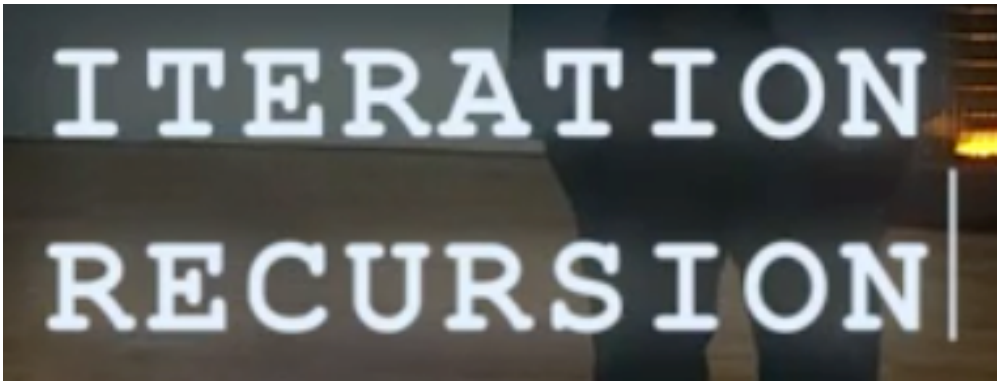
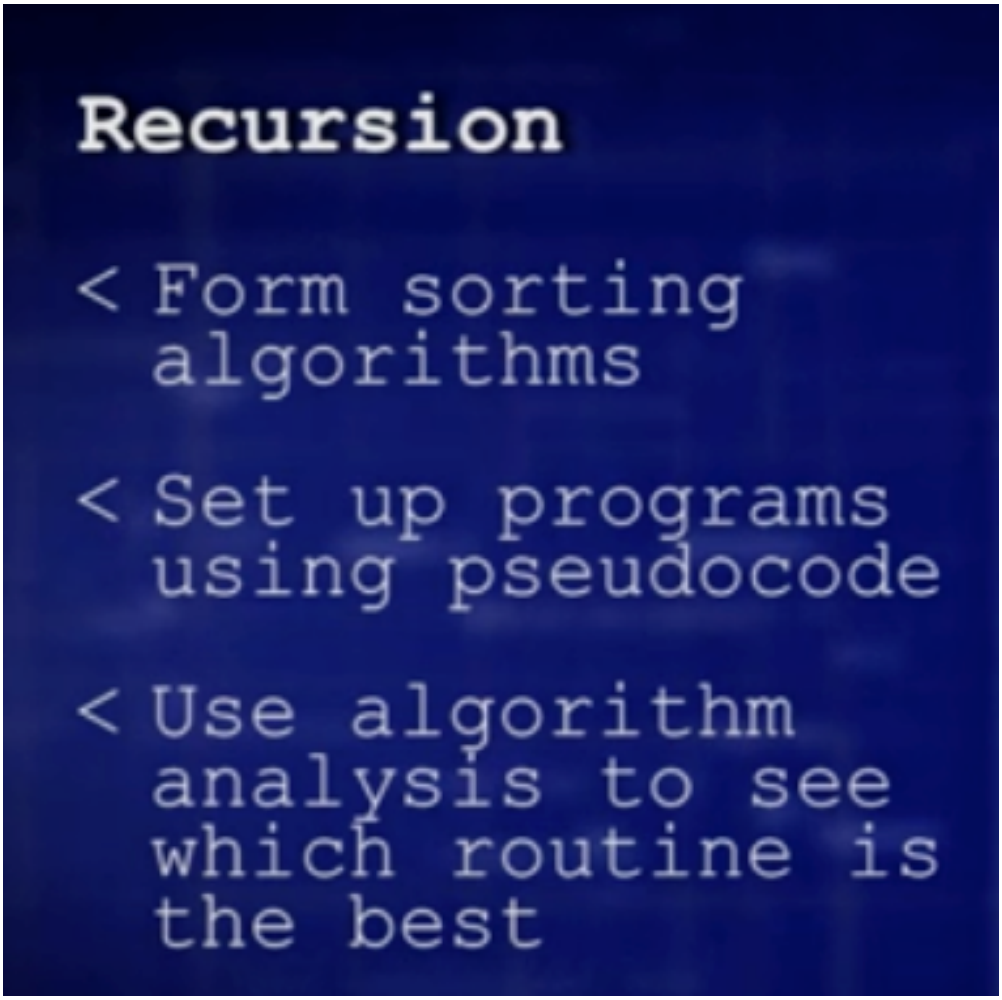
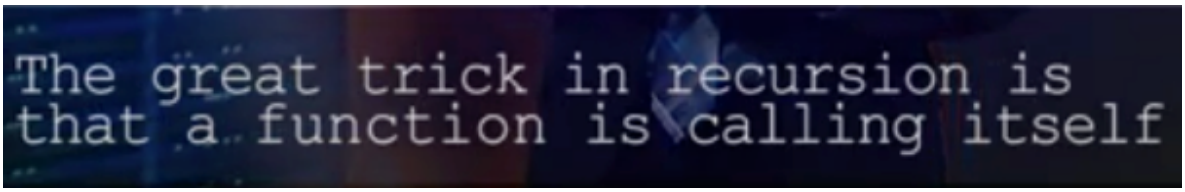
CSIes21
Recursion and Running Time



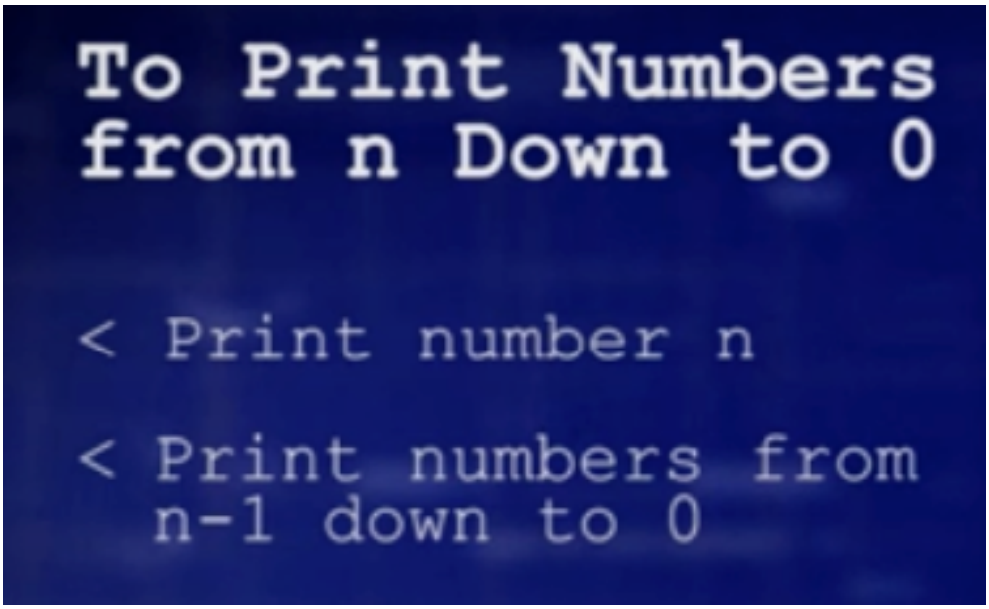







```
5
4
3
2
1
0
```

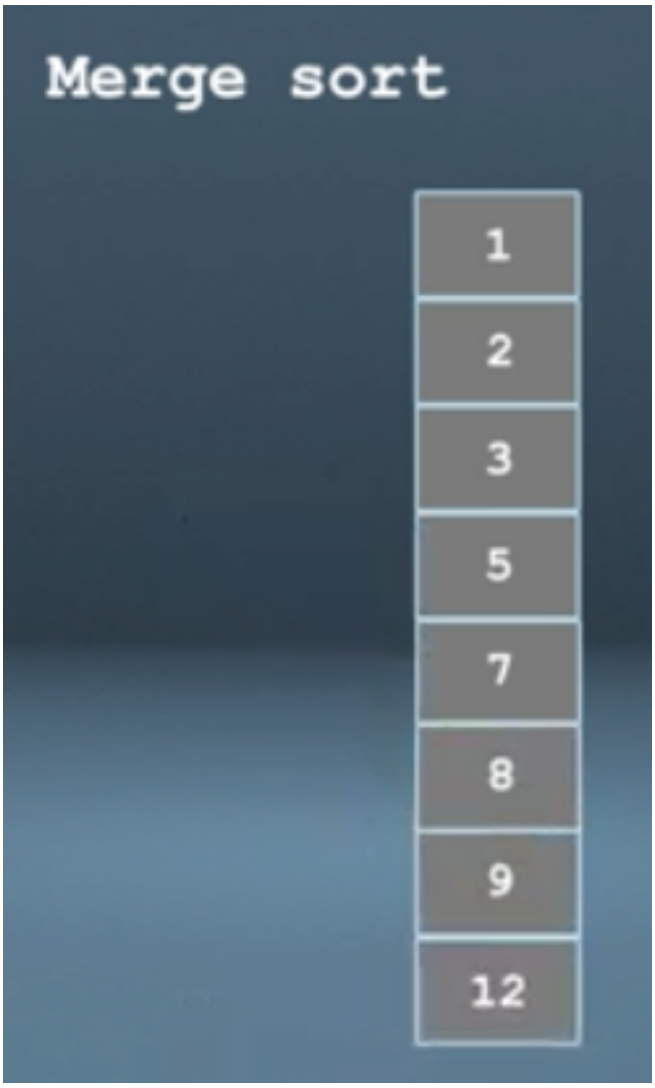
```
1 def countdown(n):
2     print (n)
3     if n > 0:
4         countdown(n-1)
5
```


To Print Numbers from n Down to 0
< Print number n
< Print numbers from n-1 down to 0


Divide and conquer
Taking a large data set and dividing it into subsets that we handle independently


Merge sort
3
9
8
1
5
2
12
7


Merge sort
1
8
2
5
7
12




```
#Input: unsorted list L, length n
#Output: L, with elements sorted smallest to largest

If n <= 1
    Return L
    #a list of length 1 is already sorted
```

Base case

```
#Input: unsorted list L, length n
#Output: L, with elements sorted smallest to largest

If n <= 1
    Return L
    #a list of length 1 is already sorted
L1 = L(0:n/2-1) L2 = (n/2:n-1)
MergeSort(L1) MergeSort(L2)
L = Merge(L1, L2)
#Merge will be defined separately
```

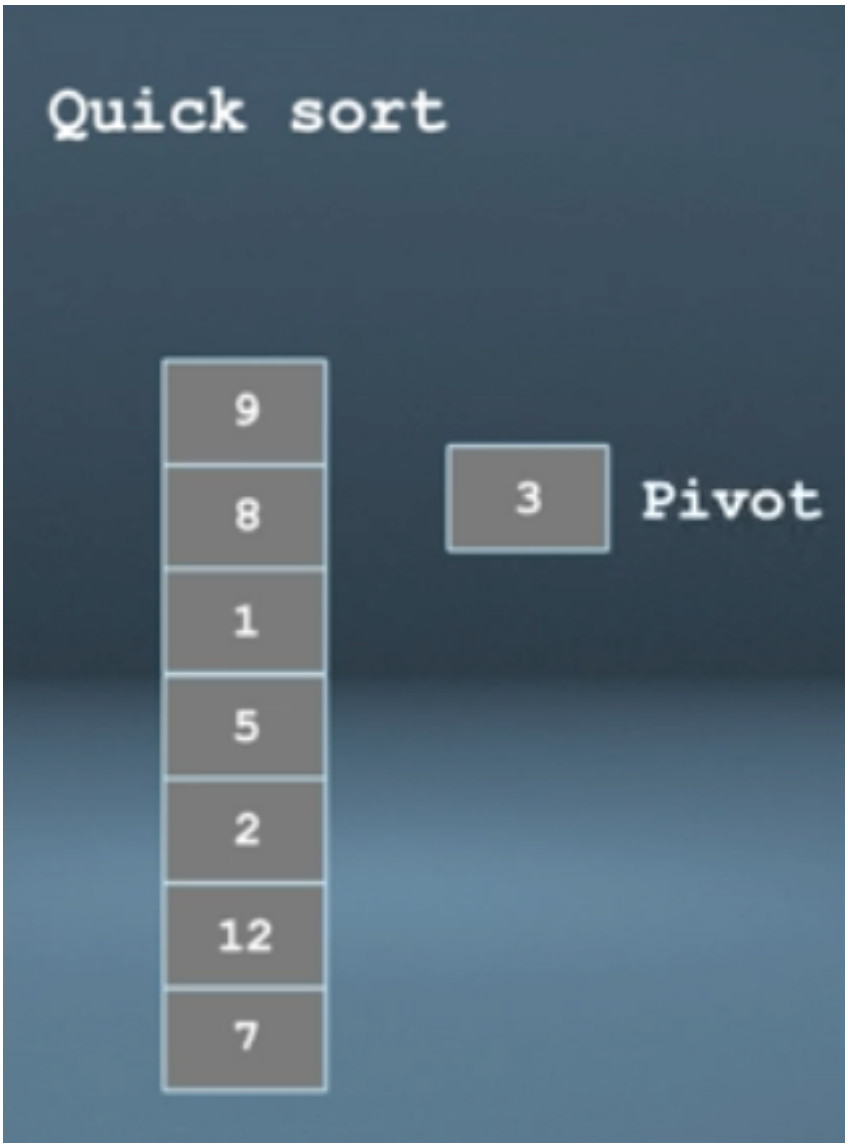
```
1  def mergeSort(L):
2      n = len(L)
3      if n <= 1:
4          return
5      L1 = L[:n//2]
6      L2 = L[n//2:]
7      mergeSort(L1)
8      mergeSort(L2)
9      merge(L, L1, L2)
10     return
11
12
13
14
15
16
```

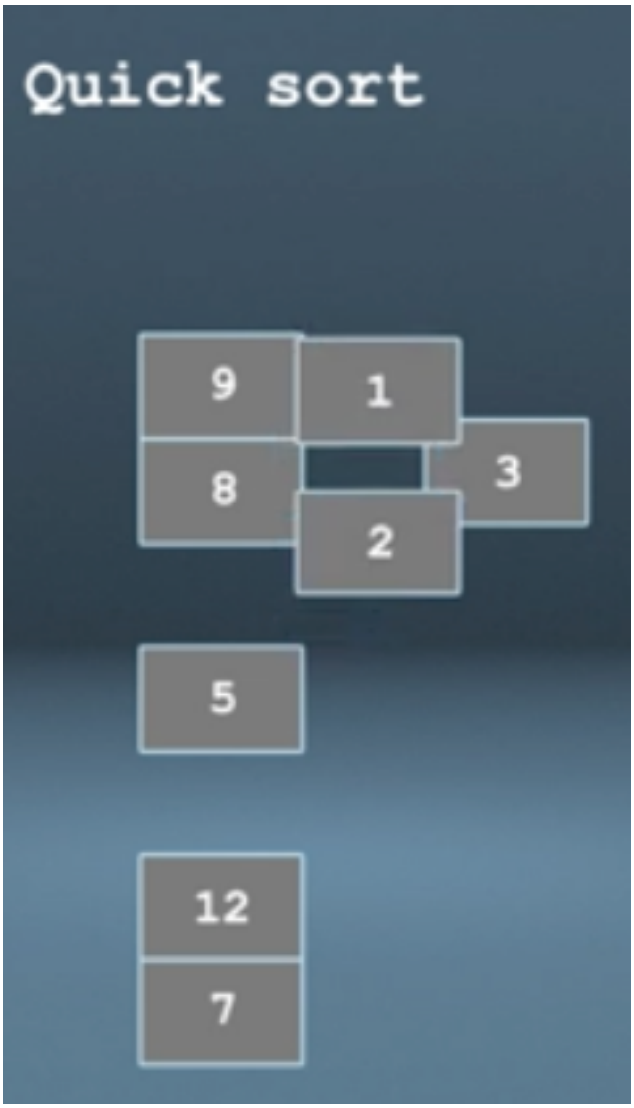
```
If n <= 1
    Return L

L1 = L(0:n/2-1) L2 = (n/2:n-1)
MergeSort(L1) MergeSort(L2)
L = Merge(L1, L2)
```

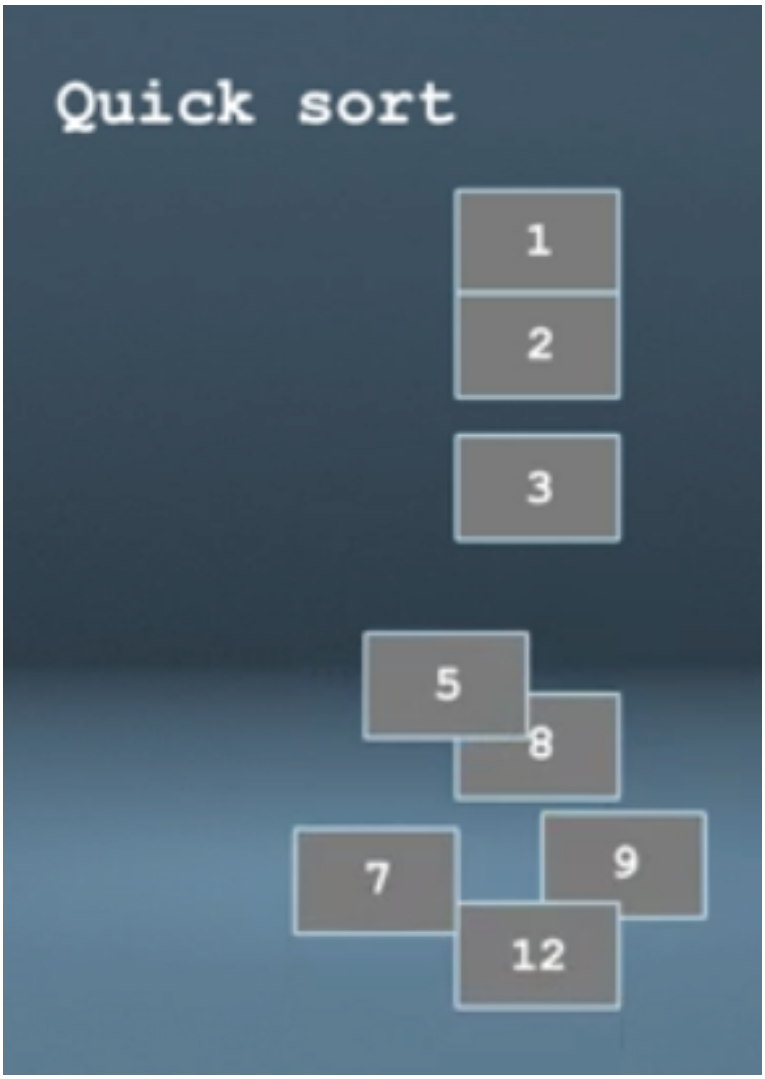
```
def merge(L, L1, L2):
    i = 0
    j = 0
    k = 0
    while (j < len(L1)) or (k < len(L2)):
        if j < len(L1):
            if k < len(L2):
                # we are not at the end of L1 or L2, so pull the smaller value
                if L1[j] < L2[k]:
                    L[i] = L1[j]
                    j += 1
                else:
                    L[i] = L2[k]
                    k += 1
            else:
                # we are at the end of L2, so just pull from L1
                L[i] = L1[j]
                j += 1
        else:
            # we are at the end of L1, so just pull from L2
            L[i] = L2[k]
            k += 1
        i += 1
    return
```

```
['barbeque', 'chicken and dumplings', 'gumbo', 'ice cream', 'pecan pie', 'pizza']
```

```
favorite_foods = ['pizza', 'barbeque', 'gumbo', 'chicken and dumplings', 'pecan pie', 'ice cream']
mergeSort(favorite_foods)
print(favorite_foods)
```


Quick sort
9
8
1
5
2
12
7
3
Pivot


Quick sort
9
1
8
3
2
5
12
7


Quick sort
1
2
3
5
8
7
9
12


Quick sort
1
2
3
5
7
8
9
12

Quick Sort

< Input: unsorted list L, length n

< Output: L, with elements sorted smallest to largest

Quick Sort

< Input: unsorted list L, length n

< Output: L, with elements sorted smallest to largest

```
1. If n <= 1
     1. Return L
          #a list of length 1 is already sorted
```

Quick Sort

< Input: unsorted list L, length n

< Output: L, with elements sorted smallest to largest

1. If n <= 1
    1. Return L
        #a list of length 1 is already sorted
2. pivot = L[0]
3. Use remaining elements to form list L1 less than, and L2 greater than, the pivot

Quick Sort

< Input: unsorted list L, length n

< Output: L, with elements sorted smallest to largest

1. If n <= 1
    1. Return L
        #a list of length 1 is already sorted
2. pivot = L[0]
3. Use remaining elements to form list L1 less than, and L2 greater than, the pivot
4. QuickSort(L1) QuickSort(L2)

Quick Sort

< Input: unsorted list L, length n

< Output: L, with elements sorted smallest to largest

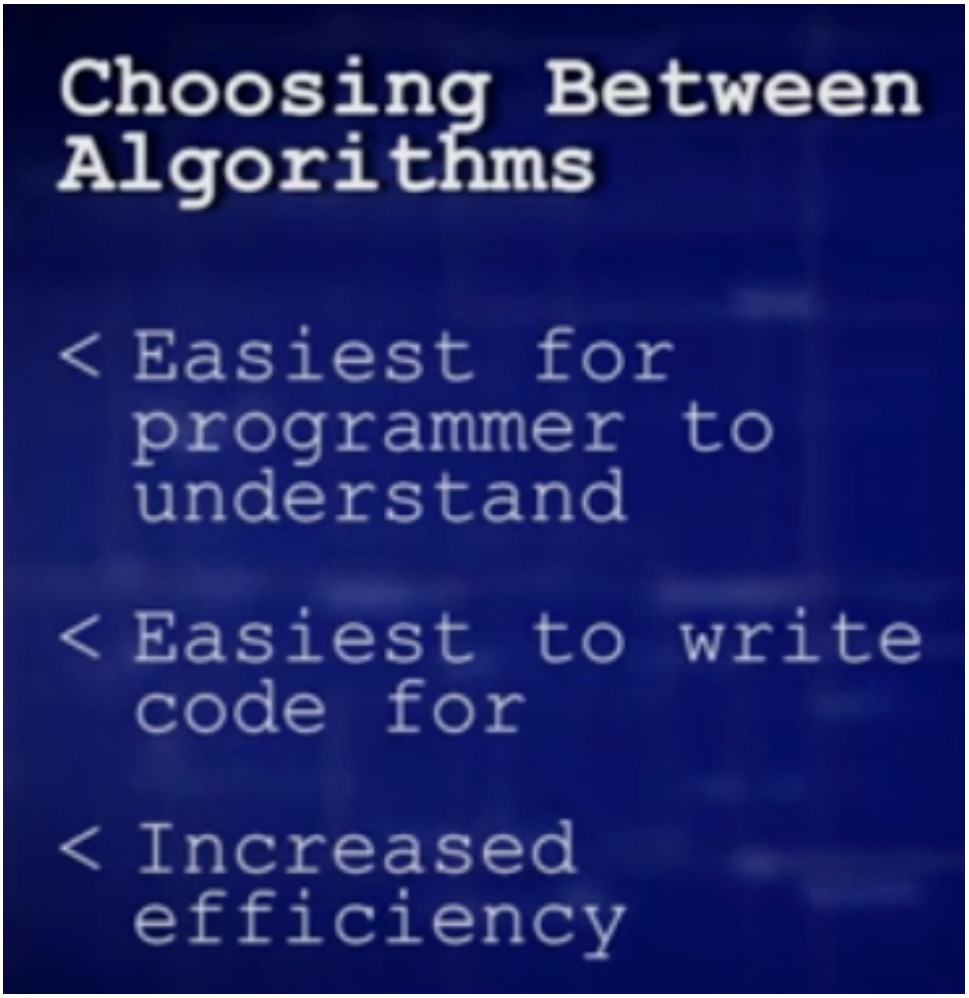
1. If n <= 1
    1. Return L
        #a list of length 1 is already sorted
2. pivot = L[0]
3. Use remaining elements to form list L1 less than, and L2 greater than, the pivot
4. QuickSort(L1) QuickSort(L2)
5. L = Join(L1, pivot, L2)

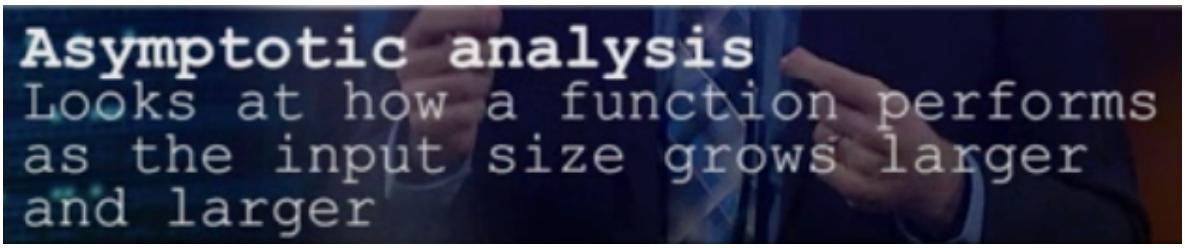
```
1  def quickSort(L):
2      #handle base case
3      if len(L) <= 1:
4          return
5
6      #pick pivot
7      pivot = L[0]
8
9      #form lists less/greater than pivot
10     L1 = []
11     L2 = []
12     for element in L[1:]:
13         if element < pivot:
14             L1.append(element)
15         else:
16             L2.append(element)
17
18     #sort sublists
19     quickSort(L1)
20     quickSort(L2)
21
22     #join the sublists and pivot
23     L[:] = []
24     for element in L1:
25         L.append(element)
26     L.append(pivot)
27     for element in L2:
28         L.append(element)
29
30     return
```

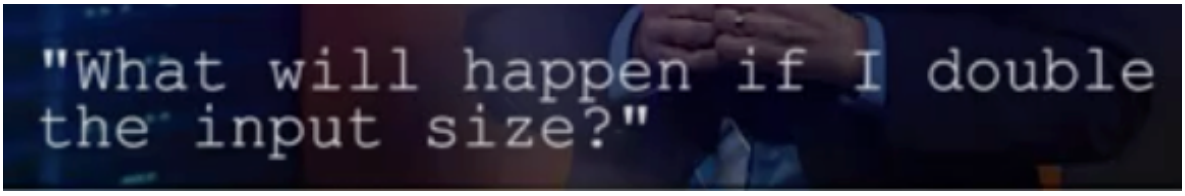
```
'barbeque', 'chicken and dumplings', 'gumbo', 'ice cream', 'pecan pie', 'pizza'
```

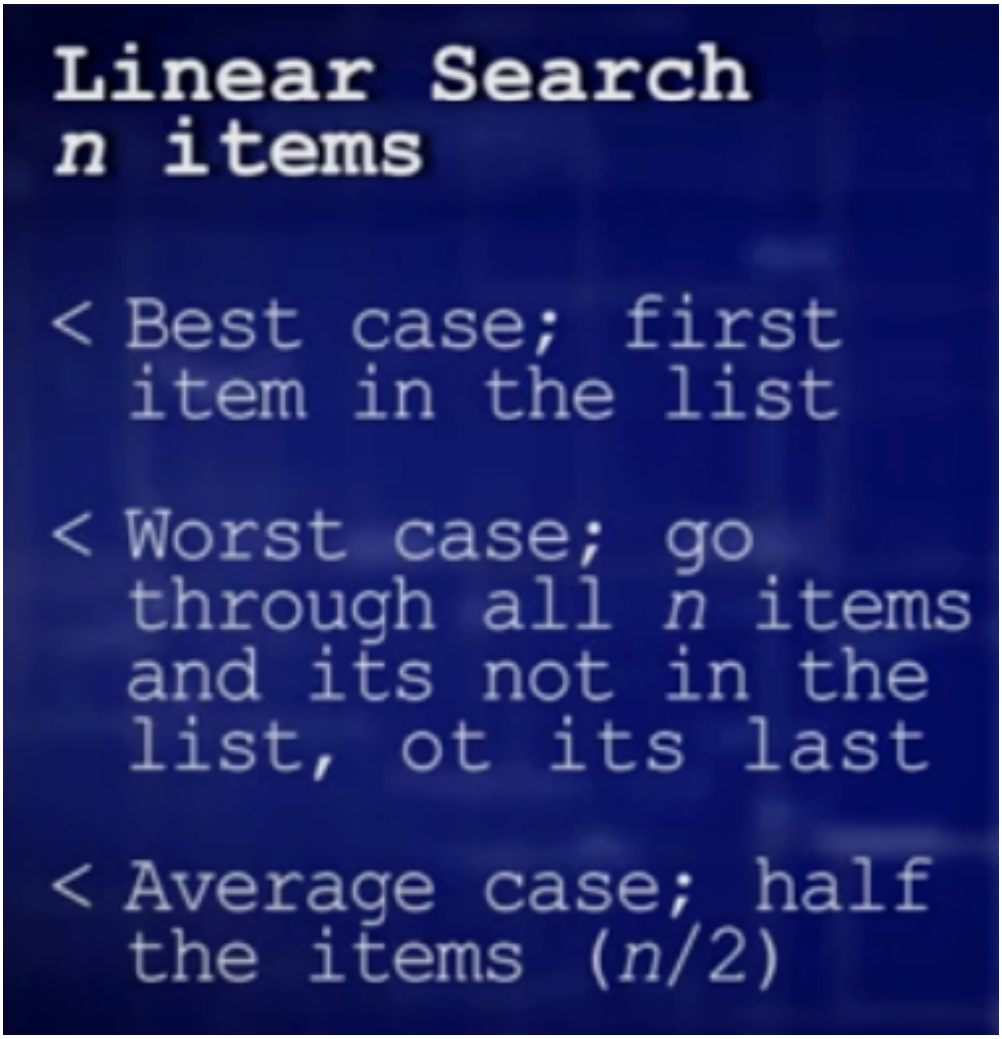
```
1  favorite_foods = ['pizza', 'barbeque', 'gumbo', 'chicken and dumplings', 'pecan pie', 'ice cream']
2  quickSort(favorite_foods)
3  print(favorite_foods)
4
```


Sort Routines
< Selection sort
< Insertion sort
< Merge sort
< Quick sort


Choosing Between Algorithms
< Easiest for programmer to understand
< Easiest to write code for
< Increased efficiency


Asymptotic analysis
Looks at how a function performs as the input size grows larger and larger


"What will happen if I double the input size?"


Linear Search
n items
< Best case; first item in the list
< Worst case; go through all n items and its not in the list, ot its last
< Average case; half the items (n/2)

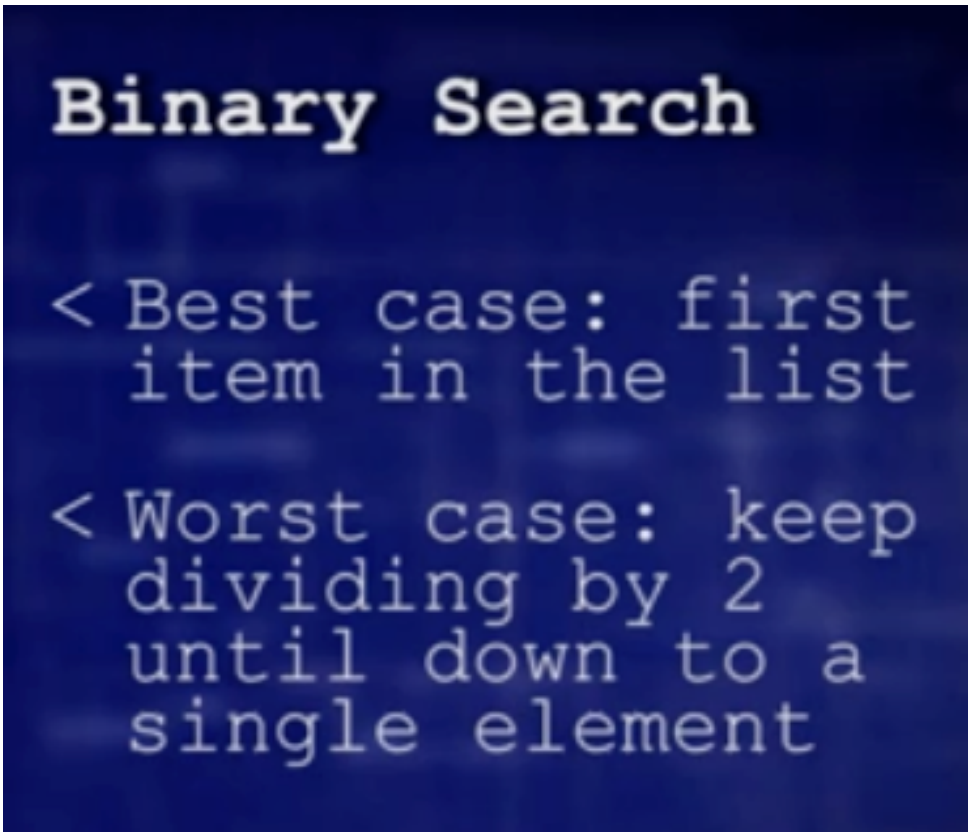
Linear Search
2*n* items

- Best case; first item in the list
- Worst case; go through all 2*n* items and its not in the list
- Average case; half the items (2*n*/2=*n*)

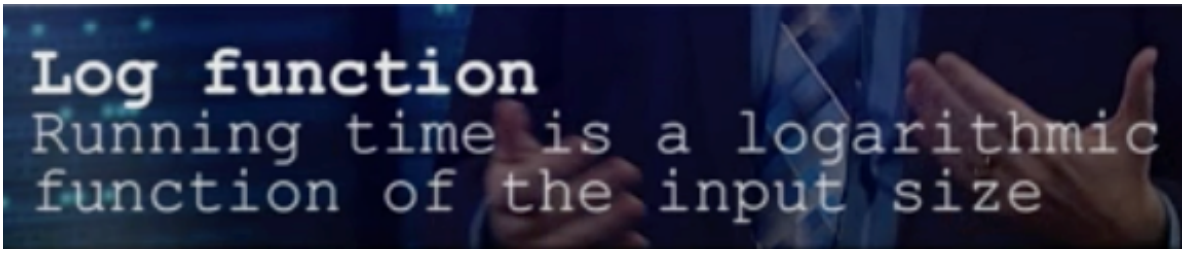
Linear Search
2*n* items

- Best case; first item in the list
- Worst case; go through all 2*n* items and its not in the list
- Average case; half the items (2*n*/2=*n*)

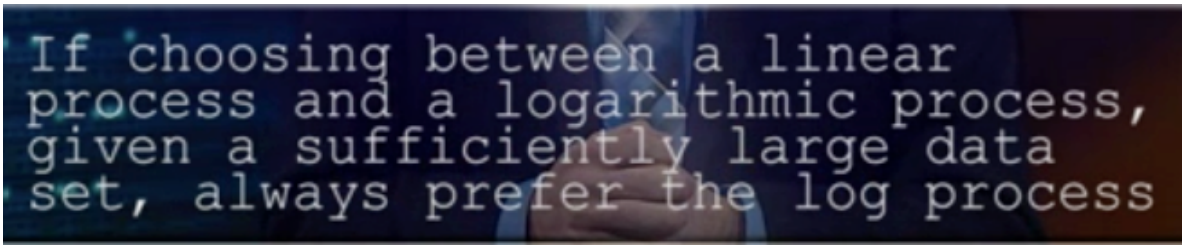
Linear search; O(n)


Binary Search
< Best case: first item in the list
< Worst case: keep dividing by 2 until down to a single element

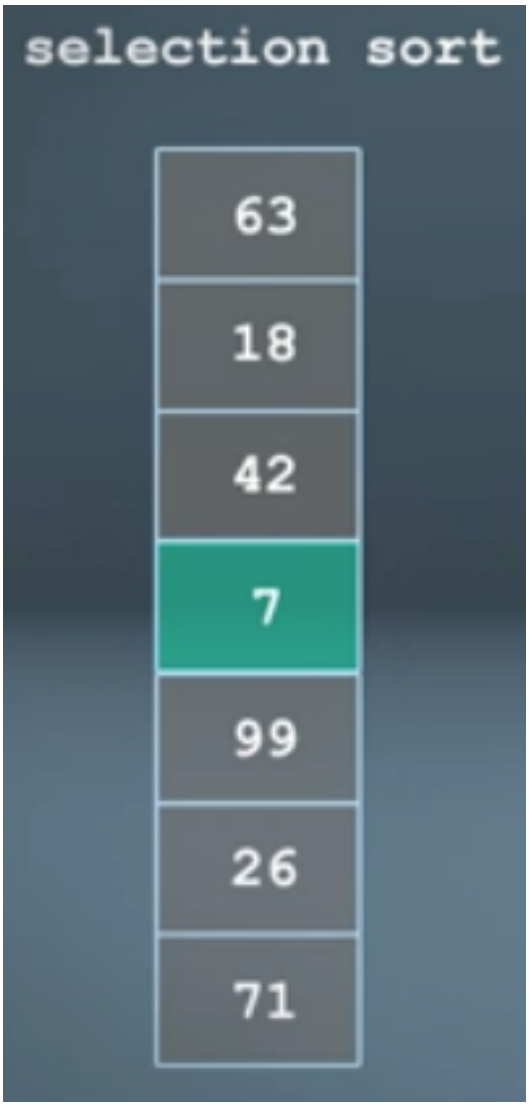
Size of List	Max Number of iterations
1	1
2	2
3	2
4	3
5	3
6	3
7	3
8	4
9-15	4
16-31	5
32-63	6
64-127	7
128-255	8
256-511	9
512-1023	10
2^{n-1} through $2^{n}-1$	n




Binary Search
< Best case: first item in the list
< Worst case: keep dividing by 2 until down to a single element
O(log2 n)


If choosing between a linear process and a logarithmic process, given a sufficiently large data set, always prefer the log process


selection sort
63
18
42
7
99
26
71


selection sort
63
18
42
7
99
26
71

Selection Sort

< n

< n-1

< n-2

< n-3

< ...

< 1

< **Summation formula:** $\frac{n(n+1)}{2}$

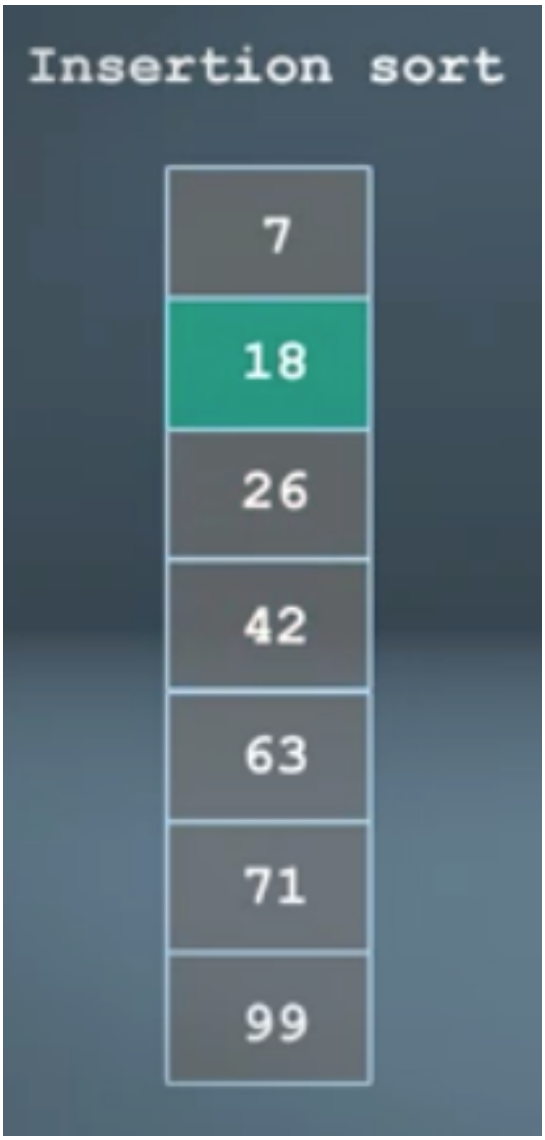
< **Quadratic running time:** $O(n^2)$

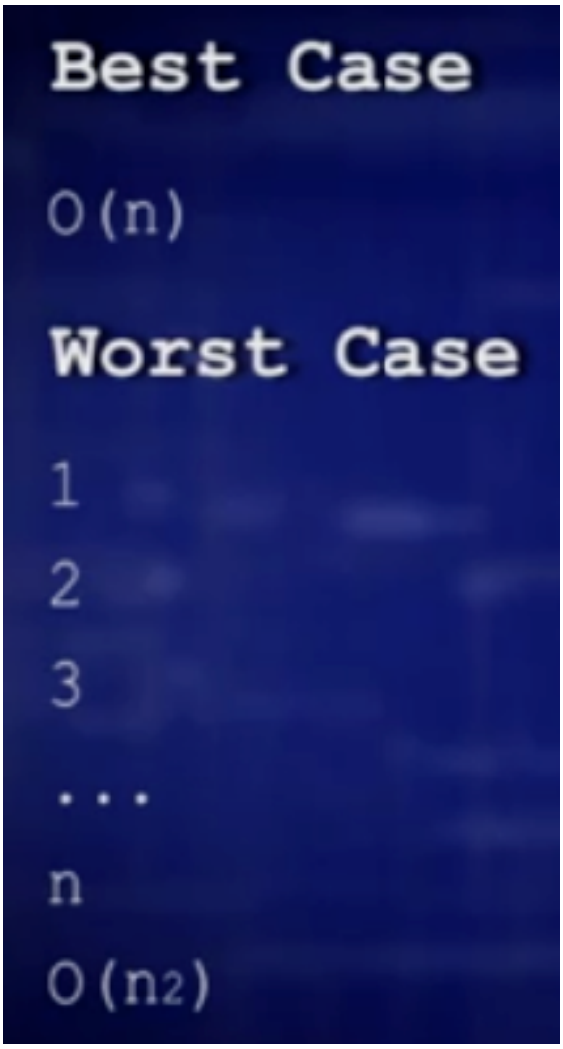
Quick Sort

< Input: unsorted list L, length n

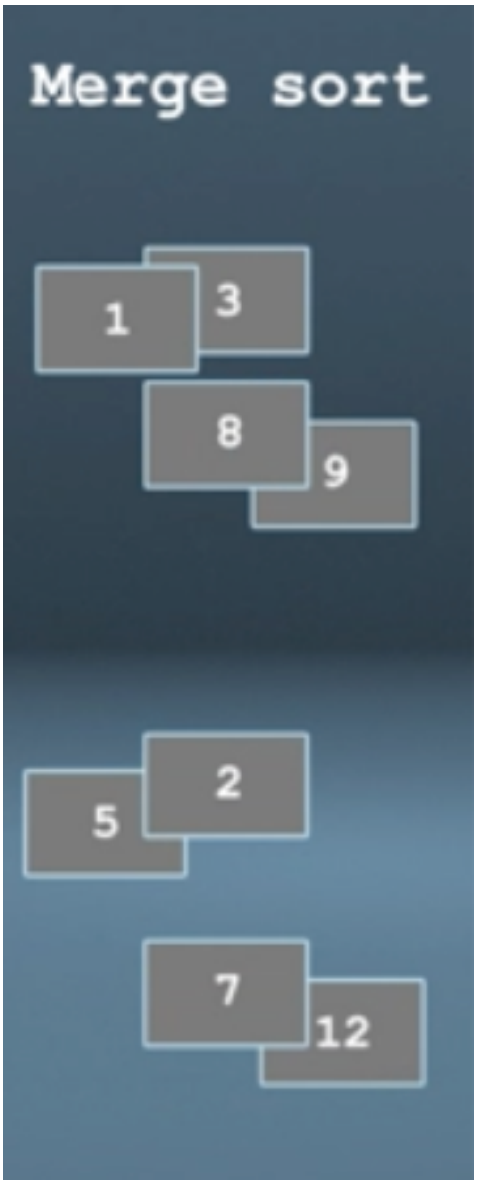
< Output: L, with elements sorted smallest to largest

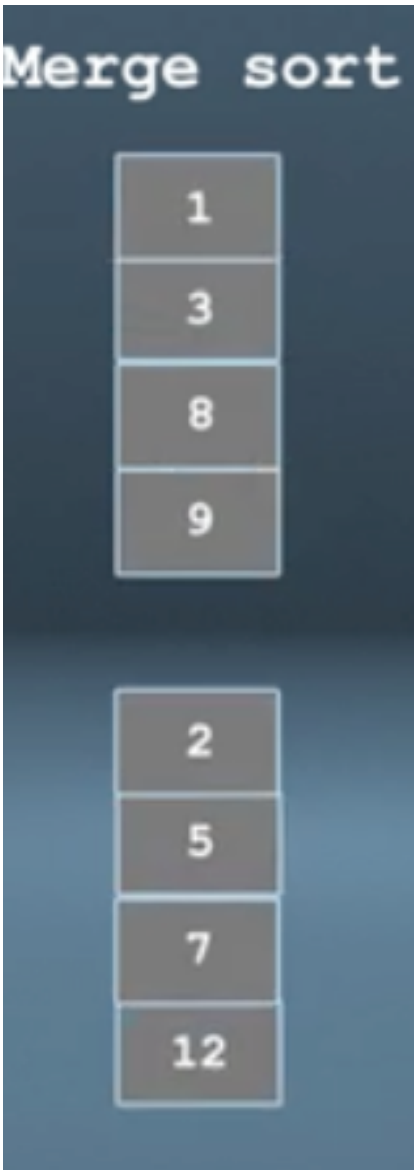
```
1. If n <= 1
      1. Return L
         #a list of length 1 is already sorted

2. pivot = L[0]

3. Use remaining elements to form list L1 less
   than, and L2 greater than, the pivot

4. QuickSort(L1) QuickSort(L2)

5. L = Join(L1, pivot, L2)
```

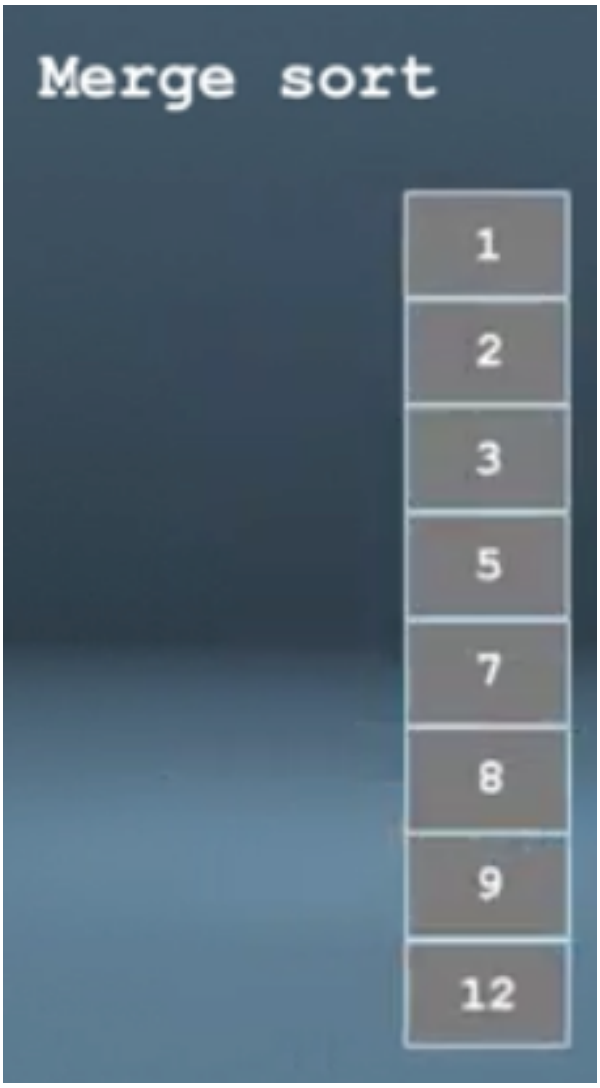

Insertion sort
7
18
26
42
63
71
99


Best Case
O(n)
Worst Case
1
2
3
...
n
O(n2)

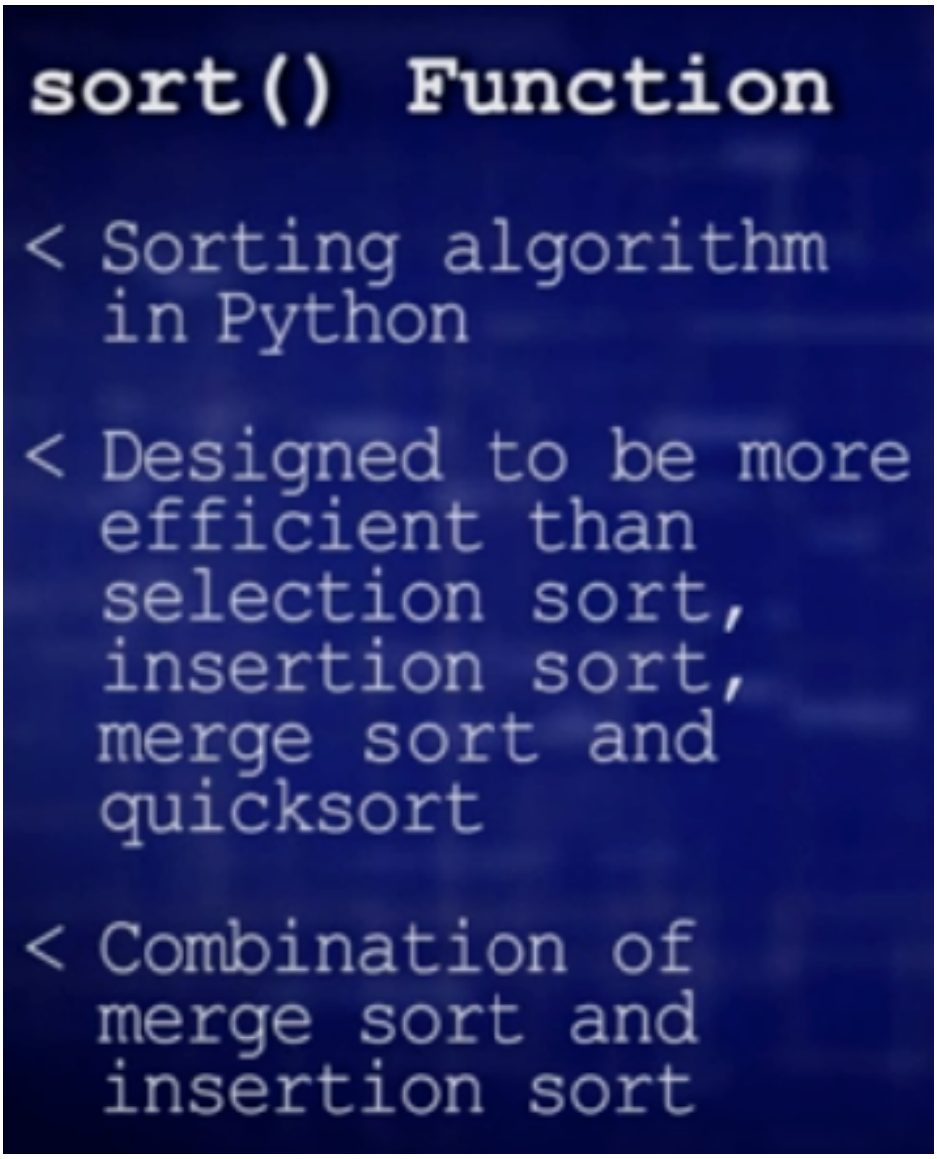

Merge sort
3
9
8
1
5
2
12
7

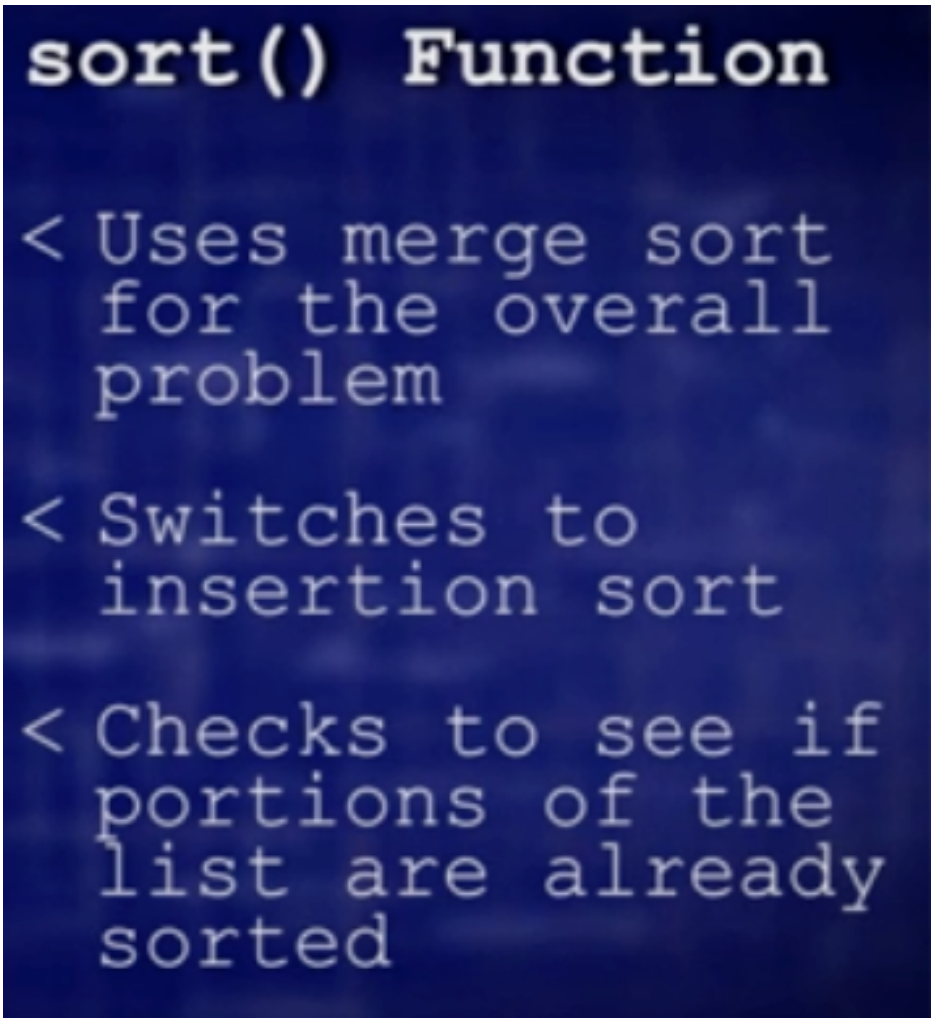

Merge sort
1
3
8
9
2
5
7
12


Merge sort
1
3
8
9
2
5
7
12



Operation	Best	Average	Worst
Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quicksort	O(n log n)	O(n log n)	$O(n^2)$


sort() Function
< Sorting algorithm in Python
< Designed to be more efficient than selection sort, insertion sort, merge sort and quicksort
< Combination of merge sort and insertion sort


sort() Function
< Uses merge sort for the overall problem
< Switches to insertion sort
< Checks to see if portions of the list are already sorted

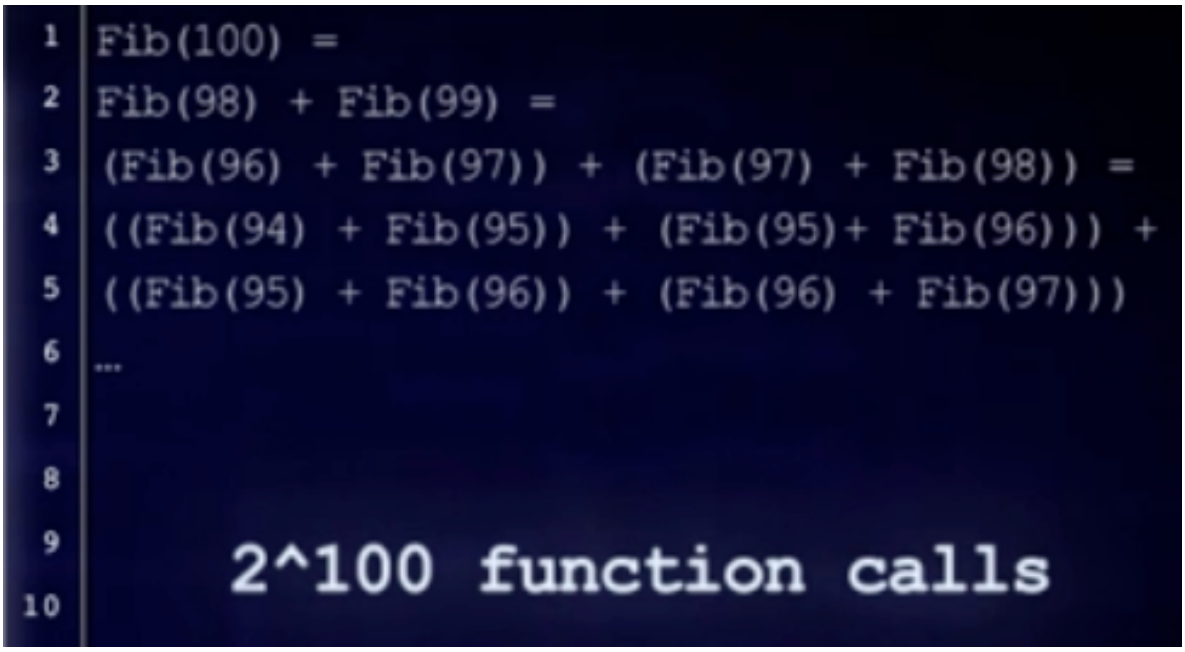
Fibonacci Numbers

< F(0) = 0, F(1) = 1

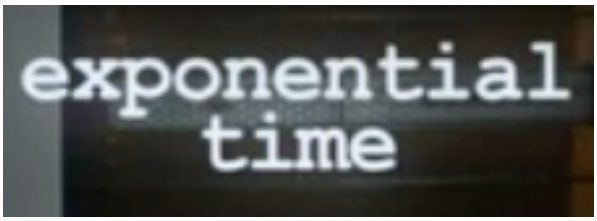
< F(n) = F(n-2) + F(n-1)

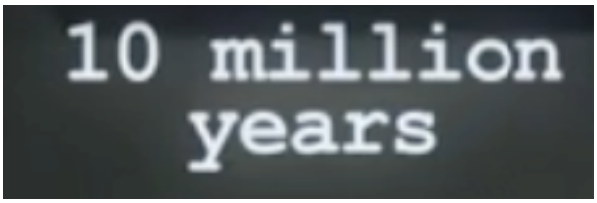
< 0
1
(0+1) = 1
(1+1) = 2
(1+2) = 3
(2+3) = 5
(3+5) = 8
...

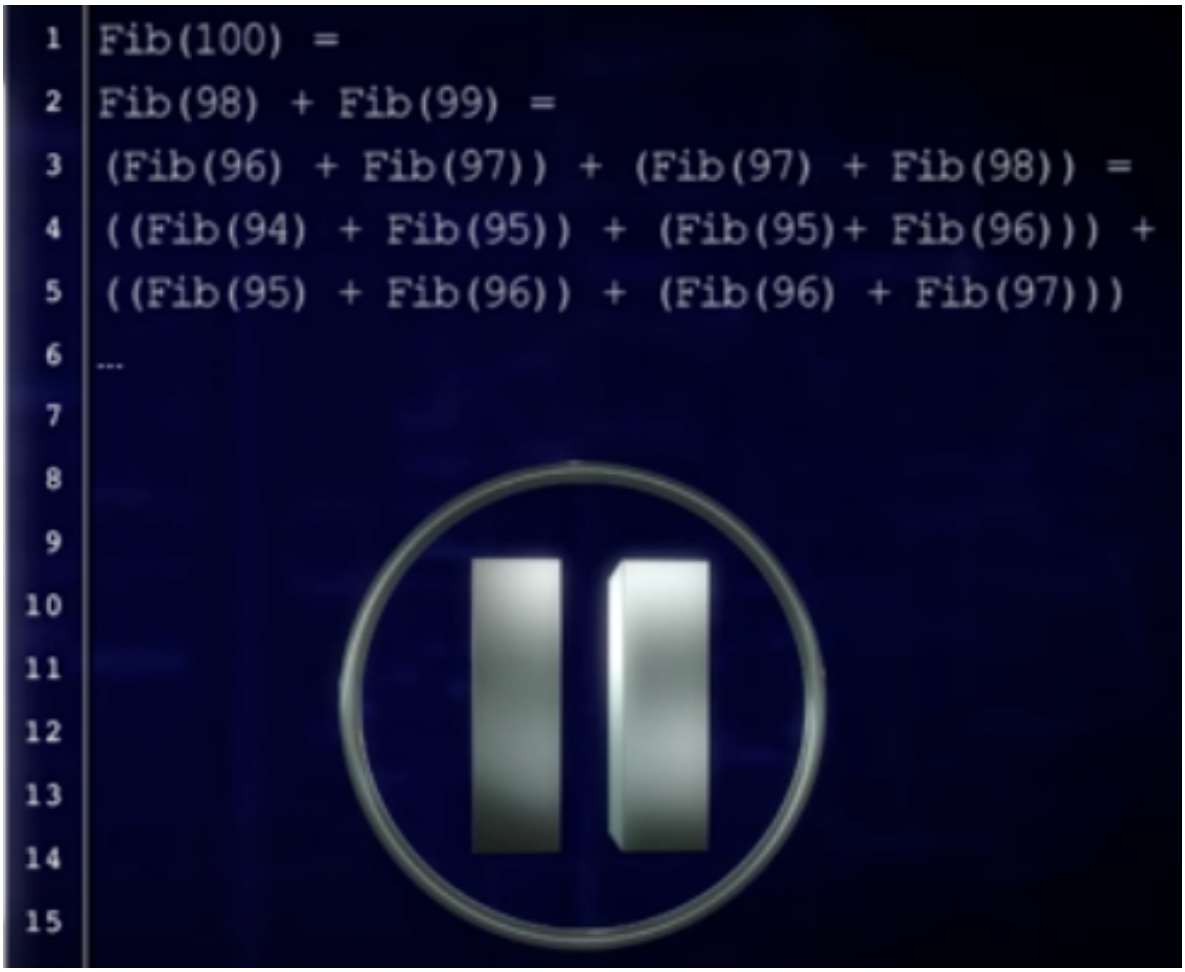
```
1 def Fib(n):
2     if n == 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         return Fib(n-2) + Fib(n-1)
8
```


1 Fib(100) =
2 Fib(98) + Fib(99) =
3 (Fib(96) + Fib(97)) + (Fib(97) + Fib(98)) =
4 ((Fib(94) + Fib(95)) + (Fib(95)+ Fib(96))) +
5 ((Fib(95) + Fib(96)) + (Fib(96) + Fib(97)))
6 ...
7
8
9
10
2^100 function calls


Nonillion
trillion * trillion * trillion

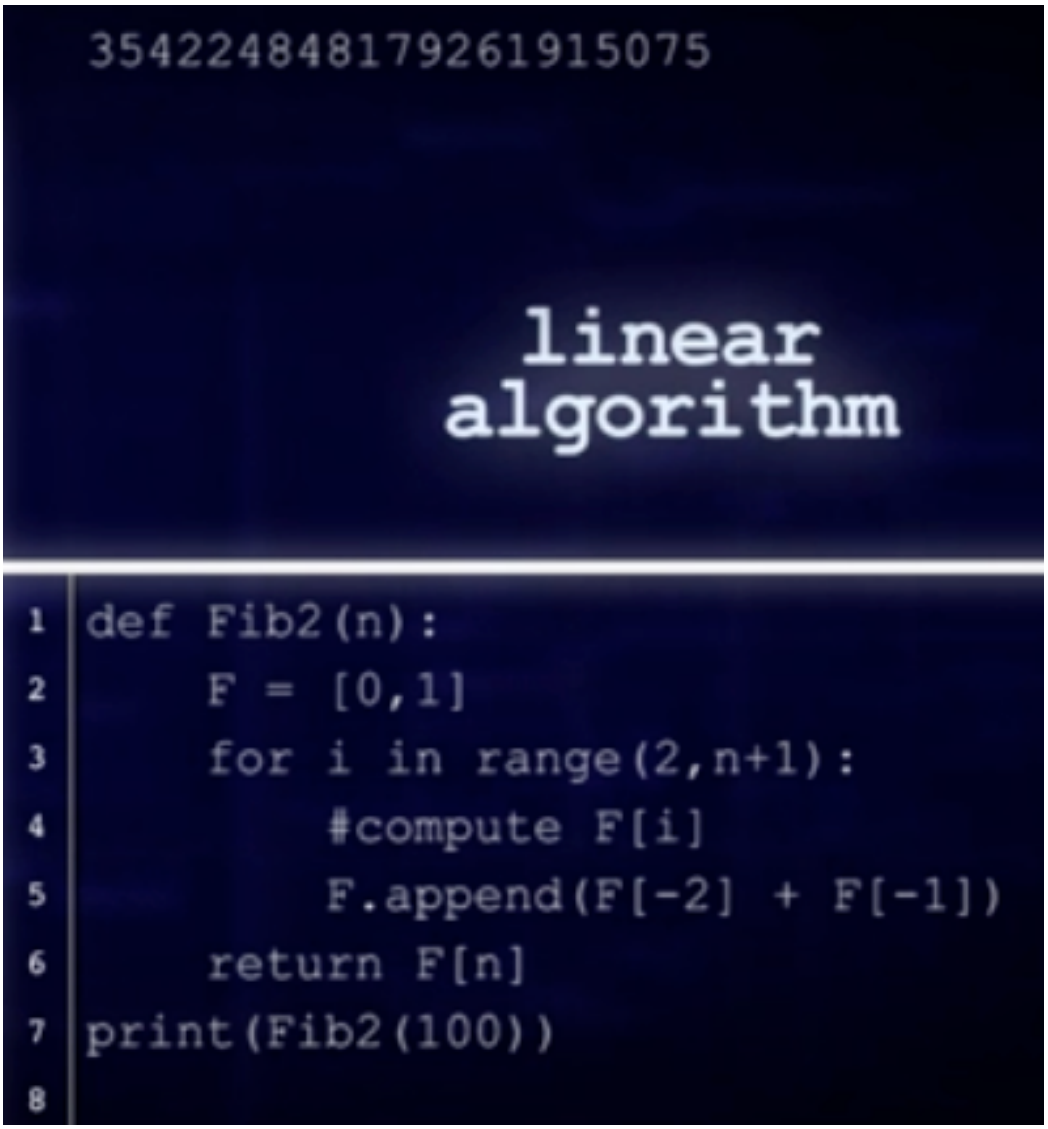

exponential
time

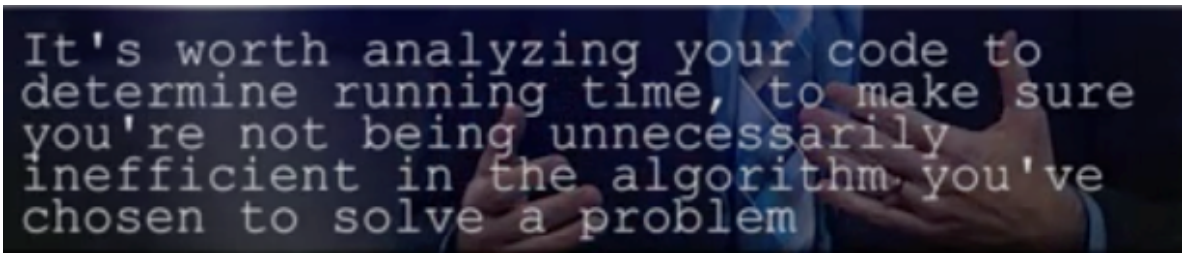

10 million
years


1 Fib(100) =
2 Fib(98) + Fib(99) =
3 (Fib(96) + Fib(97)) + (Fib(97) + Fib(98)) =
4 ((Fib(94) + Fib(95)) + (Fib(95)+ Fib(96))) +
5 ((Fib(95) + Fib(96)) + (Fib(96) + Fib(97)))
6 ...
7
8
9
10
11
12
13
14
15

```
354224848179261915075
```

```
1  def Fib2(n):
2      F = [0,1]
3      for i in range(2,n+1):
4          #compute F[i]
5          F.append(F[-2] + F[-1])
6      return F[n]
7  print(Fib2(100))
8
```


354224848179261915075
linear
algorithm
1 def Fib2(n):
2 F = [0,1]
3 for i in range(2,n+1):
4 #compute F[i]
5 F.append(F[-2] + F[-1])
6 return F[n]
7 print(Fib2(100))
8


It's worth analyzing your code to determine running time, to make sure you're not being unnecessarily inefficient in the algorithm you've chosen to solve a problem