

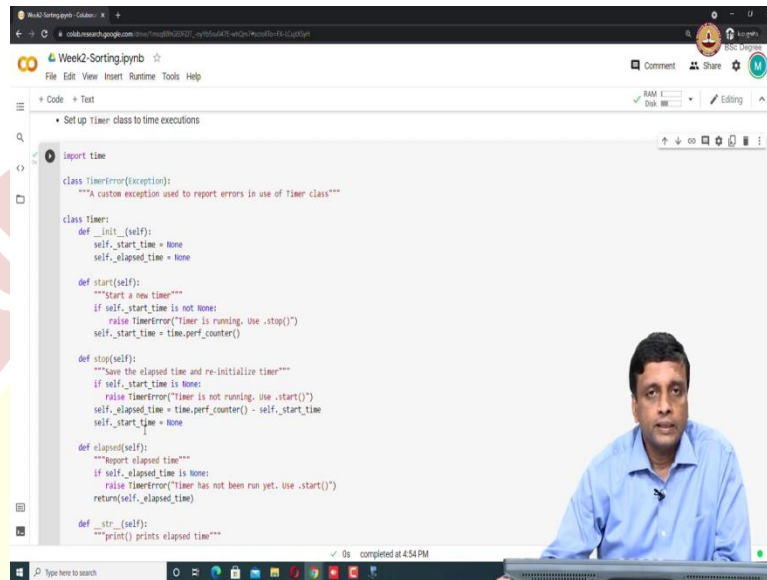


IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms Using Python
Professor. Madhavan Mukund
Implementation of Searching and Sorting Algorithms

(Refer Slide Time: 0:09)



The screenshot shows a Jupyter Notebook window titled 'Week2-Sorting.ipynb'. The code defines a custom exception class 'TimerError' and a 'Timer' class. The 'Timer' class has methods for starting, stopping, and getting the elapsed time. A small video inset of Professor Madhavan Mukund is visible in the bottom right corner of the notebook window.

```
import time

class TimerError(Exception):
    """A custom exception used to report errors in use of Timer class"""

class Timer:
    def __init__(self):
        self._start_time = None
        self._elapsed_time = None

    def start(self):
        """Start a new timer"""
        if self._start_time is not None:
            raise TimerError("Timer is running. Use .stop()")
        self._start_time = time.perf_counter()

    def stop(self):
        """Save the elapsed time and re-initialize timer"""
        if self._start_time is None:
            raise TimerError("Timer is not running. Use .start()")
        self._elapsed_time = time.perf_counter() - self._start_time
        self._start_time = None

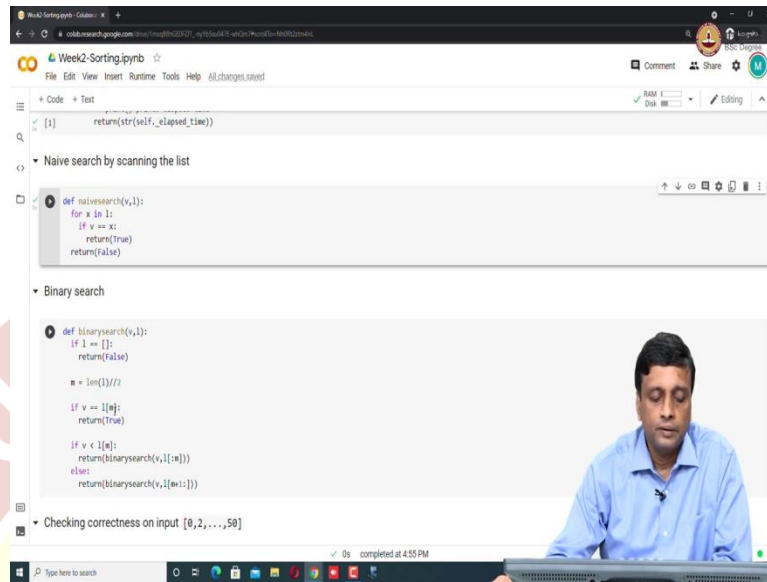
    def elapsed(self):
        """Report elapsed time"""
        if self._elapsed_time is None:
            raise TimerError("Timer has not been run yet. Use .start()")
        return(self._elapsed_time)

    def __str__(self):
        """print() prints elapsed time"""
```

So, we have seen a lot of searching and sorting algorithms abstractly. So, let us try and understand in a more concrete setting how the claims that we made about the complexity actually work. So, so here are some experiments that we will run. So, we will start remember, in the beginning, we talked about this timer class, which will be used to measure time.

So, this is just the same class. So, it is called timer error. So, remember, it allows us to start a timer and to stop a timer. So, that is basically, we will use this to show in physical time, how much time execution takes.

(Refer Slide Time: 0:45)



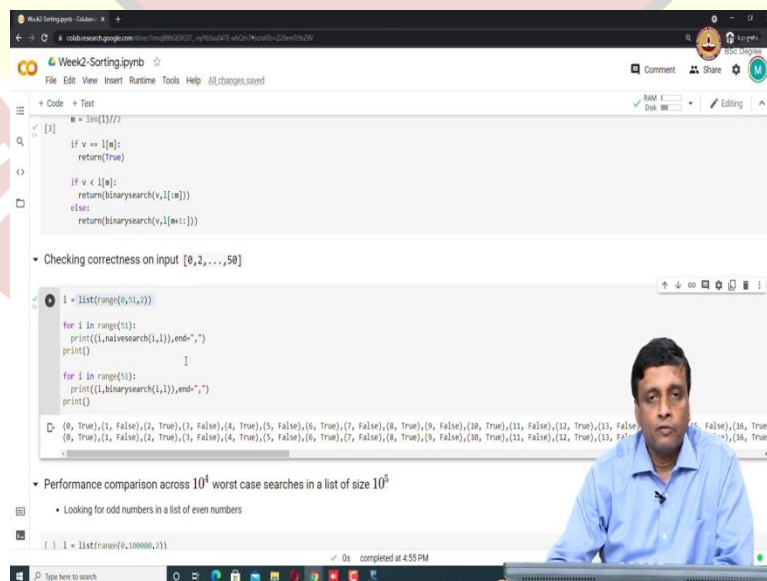
```
def naive_search(v, l):
    for x in l:
        if v == x:
            return True
    return False

def binarysearch(v, l):
    if l == []:
        return False
    m = len(l)//2
    if v == l[m]:
        return True
    if v < l[m]:
        return binarysearch(v, l[:m])
    else:
        return binarysearch(v, l[m+1:])
```

Checking correctness on input [0, 2, ..., 50]

So, the first thing that we were looking at was searching a list. So, we had two strategies, so we had this naive search. So, naive search basically, was one thing, which just scan through the entire list. And whenever it found the item, it would return true otherwise, it would eventually come out from the list saying that the item is not there and return false. So, the worst case is when the item is missing. And then we had binary search, which would basically divide the interval into two and search either the first half of the second half if we did not find.

(Refer Slide Time: 1:15)



```
l = list(range(0, 51, 2))

for i in range(51):
    print((i, naive_search(i, l)), end=", ")
    print()

for i in range(51):
    print((i, binarysearch(i, l)), end=", ")
    print()
```

Performance comparison across 10^4 worst case searches in a list of size 10^5

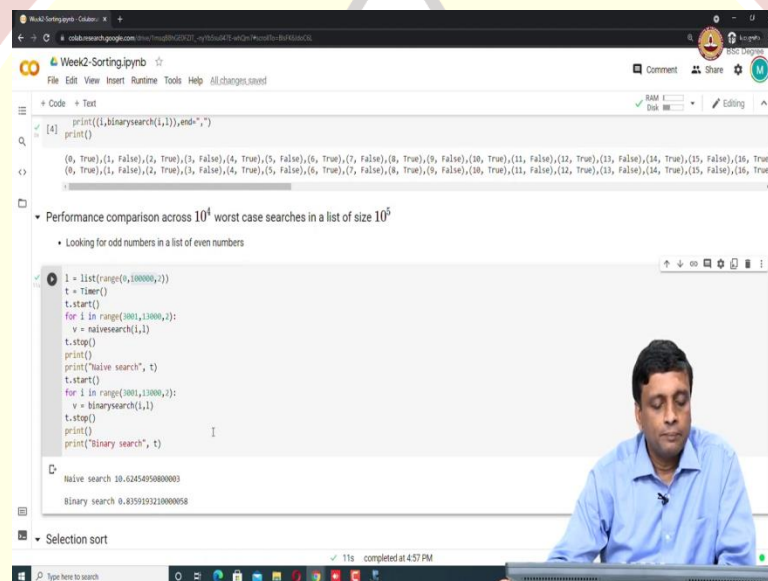
- Looking for odd numbers in a list of even numbers

```
l = list(range(0, 100000, 2))
```

So, let us just check that these two implementations are correct by looking at some trivial example. So, we just set up a list here, which is the list of all the even numbers from 0 to 50. So, now, I searched then for all the numbers from 0 to 50. So, it should say that all the even numbers are found and all the odd numbers are not found, which is indeed the case.

So for 0, naive search; so the first line is naive search. Second line is binary search. So 0, 2, 4, 6, 8 are true 1, 3, 5, 7 are false. So, this is just to convince myself that I have not made any silly mistake in writing those two functions. So, now what I really want to do is compare the performance of these two algorithms.

(Refer Slide Time: 1:59)



```
print(binarysearch(1,1),end=" ")
print()

(0, True), (1, False), (2, True), (3, False), (4, True), (5, False), (6, True), (7, False), (8, True), (9, False), (10, True), (11, False), (12, True), (13, False), (14, True), (15, False), (16, True), (17, False), (18, True), (19, False), (20, True), (21, False), (22, True), (23, False), (24, True), (25, False), (26, True), (27, False), (28, True), (29, False), (30, True), (31, False), (32, True), (33, False), (34, True), (35, False), (36, True), (37, False), (38, True), (39, False), (40, True), (41, False), (42, True), (43, False), (44, True), (45, False), (46, True), (47, False), (48, True), (49, False), (50, True)

Performance comparison across 104 worst case searches in a list of size 105
• Looking for odd numbers in a list of even numbers

l = list(range(0,100000,2))
t = Timer()
t.start()
for i in range(1001,13000,2):
    v = naiveSearch(l,i)
t.stop()
print()
print("Naive search", t)
t.start()
for i in range(1001,13000,2):
    v = binarysearch(l,i)
t.stop()
print()
print("Binary search", t)

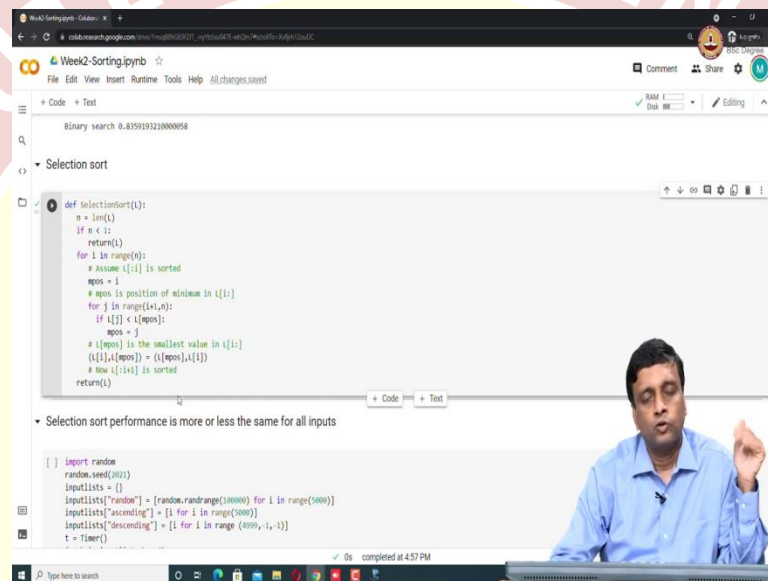
Naive search 10.6245495080003
Binary search 0.835919121000058
```

So, one way to compare it is to run it on large inputs, and also to run it multiple times. So, just for the interests of making sure that this does not take too long, I am going to run 10 to the power 4 Worst case searches in a list of size 10 to the power 5. So, what is the worst case search, it is going to be a search, which is not going to work.

So here, I am going to take 10 to the power of 5. So, I am taking notice here 10 to the power 5, I am taking all the even numbers between 0 and 10 to the power 5. So, this is actually half of 10 to the power 5. And then I am going to take a bunch of odd numbers 1000 odd numbers from 3000 to 13000. And I am going to search for them in this list. And for every search thing, I am going to start a timer, then search and then stop it and then print the timer.

So first, I will do it for naive search. And then I will do it for binary search. So, let us run this thing. So, it takes some amount of time. The naive search for this takes 10.6 seconds. And binary search takes less than one second. So, there is a factor of 10 speed up, it should possibly be more dramatic than this. But at least it is clear that binary search is much better than naive search. And so we will come back to this issue about why binary search is not as good as it seems, compared to naive search, but naive, which is certainly worse.

(Refer Slide Time: 3:31)



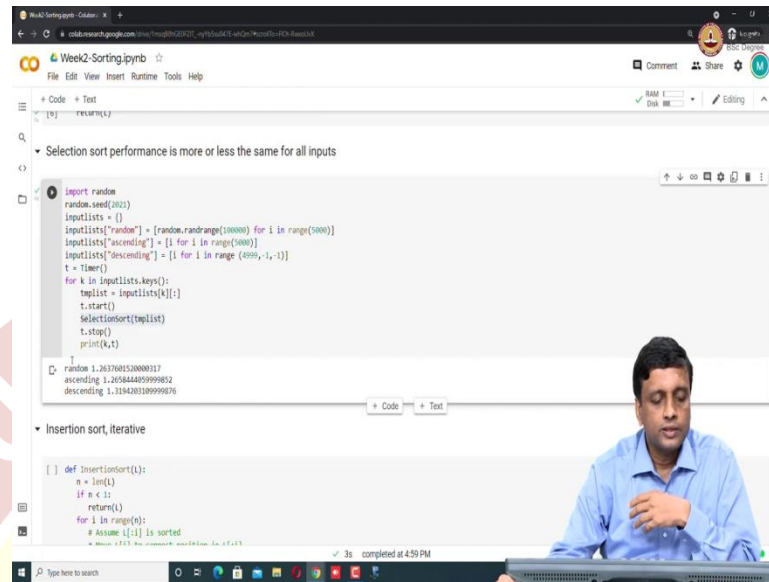
```
def selectionsort(l):
    n = len(l)
    if n <= 1:
        return l
    for i in range(n):
        # assume l[i] is sorted
        mpos = i
        # mpos is position of minimum in l[i:]
        for j in range(i+1, n):
            if l[j] < l[mpos]:
                mpos = j
        # l[mpos] is the smallest value in l[i:]
        (l[i], l[mpos]) = (l[mpos], l[i])
        # now l[i:] is sorted
    return l
```

```
[ ] import random
random.seed(2021)
inputlists = []
inputlists["random"] = [random.randrange(10000) for i in range(5000)]
inputlists["ascending"] = [i for i in range(5000)]
inputlists["descending"] = [i for i in range(4999, -1, -1)]
t = Timer()
```

So, next thing we looked at was some naive sorting algorithms. So, the first naive sorting algorithm was selection sorts. So, this is an implementation of selections or so remember, in selection sort, say we are doing it in ascending order, we take the we find the minimum element and move it to the beginning. So, we assume that we have sorted up to position i minus 1, we take the value at i onwards find the minimum and put it at position i .

So, this is selection sort. So, this scanning and finding the minimum is not going to change whether or not that suffix of the list or sequence is sorted on. So, whether it is in random order, or ascending order or so on does not matter. So, let us set up an experiment with that.

(Refer Slide Time: 4:13)



```
Week2-Sorting.ipynb
File Edit View Insert Runtime Tools Help

+ Code + Text
[1]: return(x)
```

Selection sort performance is more or less the same for all inputs

```
import random
random.seed(2021)
inputlists = {}
inputlists["random"] = [random.randrange(100000) for i in range(5000)]
inputlists["ascending"] = [i for i in range(5000)]
inputlists["descending"] = [i for i in range(4999, -1, -1)]
t = Timer()
for k in inputlists.keys():
    tmp_list = inputlists[k][:]
    t.start()
    SelectionSort(tmp_list)
    t.stop()
    print(k,t)
```

```
random 1.2637605120000117
ascending 1.205644899999982
descending 1.2184201899999876
```

Insertion sort, iterative

```
[ ] def InsertionSort(l):
    n = len(l)
    if n < 2:
        return(l)
    for i in range(n):
        # Assume l[i:] is sorted
        # Insert l[i] into the sorted sequence
```

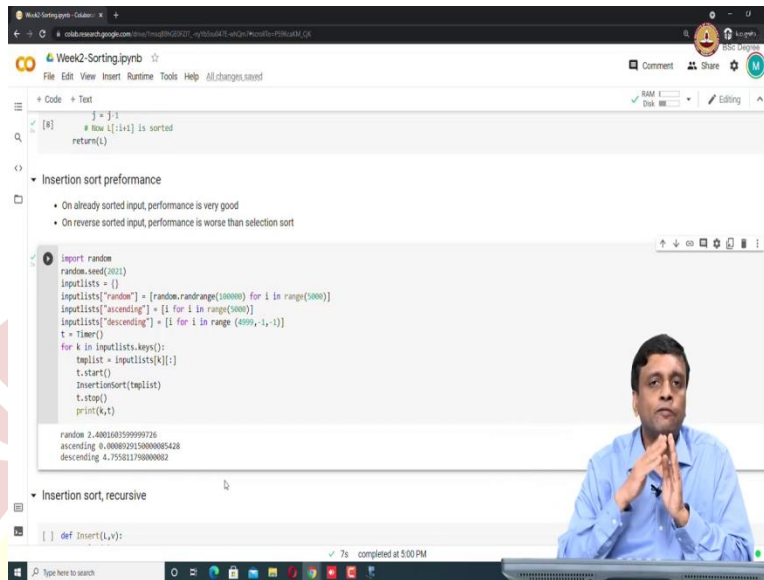
3s completed at 4:59 PM

So, what we do is we take 5000 elements, and we create 5000 random elements. So, we start with a random library of Python and what this says is, it creates for i in range 5000. So, it creates 5000 random elements between 0 and 10 to the power 5, so I am basically creating a random list of size 5000. And then I am creating an ascending list of 5000, which is 0 to 4999.

And then I am creating a descending list of 5000, which is 4999 to 0. So, I create these three lists, and I put each of them in a dictionary called input lists with the key random ascending and descending. And now what I do is I put this whole thing in the loop. So, I have this timer which I create, and then for each of these random ascending descending, I first make a copy of the list, I do not want to change it remember that these are in place sort.

So, I just take a slice. And then I started the timer, run the selection sort on this list that I am passing it, stop it and print it. So, if I run this, then you will see that I have chosen numbers to run this fast. So, each of them takes about a second, but they take the same amount of time, this is the important thing. So, if it is random or ascending or descending, it does not matter, they all take roughly as you can see, 1.2 seconds or so.

(Refer Slide Time: 5:41)



```
def insertion_sort(L):
    for i in range(1, len(L)):
        j = i - 1
        while j > 0 and L[j] > L[j + 1]:
            L[j], L[j + 1] = L[j + 1], L[j]
            j -= 1
    return L
```

Insertion sort performance

- On already sorted input, performance is very good
- On reverse sorted input, performance is worse than selection sort

```
import random
random.seed(1021)
inputlists = []
inputlists["random"] = [random.randrange(100000) for i in range(1000)]
inputlists["ascending"] = [i for i in range(10000)]
inputlists["descending"] = [i for i in range(9999, 0, -1)]
t = Timer()
for k in inputlists.keys():
    taplist = inputlists[k][:]
    t.start()
    insertion_sort(taplist)
    t.stop()
    print(k, t)
```

random 2.4001682599999726
ascending 0.00002915000005428
descending 4.755811700000002

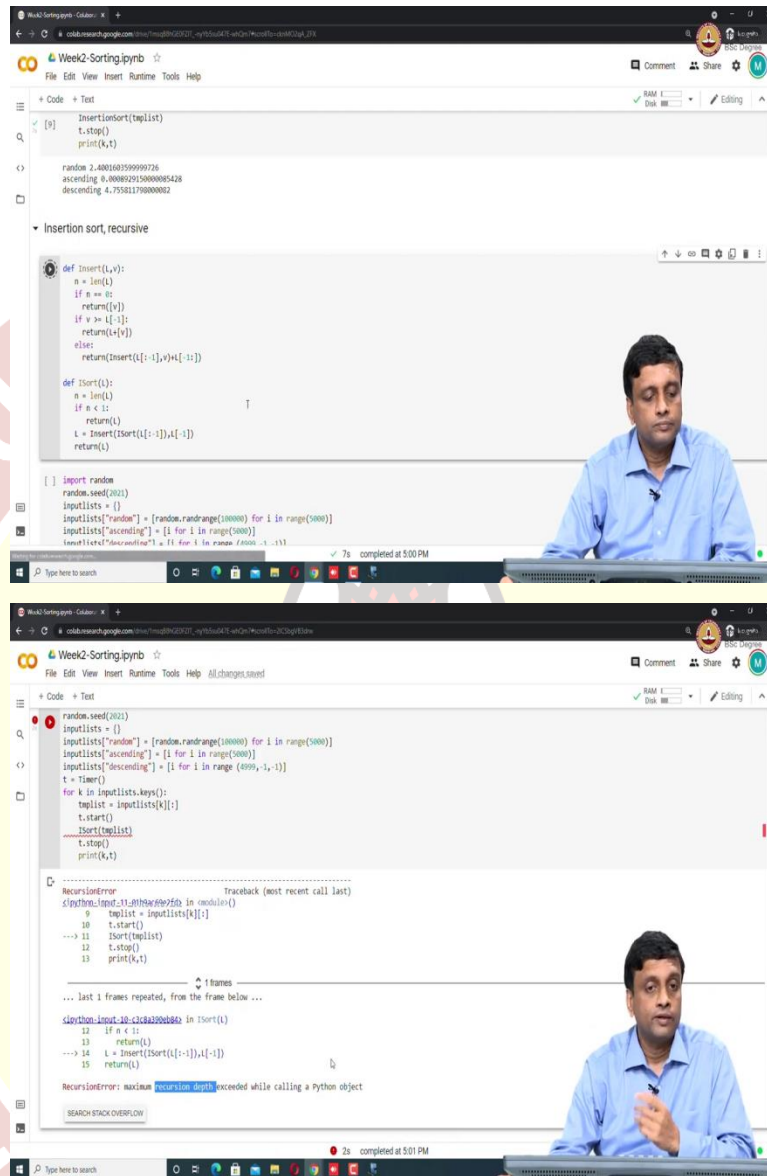
Insertion sort, recursive

```
def Insert(L, v):
```

Now, here is Insertion Sort, which was another naive sort that we looked at. And in insertion sort, we said that the performance actually should vary according to whether it is ascending or not. So, if we run insertion sort on the same thing, we would expect that it will work better on the sorted arrays than on the unsorted arrays.

So indeed, if I, if I run insertion sort on the same random elements, on the random elements, it actually takes a little longer than selections sort, there it took 1.2 seconds here is taking 2 seconds. But on ascending order, it takes a fraction of a second, because an ascending order, the insert operation is trivial. In any descending order, it is taking something like 5 seconds, so it is actually much worse, because in descending order, essentially whenever we are taking the minimum element, we have to insert it all the way to the back, because each element that we see is smaller than everything before it. So, we have to the maximum number of inserts.

(Refer Slide Time: 6:38)



The image displays two screenshots of a Jupyter Notebook titled 'Week2-Sorting.ipynb'. The top screenshot shows the implementation of a recursive Insertion Sort. The code defines a function `Insert(L, v)` that inserts an element `v` into a list `L` at the correct position, and a function `ISort(L)` that recursively sorts the list. The notebook also shows the generation of random lists and the execution of the sorting function, which completes successfully at 5:00 PM.

The bottom screenshot shows the same notebook after running the sorting function on a larger list. A `RecursionError` is raised, indicating that the maximum recursion depth has been exceeded. The error message states: 'RecursionError: maximum recursion depth exceeded while calling a Python object'. The traceback shows the call stack leading to the `ISort` function.

What about a recursive Insertion Sort? So, a recursive Insertion Sort. Now will, if we run it on the same inputs, we first see something interesting, which is that we will get an error, which is because of something called the recursion depth. So, you may have seen this before in the Python course.

But Python by default has a very small tolerance for recursion, because it is worried that you are out of control. So typically, the recursion depth is. So, you can have about 1000 nested calls, which is not enough for Insertion Sort sorting, say an ascending or descending order list where it has to keep doing something for, say 5000 elements.

(Refer Slide Time: 7:26)

The image displays two screenshots of a Jupyter Notebook titled "Week2-Sorting.ipynb". The notebook is open in a web browser, showing code cells and output. A video overlay of a man in a blue shirt is present in the center of the notebook interface.

Top Screenshot:

- Setup**
 - Set recursion limit to $2^{31} - 1$
 - This is the highest value Python allows
- Recursive insertion sort is slower than iterative**
 - Input of 2000 (40%) takes more time than 5000 for iterative
 - Overhead of recursive calls
 - Performance pattern between unsorted, sorted and random is similar
- Code Cell:**

```
import sys
sys.setrecursionlimit(2**31-1)
```
- Code Cell:**

```
import random
random.seed(4021)

inputlists = []
inputlists["random"] = [random.randrange(100000) for i in range(2000)]
inputlists["ascending"] = [i for i in range(2000)]
inputlists["descending"] = [i for i in range(1999, -1, -1)]
t = Timer()
for k in inputlists.keys():
    taptlist = inputlists[k][:]
    t.start()
    Sort(taptlist)
    t.stop()
    print(k,t)
```
- Output:** 0s completed at 5:01 PM

Bottom Screenshot:

- Code Cell:**

```
import random
random.seed(4021)

inputlists = []
inputlists["random"] = [random.randrange(100000) for i in range(2000)]
inputlists["ascending"] = [i for i in range(2000)]
inputlists["descending"] = [i for i in range(1999, -1, -1)]
t = Timer()
for k in inputlists.keys():
    taptlist = inputlists[k][:]
    t.start()
    Sort(taptlist)
    t.stop()
    print(k,t)
```
- Output:**

```
random 11.972421354000007
ascending 0.13484728400004107
descending 20.875667678000013
```
- Merge sort**
 - Code Cell:**

```
def merge(A,a):
    (n,n) = (len(A),len(a))
    (i,j,k) = (0,0,0)
    while k < n+n:
        if i == n:
            C.extend(a[j:])
            k = k + (n-j)
        elif j == n:
```
- Output:** 32s completed at 5:02 PM

```
Week2-Sorting.ipynb
File Edit View Insert Runtime Tools Help
Code Text
On reverse sorted input, performance is worse than selection sort

import random
random.seed(1021)
inputlists = []
inputlists["random"] = [random.randrange(100000) for i in range(5000)]
inputlists["ascending"] = [i for i in range(5000)]
inputlists["descending"] = [i for i in range(4999, -1, -1)]
t = Timer()
for k in inputlists.keys():
    taptlist = inputlists[k][:]
    t.start()
    InsertionSort(taptlist)
    t.stop()
    print(k,t)

random 3.4801610559999726
ascending 0.000829550000085428
descending 4.755811780000082

Insertion sort, recursive

[10] def Insert(L,v):
    n = len(L)
    if n == 0:
        return([v])
    if v >= L[-1]:
        return(L+[v])
    else:
        return(Insert(L[:-1],v)+L[-1:])

def InsertionSort(L):
```

So, one way to fix that is to set this thing called the recursion limit. And it turns out that the largest value you are allowed to set in Python is this number 2 to the power 31 minus 1 . So, you can do this thing of setting the recursion limit by importing this system library and calling it with this number.

So, once we do this, then we can run our Insertion Sort as before. And now I am running it on even smaller lists. So there, initially I was running into on a 5000 size list. Now I am running it on a 2000 size list, which is only 40 percent. If you remember insertion sort on the iterative version took 2 seconds.

Now here on something which is much smaller, it is taking 12 seconds. And there, if you remember, the insertion sort on the sorted version took really a very small amount like $1/10000$ th of a second. And here it is taking almost $1/10$ of a second. So, what this is saying is that, although they are equivalent as algorithms, the recursion basically makes the whole thing much slower. Because every time we have to call and return from a function, there is an extra cost involved. So, our basic counting is not working as well when we have to deal with these function calls.

(Refer Slide Time: 8:43)

```
[14]: def merge(A, B):
    (n, m) = (len(A), len(B))
    (C, i, j, k) = ([], 0, 0, 0)
    while k < n+m:
        if i == n:
            C.extend(B[j:])
            k = k + (n-j)
        elif j == m:
            C.extend(A[i:])
            k = k + (m-i)
        elif A[i] < B[j]:
            C.append(A[i])
            (i, k) = (i+1, k+1)
        else:
            C.append(B[j])
            (j, k) = (j+1, k+1)
    return C

def mergesort(A):
    n = len(A)
    if n <= 1:
        return A
    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])
    S = merge(L, R)
    return S
```

A simple input to check correctness

```
983,
984,
985,
986,
987,
988,
989,
990,
991,
992,
993,
994,
995,
996,
997,
998,
999]
```

Performance on large inputs, 10^6 , random and sorted

```
[ ] import random
random.seed(42)
inputlists = []
inputlists["random"] = [random.randrange(100000000) for i in range(100000)]
inputlists["ascending"] = [i for i in range(100000)]
inputlists["descending"] = [i for i in range(999999, -1, -1)]
t = Timer()
for k in inputlists.keys():
    taptlist = inputlists[k]:
    t.start()
    mergesort(taptlist)
    t.stop()
    print(k, t)
```

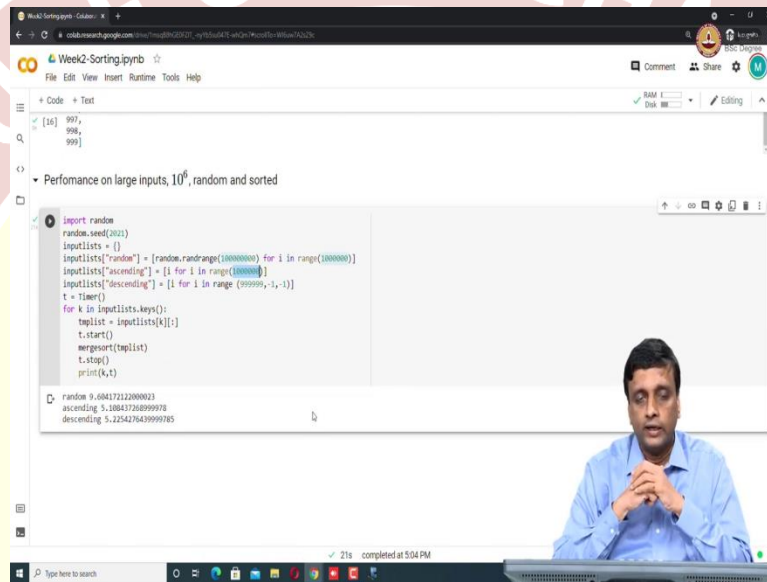
And finally, we have merge sort, so we have this merge function. And then we have merge sort, which calls this merge function. And we said that merge sort is an order $n \log n$ insert, so it should work much better on even large inputs. So again, we can run it on a trivial thing. So, we just take sorting the even numbers from 0 to 1000, followed by the odd numbers from 0.

So, we construct a list which is 0, 2, 4 followed by 1, 3, 5, 7. And we sorted and indeed Merge Sort produces a sorted output, but our real interest is in the performance. So, supposing now we take a seriously large array which we have not tried at all with insertion sort and selection sort, if

you try it, you will find it takes forever, but 10 to the power 6 , remember, if I do $n \log n$ and 10 to the power 6 , $\log n$ of 10 to the power 3 is about 10 .

So, $n \log n$ on 10 to power 6 is going to be more than 10 to the power 7 . And we said that 10 to the power 7 is what we would expect in 1 second. So, $n \log n$ should take between 1 second and 10 seconds is what we would estimate.

(Refer Slide Time: 9:41)



```
import random
random.seed(42)
inputlists = {}
inputlists["random"] = [random.randrange(1000000) for i in range(1000000)]
inputlists["ascending"] = [i for i in range(1000000)]
inputlists["descending"] = [i for i in range(999999, -1, -1)]
t = Timer()
for k in inputlists.keys():
    tlist = inputlists[k]
    t.start()
    mergesort(tlist)
    t.stop()
    print(k,t)
```

random 9.684172122000023
ascending 5.180437268999978
descending 5.2234276439999785

So, if we run this again on our three types of inputs, random ascending and descending, but this time the numbers are 10 to the power 6 and not those 5000 which we are doing with Insertion Sort, then you will find that indeed, the random one takes a little less than 10 seconds and the ascending and descending will take about half the time. So, this shows that merge sort is indeed, significantly faster than the naive sorts working in $n \log n$ time.